Universität Ulm
Institut für Datenbanken und Informationssysteme
(Leiter: Prof. Dr. P. Dadam)

OBJECT-AWARE PROCESS MANAGEMENT

DISSERTATION

zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Ingenieurwissenschaften und Informatik
der Universität Ulm

vorgelegt von

VERA KÜNZLE (GEB. PFEIFFER)
aus Herbrechtingen

APRIL 2013

Amtierender Dekan:	Prof. Dr.-Ing. K. Dietmayer
Gutachter:	Prof. Dr. M. Reichert
	Prof. Dr. H. Partsch
	Prof. Dr. S. Rinderle-Ma
Tag der Promotion:	16. Juli 2013

Vorwort

Diese Dissertation entstand im Rahmen einer Kooperation der Abteilung Datenbanken und Informationssysteme (DBIS) der Universität Ulm mit der Persis GmbH in Heidenheim, bei der ich seit 2005 angestellt bin. Parallel zu dieser Anstellung habe ich 2008 meine Promotion begonnen. Die Ausgangsmotivation dafür fand ich im Rahmen meiner praktischen Tätigkeit als Softwareentwicklerin. An dieser Stelle möchte ich mich nun bei allen Personen bedanken, die mich während meiner Promotionszeit unterstützt haben.

Mein größter Dank gilt meinem Doktorvater Prof. Dr. Manfred Reichert, der schon zu Studienzeiten mein Interesse am wissenschaftlichen Arbeiten geweckt hat. Er hat diese Arbeit mit einem außerordentlich tatkräftigen Einsatz unterstützt. Die zahlreichen fachlichen Diskussionen gaben mir stets wertvolle Anregungen und haben maßgeblich zum erfolgreichen Abschluss dieser Arbeit beigetragen. Sein akribisches und zeitaufwendiges Lesen meiner Schriftstücke war unentbehrlich für den Erfolg meiner wissenschaftlichen Publikationen und Vorträge.

Ebenso bedanke ich mich sehr herzlich bei den Geschäftsführern der Persis GmbH Detlef Hergesell, Rainer Kolb und Michael Barth, die mir die Promotion parallel zu meiner praktischen Tätigkeit ermöglicht und finanziell unterstützt haben. Auch danke ich Adrian Kluge, der als Entwicklungsleiter mit das Ziel verfolgt, dass die erarbeiteten Konzepte produktiv zum Einsatz kommen werden.

Weiter danke ich Prof. Dr. Helmut Partsch und Prof. Dr. Stefanie Rinderle-Ma für die Übernahme der Zweitgutachten und ihr Interesse am Thema. Prof. Dr. Peter Dadam danke ich für seine Unterstützung bei Vorträgen und die Übernahme von Gutachten für die von mir betreuten Abschlussarbeiten.

Ferner danke ich allen DBIS-Mitarbeitern für das tolle Arbeitsumfeld, das ich als Externe dort vorgefunden habe. Die Diskussionen mit Euch waren nicht nur sehr hilfreich und motivierend, sondern haben auch enorm großen Spaß gemacht. Erwähnen möchte ich Rüdiger Pryss für die wegweisenden, freundschaftlichen Ratschläge, Andreas Lanz, der als Tex-Profi mit seinen Tipps eine große Hilfe war, sowie Carolina Chiao, die das Thema weiter bearbeitet. Des Weiteren möchte ich mich bei Julian Tiedeken sowie bei meinen studentischen Hilfskräften Nicole Wagner, Christian Scheb, Andreas Pröbstle, Hannes Beck, Stefan Schultz und Thomas Spindler für ihre Unterstützung bei der prototypischen Realisierung der Konzepte bedanken. Gleichermaßen danke ich jedoch auch meinen Kollegen bei der Persis GmbH. Ihre Rücksichtnahme war sehr wertvoll, damit ich die Promotion mit meiner Arbeit dort vereinbaren konnte.

Zu guter Letzt bedanke ich mich bei meiner Familie. Meinem Mann Oliver danke ich für seine Unterstützung, sein Verständnis und seine Geduld. Ohne diesen Rückhalt hätte ich die notwendige Motivation und Disziplin über diesen Zeitraum hinweg nicht aufbringen können. Wesentliche Teile dieser Dissertation sind während meiner Elternzeit - im ersten Lebensjahr meiner Tochter Madita - entstanden. Ich danke von ganzem Herzen meinen Eltern und Schwiegereltern Anneliese Pfeiffer, Hannelore Künzle, Stephan Pfeiffer und Gerd Künzle für die ausdauernde, flexible und vor allem liebevolle Betreuung von Madita. Ebenso danke ich meiner Oma Anna Mäule für ihre tatkräftige Unterstützung seit ich auf der Welt bin. Ihr alle habt es mir ermöglicht, trotz der vielen Arbeit immer bei Madita sein zu können, wenn sie mich gebraucht hat. Dadurch konnte diese Arbeit überhaupt erst fertig gestellt werden. Madita - Dein erstes Lebensjahr war mit Sicherheit das anstrengendste in meinem bisherigen Leben - aber zugleich das wunderschönste!

Abstract

Companies increasingly adopt process management systems (PrMS), which offer promising perspectives for a more flexible and efficient process execution. However, for many process-aware application systems (e.g., ERP or CRM systems), the underlying process logic is still hard-coded. As a consequence, these business applications are both complex to design and costly to maintain; i.e., they require long development cycles, and even simple process changes might result in costly code adaptions and high efforts for testing.

One reason for this situation stems from the fact that contemporary PrMS were primarily designed for the support of highly structured, repetitive business processes. By contrast, many processes found in practice are rather *unstructured* or *semi-structured*, i.e., they are *knowledge-intensive* and driven by *user decisions*. In addition, the business functions to be integrated with these processes usually cannot be straight-jacketed into activities. For all these reasons, the *activity-centered paradigm* of contemporary PrMS is by far too inflexible for realizing more advanced business applications. This deficiency mainly stems from the unsatisfactory integration of processes and data in existing PrMS. Despite emerging approaches, which target at a tighter integration of process and data, a unified and comprehensive understanding of the relationships between them is still missing.

This thesis first analyzes real processes not adequately supported by existing PrMS and elaborates their characteristic properties. As a major insight it became clear that in many application scenarios comprehensive process support requires both *object-* and *process-awareness*. This means, business processes and business data must not be treated independently from each other. Instead, business processes must comply with the underlying data structure. In particular, in accordance to the given data model comprising object types and object relations, the modeling, execution and monitoring of processes must be based on two levels of granularity: *object behavior* and *object interactions*. Further, the individual processes, coordinating the behavior of single object instances, must be coordinated with the ones of related object instances. Opposed to these well-defined process support granularity levels, *activities must be executable at different levels of granularity*. In particular, while a particular user may only want to work on a particular object instance, another one may want to process a number of related object instances in one go. Furthermore, process execution must be accomplished in a *data-driven* manner; i.e., the progress of a process instance mainly depends on available business objects and on value changes of their attributes. Finally, authorized users must be able to access and manage process-related objects at any point in time.

Based on the properties identified, this thesis elaborates major requirements for enabling object-awareness in processes management systems. The major contribution of the thesis is the *PHILharmonicFlows framework*, which addresses these requirements and enables object-aware process management in a comprehensive manner. In particular, the framework not only provides a *new process modeling approach*, but also establishes a *well-defined operational semantics* enabling the automatic and dynamic generation of end-user components for object- and process-aware business applications at run-time (e.g., overview tables and user forms).

Overall, a holistic framework integrating data, processes and users offers promising perspectives in order to overcome the numerous limitations of contemporary PrMS. This thesis considers research in this area as fundamental maturation of process management technology.

Kurzfassung

Trotz der Mächtigkeit heutiger Prozess-Management-Systeme (PrMS) gibt es in Unternehmen noch zahlreiche Prozesse, die technologisch nicht adäquat unterstützt werden. Existierende PrMS basieren auf einem aktivitätsorientierten Paradigma, bei dem Prozesse anhand von Aktivitäten modelliert werden, die dann mit bestehenden Anwendungen zu verknüpfen sind. Die Verwaltung der Anwendungsdaten dagegen erfolgt dabei allerdings außerhalb des PrMS in den integrierten Anwendungen.

Diese strikte Trennung von Prozessen und Daten hat sich für viele prozessorientierte Anwendungen jedoch als zu inflexibel und eingeschränkt erwiesen. Insbesondere fehlt ein Verständnis der inhärenten Zusammenhänge dieser beiden Perspektiven eines Informationssystems. Der erste Teil dieser Dissertation beinhaltet eine detaillierte Untersuchung der grundlegenden Beziehungen zwischen Prozessen, Daten, Funktionen und Benutzern. Diese hat gezeigt, dass viele Prozesse objektzentriert (engl. object-aware) sind. Objekte besitzen Bearbeitungsprozesse, die koordiniert werden müssen und eine datengetriebene Ausführung erfordern. Letzteres bedeutet, dass der Bearbeitungsfortschritt zur Laufzeit nicht von den ausgeführten Aktivitäten, sondern von den jeweiligen Attributwerten eines Objekts abhängt. Innerhalb eines Geschäftsprozesses müssen dann zahlreiche solcher objektspezifischen Prozesse koordiniert werden, wobei die unterschiedlichen Beziehungen der Objekte zueinander zu berücksichtigen sind. Dadurch entsteht zur Laufzeit, analog zur Datenstruktur, eine komplexe Prozessstruktur. Bei der Ausführung von Aktivitäten dagegen können einzelne Objekte oder ganze Objektmengen gleichzeitig bearbeitet werden. Somit ist eine Aktivität nicht mehr notwendigerweise eindeutig einer bestimmten Prozessinstanz zuzuordnen und dadurch flexibler ausführbar.

Die mangelnde Kontrolle heutiger PrMS über die Anwendungsdaten hat zwei grundlegende Nachteile. Erstens bietet das ausschließlich aktivitätsorientierte Paradigma nicht die Mächtigkeit, objektzentrierte Prozesse adäquat zu unterstützen. Zweitens bieten PrMS den Endbenutzern zur Laufzeit lediglich eine prozessorientierte Sicht mittels Arbeitslisten. Eine daten- und funktionsorientierte Sicht dagegen fehlt. Dadurch ist es nicht möglich, Daten unabhängig vom Prozessstatus zu beliebigen Zeitpunkten einzugeben, einzusehen oder zu ändern. Die Folge ist, dass bestehende daten- und funktionsorientierte Anwendungen oftmals durch eine fest kodierte Prozesslogik erweitert werden, was den Anwendungscode unnötig komplex und nur schwer wartbar macht.

Im zweiten Teil der Arbeit wird ein ganzheitlicher, generischer Ansatz beschrieben, der die komplexen Zusammenhänge zwischen Prozessen, Daten, Funktionen und Benutzern innerhalb von Informationssystemen berücksichtigt. Dies bildet die Basis für eine flexiblere Ausführung zur Laufzeit. Dadurch werden viele Limitationen heutiger PrMS überwunden. Das vorgestellte Rahmenwerk basiert auf einem weitreichenden, konzeptionellen Modell mit präziser operationaler Semantik. Es ermöglicht eine datengetriebene Prozesssteuerung, ohne jedoch das aktivitätsorientierte Paradigma vollständig aufzugeben. Zur Laufzeit wird dem Benutzer sowohl eine prozess- als auch eine daten- und funktionsorientierte Perspektive zur Verfügung gestellt. Hierbei steht nicht nur für die Prozesslogik, sondern auch für die Daten- und Funktionslogik eine generische Implementierung zur Verfügung. Dadurch sind Anwendungskomponenten (z.B. Arbeits- und Übersichtslisten, Formulare) zur Laufzeit automatisch aus den Modellbeschreibungen generierbar.

Im Gegensatz zu existierenden Prozess-Management-Paradigmen, bei denen die Granularität von Prozessen und Aktivitäten frei wählbar ist, basiert dieser Ansatz auf einer klaren Methodik. Hierzu wird eindeutig zwischen Objektverhalten und Objektinteraktionen differenziert. Existierende Ansätze zur Beschreibung von Objektverhalten können in zwei Kategorien eingeteilt werden. Ansätze der ersten Kategorie beschreiben eine zustandsbasierte Modellierung, deren Ausführungssemantik rein auf Aktivitäten basiert. Im Gegensatz dazu ermöglichen andere Ansätze zunehmend eine datengetriebene Ausführung, jedoch fehlt eine Berücksichtigung der jeweils zugehörigen Datenzustände. Der in dieser Dissertation entwickelte Ansatz dagegen verbindet erstmalig eine zustandsbasierte Beschreibung von Prozessen mit einer datengetriebenen Ausführungssemantik. Dadurch wird eine Konsistenz zwischen Prozess- und Daten-Zustand gewährleistet. Durch die Berücksichtigung objektspezifischer Prozesse entsteht zur Laufzeit eine sehr komplexe Prozessstruktur, für welche die Anzahl der laufenden Instanzen variabel ist und sich dynamisch verändern kann. In diesem Zusammenhang werden adäquate Abstraktionsmechanismen zur Verfügung gestellt. Betrachtet man in diesem Zusammenhang die Prozesskoordination zur Beschreibung von Objektinteraktionen, bietet das entwickelte Rahmenwerk auch hier bahnbrechende Konzepte. Die Prozesssynchronisation erfolgt nicht mehr über den Austausch von Nachrichten, sondern basiert auf den definierten Bearbeitungszuständen der involvierten Objekte. Dadurch wird einerseits eine weitgehend asynchrone Ausführung einzelner Prozessinstanzen ermöglicht, andererseits stehen weitreichende Konzepte zur Verfügung, die es erlauben, Prozessinstanzen geeignet zu aggregieren. Hierbei wird auch die Kardinalität zwischen den zugehörigen Objektinstanzen berücksichtigt.

Zur Evaluation werden sowohl die Konzepte für die Modellierung als auch Ausführung prototypisch implementiert und anhand von Fallstudien validiert.

Parts of this thesis have been published in the following referred papers:

[KR09b] V. Künzle and M. Reichert. Integrating Users in Object-Aware Process Management Systems: Issues and Challenges. In *Business Process Management Workshops (Proc. BPD'09)*, volume 43 of *LNBIP*, pages 29–41. Springer Berlin Heidelberg, 2009

[KR09c] V. Künzle and M. Reichert. Towards Object-aware Process Management Systems: Issues, Challenges, Benefits. In *Enterprise, Business-Process and Information Systems Modeling (BPMDS'09)*, volume 29 of *LNBIP*, pages 197–210. Springer Berlin Heidelberg, 2009

[KR11b] V. Künzle and M. Reichert. PHILharmonicFlows : Towards a Framework for Object-Aware Process Management. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(4):205–244, 2011

[KR11a] V. Künzle and M. Reichert. A Modeling Paradigm for Integrating Processes and Data at the Micro Level. In *Enterprise, Business-Process and Information Systems Modeling (BPMDS'11)*, volume 81 of *LNBIP*, pages 201–215. Springer, 2011

[KR11c] V. Künzle and M. Reichert. PHILharmonicFlows: Research an Design Methodology. *Technical Report*, 2011

[KR11d] V. Künzle and M. Reichert. Striving for Object-Aware Process Support: How Existing Approaches Fit Together. In *1st Int'l Symposium on Data-driven Process Discovery and Analysis (SIMPDA'11)*, 2011

[Kï1] V. Künzle. Towards a Framework for Object-Aware Process Management. In *1st Int'l Symposium on Data-driven Process Discovery and Analysis (SIMPDA'11), PhD Seminar*, 2011

[KWR10b] V. Künzle, B. Weber, and M. Reichert. Object-aware Business Processes: Properties, Requirements, Existing Approaches. *Technical Report*, 2010

[KWR10a] V. Künzle, B. Weber, and M. Reichert. Object-aware Business Processes: Fundamental Requirements and their Support in Existing Approaches. *International Journal of Information System Modeling and Design (IJISM)*, 2(2):9–46, 2010

Contents

Part I

Motivation

1 Introduction

Companies spend a huge amount of time and money for information technology (IT) to improve their *effectiveness* and *efficiency* [HMPR04]. In this context, *effectiveness* refers to the close alignment of the business goals on one hand and the capabilities offered by IT to achieve these goals on the other. In particular, whether the efforts spent in IT pay off depends first and foremost on the *alignment of IT* with *business perspectives* (cf. Fig. 1.1); i.e., *business data*, *business functions*, and *business processes* [MRB08]. To enable such *business-IT-alignment*, therefore, IT has "to do the right things" in order to be effective [Dru67]. In turn, *efficiency* means "to do the things right" [Dru67]. More and more, the role of IT as enabler for staying competitive at the market is understood (cf. Fig. 1.2). First, for any company it is of almost importance to introduce new products and services as quickly as possible at the market. Second, in order to cope with the increasing competitive pressure and market dynamics, it is crucial to be able to continuously adapt IT in a quick and effective way to changing needs. Consequently, *rapid development* and *improved maintenance* constitute important success factors to foster *efficiency*. To realize them, the development approach (i.e., programming, modeling or configuration) chosen for realizing IT system plays a fundamental role.

According to [Dvt05], IT relies on several layers (cf. Fig. 1.1). As basic layer, the *system infrastructure* consists of hardware, operating system, and other system software. The second layer refers to *generic applications* that may be applied in a wide range of business domains. However, generic applications typically focus on a particular business perspective while neglecting the others. For example, *Database Management Systems (DBMS)* support the management of business data. In turn, *Process Management Systems (PrMS)* support the specification, execution, and monitoring of business processes. Both DBMS and PrMS allow specifying the required data and process logic through *modeling* rather than programming. Furthermore, business functions are supported by specialized tools like text editors, mail clients, or spreadsheet tools. Layers three and four of the IT stack consist of *domain-specific* and *tailor-made applications* (e.g., purchase order management, production planning, and human resource management). Usually, a *tailor-made application* is developed for a specific organization. Hence, a tailor-made application requires a customer-specific *programming* of its functionalities. By

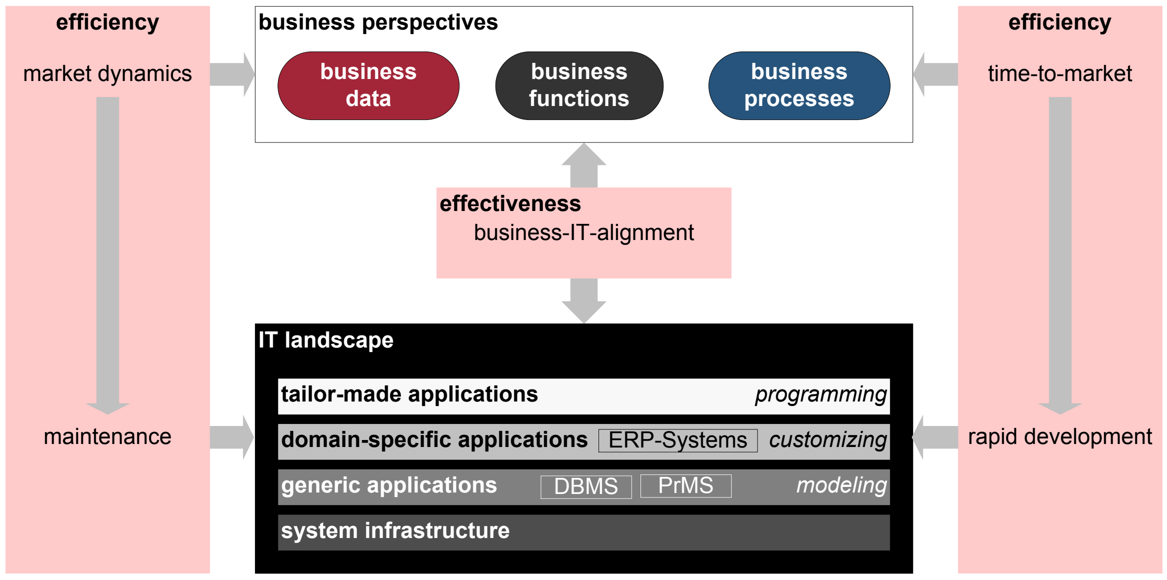


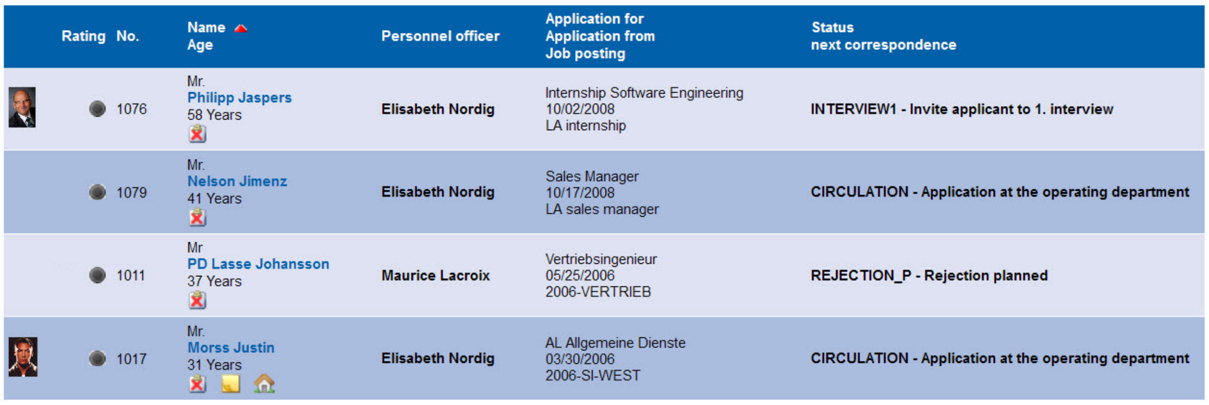
Figure 1.1: IT landscape and business perspectives

contrast, *domain-specific applications* comprise software solutions pre-implementing standard functionality for specific types of organizations. Examples include *Enterprise Resource Planning Systems (ERP)* and *Customer Relationship Management Systems (CRM)*. These applications can then be *customized* to the specific needs of an organization. Usually, such a customizability is realized through configuration support (i.e., it is based on parameter settings). In the following, we use the term *application* when meaning domain-specific as well as tailor-made applications.

1.1 Context of Object-Aware Processes

In current organizations both, domain-specific and tailor-made applications exist for most business divisions. They support enterprises in managing their core assets, including customers, staff, products, and resources [Wes07]. Examples include applications for human resource management (HR), purchase order management, and production planning. Usually, these applications allow accessing *business data* and offer a variety of *business functions* to their users. For this purpose, *data-oriented views* are provided that enable users to access and change business data at any point in time. Typically, these views offer *overview tables* for which each row corresponds to a particular business object [KR09c]. As example consider the overview table listing different job applications as implemented in an existing Human Resource Management System (cf. Fig. 1.2).

In addition to data access, various business functions are provided for accessing, changing, and managing business data. When invoking such business functions (e.g., editing an application, initiating a review), business objects (e.g., applications, job offers, interviews) may be created, deleted, or updated; i.e., attribute values may be read or written by authorized users.



	Rating	No.	Name / Age	Personnel officer	Application for / Application from / Job posting	Status / next correspondence
	●	1076	Mr. **Philipp Jaspers** 58 Years	**Elisabeth Nordig**	Internship Software Engineering 10/02/2008 LA internship	**INTERVIEW1 - Invite applicant to 1. interview**
	●	1079	Mr. **Nelson Jimenz** 41 Years	**Elisabeth Nordig**	Sales Manager 10/17/2008 LA sales manager	**CIRCULATION - Application at the operating department**
	●	1011	Mr **PD Lasse Johansson** 37 Years	**Maurice Lacroix**	Vertriebsingenieur 05/25/2006 2006-VERTRIEB	**REJECTION_P - Rejection planned**
	●	1017	Mr. **Morss Justin** 31 Years	**Elisabeth Nordig**	AL Allgemeine Dienste 03/30/2006 2006-SI-WEST	**CIRCULATION - Application at the operating department**

Figure 1.2: Overview table in an existing HR application

Usually, electronic forms are provided to users in order to execute business functions. In particular, these forms may be invoked by users for reading and writing attribute values of business objects.

Relational DBMS constitute an important backbone of any tailor-made or domain-specific application. To enable rapid development, they handle data storage and management within a separate layer (cf. Fig. 1.3). Before DBMS emerged, for each application, data storage and access had been implemented from scratch and in a proprietary manner. As a consequence, each change of an application-specific data structure had required an adoption of the application code itself [Wes07]. In this context, DBMS contribute to reduce the efforts for implementing data management functions and for maintaining large application databases. For this purpose, usually, the application data structure is modeled in the DBMS by specifying the object types required as well as their attributes (cf. Fig. 1.4a+b). At run-time, the DBMS takes care of storing and managing data (i.e., object instances) in an efficient, consistent, and persistent manner (cf. Fig. 1.4c). DBMS further provide features like security, transactions, and indexing [GR92]. Using a DBMS, the implementation of application systems is usually based on architectures consisting of logically independent components. For example, consider 3-tier architectures as illustrated in Fig. 1.3. Here, the first component provides the DBMS interface, the second one enables user interactions (i.e., GUI components), and the third one covers business logic (i.e., the implementation of business functions). By explicitly representing data structures in DBMS, generic implementations of many GUI components (e.g., user forms and overview tables) become possible (cf. Fig. 1.4d). A dynamically generated form contains (editable) input fields for those object attributes, the respective user may write or read (based

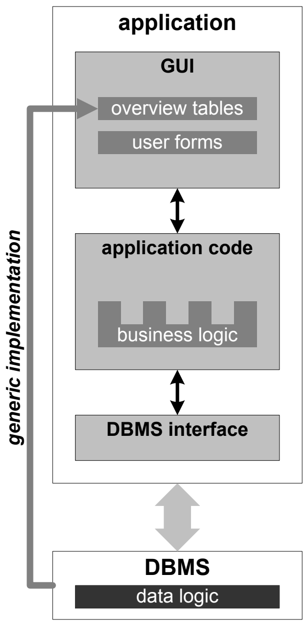


Figure 1.3: Architecture using DBMS

on his or her user role). Thereby, the type of an input field (e.g., text field, checkbox, combobox, or radio button) may depend on the data type of the respective attribute. In turn, if a user only owns read permissions, a corresponding data field displaying the value of the attribute is used. Altogether, *generic implementations of GUI components* (cf. Fig. 1.3) significantly reduce implementation efforts. Instead of programming thousands of different user forms, it becomes possible to automatically generate them at run-time based on defined data models and user authorizations (i.e., read and write permissions for individual attribute values).

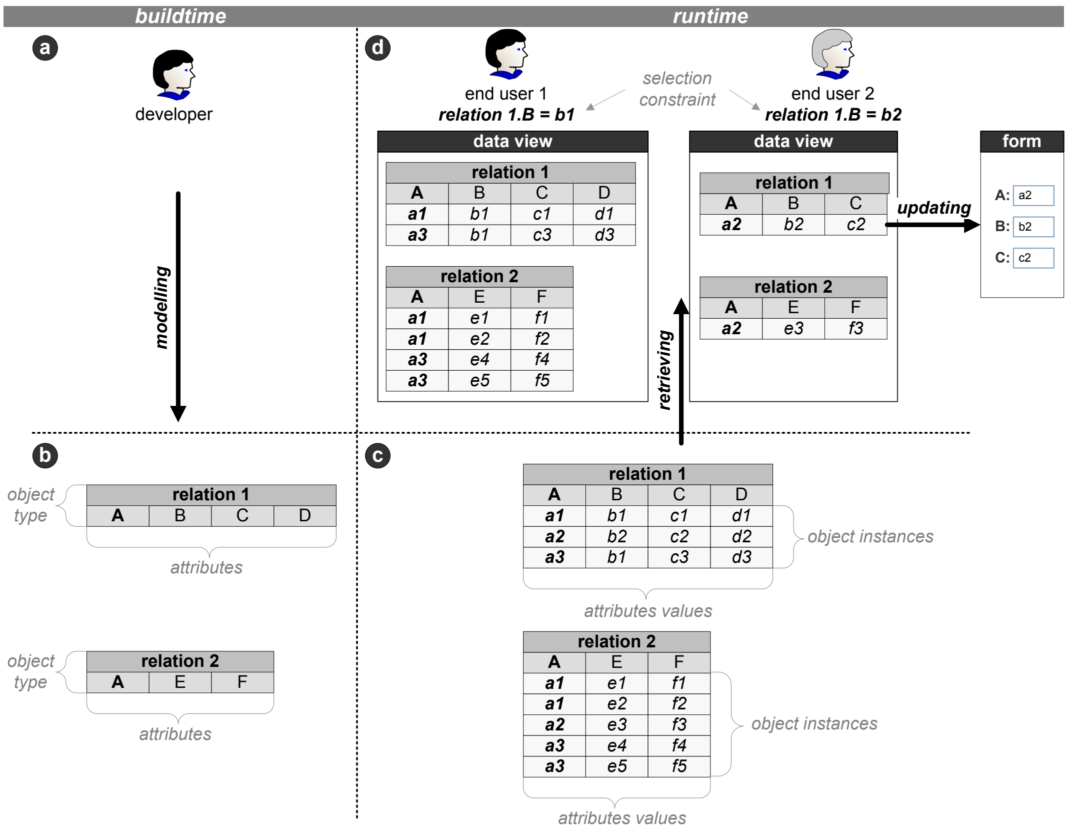


Figure 1.4: Relational Database Management System

During the last decade, new business needs have emerged [MRB08]. In addition to the provision of business data and business functions, business process support has become a major issue in information systems engineering (cf. Fig. 1.5). In order to effectively achieve business objectives [LR12], mandatory activities must be performed on selected business objects in a particular order. Furthermore, the execution of different business functions must be coordinated among different users. For the latter purpose, corresponding activities are assigned to the worklists of authorized users (cf. Fig. 7). Otherwise, users do not have the information about which activities must be executed next and whether the preceding activities have been finished.

PrMS provide generic functions for modeling, executing and monitoring processes; i.e., process logic is explicitly modeled instead of being hard-coded (cf. Fig. 1.6a). In contemporary PrMS, usually, a business process is defined in terms of activities and a number of control-flow constraints restricting their execution order [vH04]. As illustrated in Fig. 1.6b, an activity represents a task linked to a particular business function of an application system. Usually, application

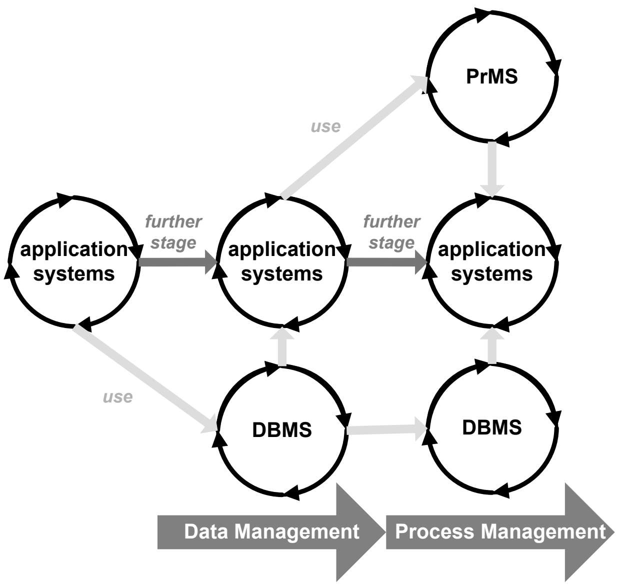


Figure 1.5: History of application system development

data is managed by the invoked applications themselves. In turn, only data relevant for process control (e.g., for evaluating transition conditions) or for providing activity input parameters are managed by the PrMS itself (cf. Fig. 1.6b). Furthermore, most PrMS are only able to deal with *atomic data elements* (i.e., attributes); i.e., it is not possible to group data elements or define semantic relations between them. Finally, roles and users are captured in an *organization model* that is usually maintained by the PrMS [RMR07].

To be able to assign human activities to the right actors, PrMS use *actor expressions* referring to entities of the organizational model (e.g., *user roles*) [RMR08, RMR09]. Actor expressions must be defined for each human activity. At run-time, for each *business case*, a separate *process instance* is created and then executed according to the defined process logic (cf. Fig. 1.6c). A particular activity will be only enabled if all preceding activities are completed or cannot be executed anymore (except loop backs). Similar to DBMS, which allow for the automatic generation of data-oriented views (i.e., overview tables) and form-based activities at run-time, PrMS automatically generate a *worklist* for each registered user based on a *generic implementation*; i.e., it is not necessary to implement specific worklists for each process defined. When a human activity becomes enabled, corresponding work items are then automatically added to worklists of authorized users (cf. Fig. 1.6d). An example of such a worklist is shown in Fig. 1.7. Finally, when a user selects a work item, the PrMS launches the associated business function as provided by a particular application system (cf. Fig. 1.6d).

Altogether, DBMS allow separating the logic of business functions from data management, whereas PrMS foster the separation of process and function logic (i.e., application code). Moreover, required data structures and process definitions are expressed in terms of models rather than manually coded. This fastens implementation, increases maintainability, and reduces cost of change [WRRM08, RW12].

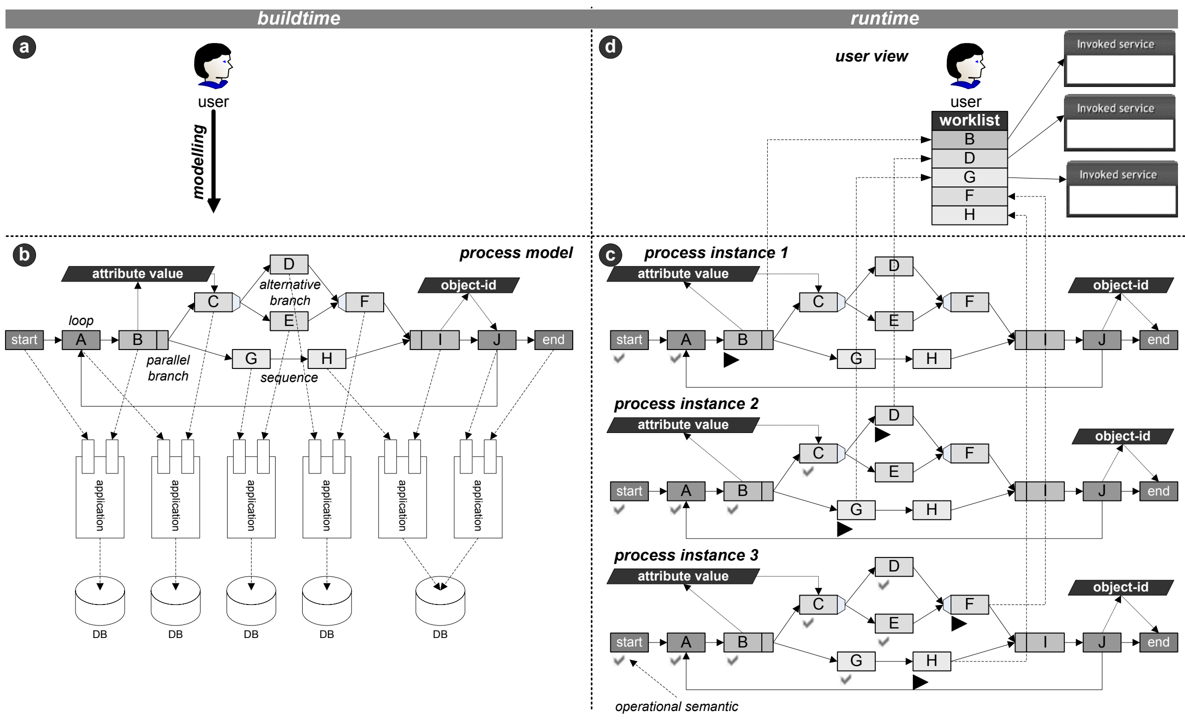
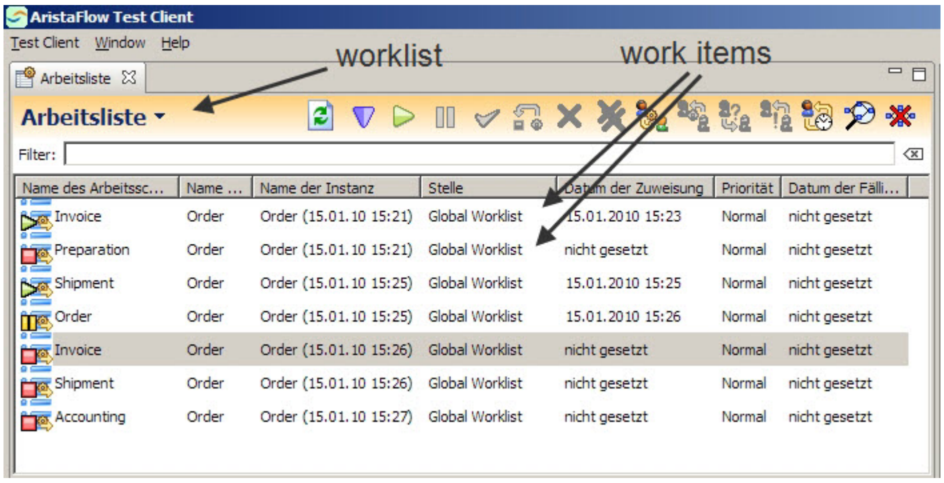


Figure 1.6: Imperative Process Management System (according to [KR09a])



Name des Arbeitssc...	Name ...	Name der Instanz	Stelle	Datum der Zuweisung	Priorität	Datum der Fälli...
Invoice	Order	Order (15.01.10 15:21)	Global Worklist	15.01.2010 15:23	Normal	nicht gesetzt
Preparation	Order	Order (15.01.10 15:21)	Global Worklist	nicht gesetzt	Normal	nicht gesetzt
Shipment	Order	Order (15.01.10 15:25)	Global Worklist	15.01.2010 15:25	Normal	nicht gesetzt
Order	Order	Order (15.01.10 15:25)	Global Worklist	15.01.2010 15:26	Normal	nicht gesetzt
Invoice	Order	Order (15.01.10 15:26)	Global Worklist	nicht gesetzt	Normal	nicht gesetzt
Shipment	Order	Order (15.01.10 15:26)	Global Worklist	nicht gesetzt	Normal	nicht gesetzt
Accounting	Order	Order (15.01.10 15:27)	Global Worklist	nicht gesetzt	Normal	nicht gesetzt

Figure 1.7: Worklist of a PrMS [LKRD10]

1.2 Problem Statement

DBMS are indispensable for implementing any applications system. In turn, PrMS have not been broadly used so far as expected by vendor software. One reason for this situation is that traditional PrMS have been primarily designed for the support of highly structured, repetitive business processes [LR00]. However, various processes are still not adequately supported

by these PrMS [RW12]. For example, [Sil09] characterizes the latter as *unstructured* or *semi-structured* processes, which are *knowledge-intensive* and *driven by user decisions* [MKR12]. They require high flexibility during process execution [DRRM11]. In turn, other authors argue that the business functions to be integrated with these processes cannot be *straight-jacketed into activities* [vWG05, SOSS05]; i.e., the activity-based execution paradigm interferes with business goals and organizational needs.
Furthermore, PrMS allow for a strict *separation of concerns* (cf. Fig. 1.8); i.e., data, functions, and processes are managed by different kinds of systems. However, this makes it impossible to provide integrated access to them. Without such integrated view, relevant context information is missing during process execution [vWG05]; i.e., application systems must be directly invoked outside the control of the PrMS. If data is changed directly in the underlying application system, however, inconsistencies between process and data states might occur. In addition, data-oriented views do not provide sufficient information on business processes and vice versa.

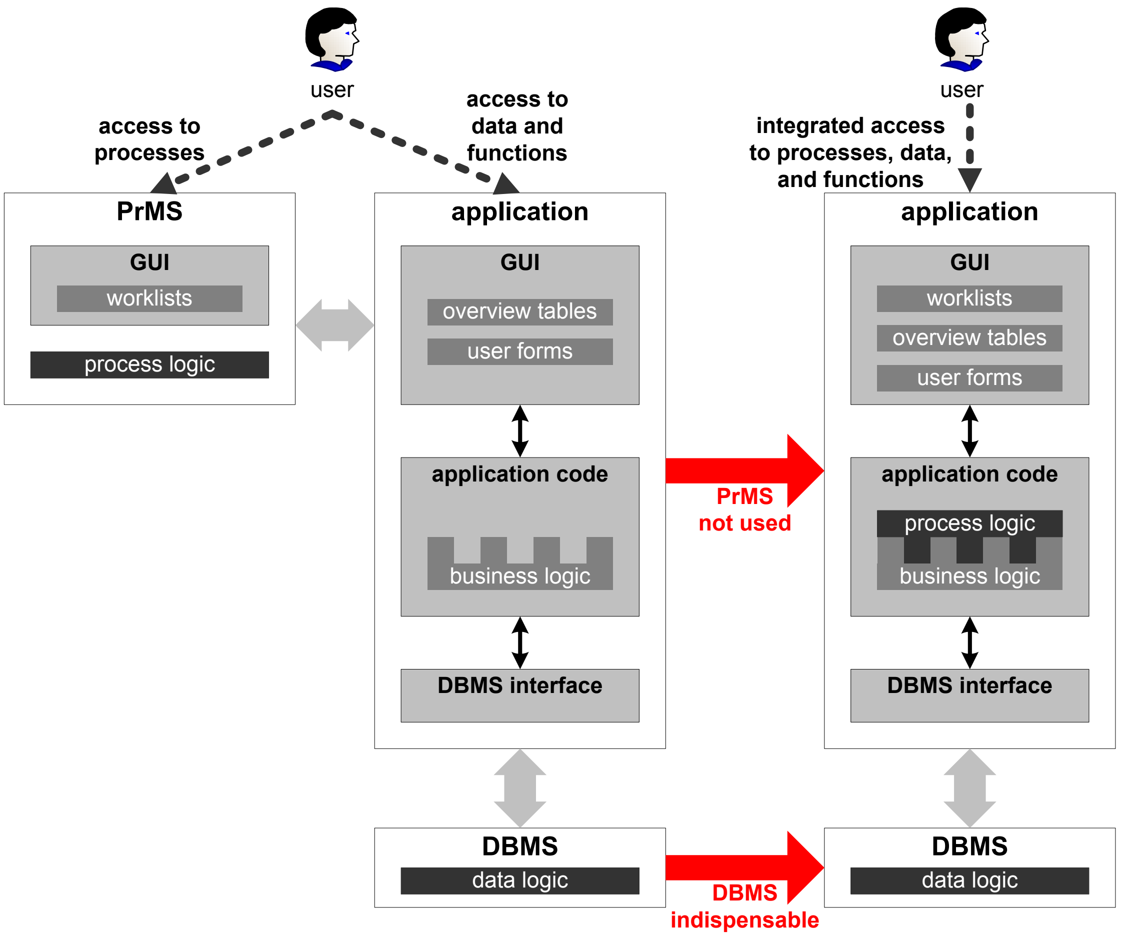


Figure 1.8: Application system architectures for providing process support

For these reasons, traditional PrMS cannot be used to support the processes in database-driven application systems. Note that in existing applications, providing such integrated support, process logic is usually hard-coded (cf. Fig. 1.8). In general, both tailor-made and domain-specific applications are complex to design and costly to maintain; e.g., even simple process changes might require costly code adaptations and high efforts for testing [RW12]. In this context, all processes require comprehensive programming efforts, and hence it is not possible to automatically generate worklists or other user components at run-time. Instead, user components like worklists need to be manually coded for each implemented process. The same applies to data-oriented views (e.g., overview tables) and user forms. In particular, which input and data fields shall be displayed when a particular user form, does not only depend on the user executing this activity, but also on the processing state of the involved process instance. Note that this requires a multiplicity of user forms whose implementation is a cumbersome and costly task.

Opposed to tailor-made applications, domain-specific applications are characterized by hard-coded processes that may be customized by setting specific system parameters [BHM01]; i.e., based on certain settings, one may configure a particular *process variant* [HBR10a, HBR10b, RW12]. From an organization's point of view, however, these process variants are usually not transparent. In addition, there exist multiple dependencies between the system settings required in this context. As a result, process variants usually can only be tailored by very experienced consultants. In turn, from the system provider's point of view, process logic as well as the settings are often (redundantly) scattered over the entire application code, which therefore becomes extremely complex. This results in long development cycles and high maintenance costs (e.g., when introducing new features).

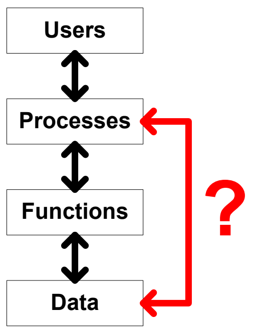


Figure 1.9: Dependencies

So far, many companies have taken benefit from their investments in ERP packages. However, since configurability depends on the range of preconfigured alternatives, these packages are very huge and extremely complex [KRG00]. In particular, configuration of domain-specific applications usually cannot be accomplished as quick and as efficient as the modeling of processes in PrMS. According to [HL99], at least 90% of ERP implementations end up late or over budget, and almost half of them fail to achieve desired results. Consequently, tailor-made as well as domain-specific applications require long development cycles as well as high maintenance efforts.

Finally, when dividing the system architecture of existing applications into data, functions, processes, and users (cf. Fig. 1.8), the role of data and processes, and especially their interdependencies, have not been well understood (cf. Fig. 1.9). However, the reasons why certain processes are currently not adequately supported by existing PrMS are not clear. Hence, fundamental research efforts in this area are required.

1.3 Contribution

On one hand, generic services as offered by DBMS and PrMS enable a faster implementation and better maintenance of information systems; i.e., a *model-driven development approach* provides higher flexibility compared to low-level programming and configuration. On the other hand, integrated access to data, processes and functions, as offered by many existing application systems, covers business needs best.
This thesis will show that the identified limitations of existing PrMS can be traced back to the unsatisfactory integration of the different business perspectives. So far, the required integration of data, functions and processes as well as their mutual relationships have not been well understood. To improve this situation, this thesis first conducts extensive case studies; i.e., it evaluates contemporary application systems (e.g., for human resource management [KB11] and management of conferences). The first part of this thesis presents the comprehensive results of these studies. More precisely, we introduce a list of properties for business processes, which we denote as "object-aware processes" and we elicit major requirements for their effective support.

Our overall vision is to develop a comprehensive framework for the support of object-aware business processes. This framework has been developed in the PHILharmonicFlows project[1] that was established in the context of this thesis. In particular, we pursue the following fundamental goals:

1. **Providing adequate business process support.** Process support for unstructured, data-driven processes, as it can be found in many contemporary application systems, shall be realized in a much more effective and generic manner.
2. **Providing integrated access to business data, business functions and business processes.** Data- as well as process-oriented views shall be provided in an integrated way.
3. **Providing generic implementations of end-user components.** Based on data and process models, a generic implementation shall be provided that allows for automatically generating of fine-grained end-user components at run-time (e.g., worklists, user forms, and overview tables) taking the given context (e.g., user, process state) into account as well.

Furthermore, the core of the PHILharmonicFlows framework shall cover the following stages of the process lifecycle:

1. **Modeling.** PHILharmonicFlows shall provide modeling components for defining executable processes and corresponding data structures in an integrated way. Opposed to traditional process support paradigms, it shall provide a uniform methodology for modeling processes at different levels of granularity. In addition, sophisticated concepts for integrating users and enabling fine-grained access control shall be provided in an effective manner.
2. **Execution.** The proper execution as well as termination of processes at run-time shall be ensured based on well-defined correctness rules. In addition, a precise and formal operational semantics shall be provided, which not only enables generic process execution, but

[1] Process, Humans and Information Linkage for harmonic Business Flows

also realizes generic business functions. For this purpose, the framework shall automatically create end-user components (e.g., work-lists, form-based activities, and overview tables).

The thesis provides an extensive validation of the developed framework. In particular, it introduces a proof-of-concept prototype for demonstrating fundamental concepts of the build- and the run-time environment of PHILharmonicFlows. In addition, we apply this prototype to real-world cases to elaborate the benefits of our approach as well as lessons learned.

Overall, this thesis comprises the following original contributions:

- It provides a software architecture for application systems, that enables a tight integration of data, functions, processes, and users, while retaining the well-established principle of separating concerns.
- It provides a sophisticated and very advanced process support paradigm that allows
  - decoupling activities from (particular) process steps; i.e., process execution is based on data rather than on black-box activities,
  - combining *object behavior*, expressed in terms of states and state transitions, with *data-driven process execution*. Opposed to existing approaches, an explicit mapping between object states and object attribute values is provided, and
  - coordinating the execution of different processes taking the relations between the involved object instances into account. In particular, coordination is not only accomplished along direct object relations, but the processing of object instances can be also coordinated based on their relationships within the overall data structure. In particular, transitive as well as transverse relationships between object instances are taken into account in this context. Finally, the provided approach allows for the asynchronous execution of process instances taking the one-to-many relationships between the processed object instances into account.
- It provides sophisticated concepts for user integration and access control. These concepts ensure consistency between data and process states at any point in time.
- It provides generic implementations for end-user components. This enables us to automatically generate overview tables, worklists, and form-based activities at run-time. In particular, users may select the object instances desired in a given context; i.e., a great variability of user forms is considered. This way, each user may choose his or her preferred work practice.

1.4 Outline

The remainder of this thesis is divided into four parts - motivation, object-aware processes and their properties, PHILharmonicFlows framework and its concepts, as well as evaluation and discussion:

Part I comprises an introduction (cf. Chapt. 1) and presents the research method we applied (cf. Chapt. 2).

Part II addresses the notion of object-aware processes. More specifically, Chapt. 3 first introduces their properties which are then underpinned by a detailed literature study. Based on the identified properties, Chapt. 4 discusses related work. Following this, fundamental requirements for the support of object-aware processes are elicited in Chapt. 5.

Part III of this thesis presents the PHILharmonicFlows[2] framework; i.e., a solution framework for object-aware process management. After giving an overview about the framework, we introduce concepts related to the underlying data model in Chapt. 6. In turn, process support is realized at different levels of granularity and therefore implemented in terms of micro and macro processes in our framework. Micro process modeling and execution is introduced in Chapts. 7 and 8. In turn, macro processes are used to coordinate the execution of micro processes. These concepts are introduced in Chapts. 10 and 12. Chapt. 9 discusses different kinds of activities as supported by the PHILharmonicFlows framework. Finally, further issues (e.g., exception handling, advanced user integration, and advanced data processing) are discussed in Chapt. 13.

Part IV provides an evaluation of the PHILharmonicFlows framework (cf. Chapt. 14) and summarizes the main contributions of the thesis (cf. Chapt. 15). We illustrate the proof-of-concept implementation of the modeling and the execution environment. Following this, the thesis gives insights into some of the lessons learned when applying this prototype to real-world processes. Finally, Chapt. 15 summarizes the contribution of this thesis, illustrates the benefit of the provided solution, and provides an outlook on future research directions.

[2]Process, Humans, and Information Linkage for harmonic Business Flows

2 Research Methodology

This chapter presents the research questions addressed by this thesis and introduces the research methodology applied. [1]

Generally, in the context of IT research (cf. Fig. 2.1), two scientific approaches may be applied: natural science and design science [MS95, HMPR04]. *Natural science research* is knowledge-producing and comprises *discovery* and *justification* as major steps [MS95]. In turn, *design science* is a knowledge-using activity [MS95], which aims at developing IT systems with *building* and *evaluation* as major tasks [HMPR04]. Research should be usually based on natural science. Concerning IT, however, design science has been considered as being more successful and relevant. Nevertheless, technology and behavior cannot be separated from each other [HMPR04]. According to [MS95, HMPR04], therefore, IT research might result in significant contributions when engaging in both scientific approaches [HMPR04]. This thesis follows this suggestion and utilizes the synergistic effects that can be achieved when combining natural and design science research.

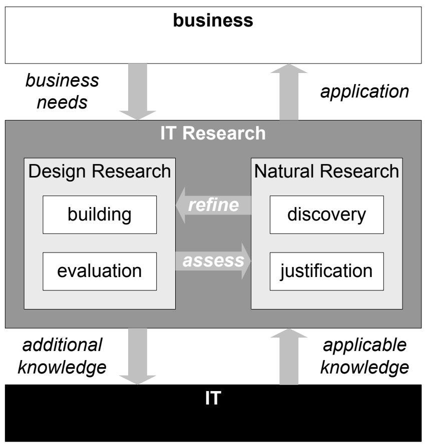


Figure 2.1: IT research [HMPR04]

[1]The chapter is based on the following referred paper:
[Kï1] V. Künzle. Towards a Framework for Object-Aware Process Management. In *1st Int'l Symposium on Data-driven Process Discovery and Analysis (SIMPDA'11), PhD Seminar*, 2011
[KR11c] V. Künzle and M. Reichert. PHILharmonicFlows: Research an Design Methodology. *Technical Report*, 2011

2.1 Research Questions

Starting with the basic observation that there exist processes not adequately supported by existing PrMS, this thesis considers the following research questions:

- **Research Question 1:** What are the common properties of these processes?
- **Research Question 2:** Which requirements must be fulfilled by a PrMS in order to adequately support processes with these properties?
- **Research Question 3:** How can these requirements be supported by an integrated conceptual framework?

To answer these fundamental questions, our main research activities are as follows (cf. Fig. 2.2): We start with natural research to identify the characteristic properties of those business processes currently not adequately supported (cf. Research Question 1). To deal with Research Question 2, we evaluate existing approaches (i.e., already available knowledge) to elicit the requirements for an information system supporting the identified properties. Finally, we address Research Question 3 and develop a comprehensive framework as well as proof-of-concept prototype for an approach supporting the requirements elicited. In particular, the development of this framework is based on design research.

2.2 Doing Natural Research: Property Investigation and Justification

To discover the properties of the business processes not adequately supported by current PrMS, first of all, we perform a comprehensive *property investigation* for respective processes. Then, we justify our findings with an extensive *literature study*.

2.2.1 Process Analysis

Data Source: Due to the limitations of existing PrMS, many business applications (e.g., ERP systems) do not rely on PrMS, but realize a hard-coded process logic instead. In order to ensure that the processes, we analyzed in our property investigation, are not self-made examples, but constitute real-world processes, we investigated the processes implemented by existing application software. Amongst others, we analyzed the processes supported by the human resource management system *Persis* and the reviewing system *Easychair* [KR09c, KR09b]. However, our evaluation was not restricted to the inspection of user interfaces solely. In addition, the author of this thesis gathered extensive practical experiences as software developer of the Persis system; i.e., she has deep insights into the application code of this system as well as the various processes implemented. Finally, we confirmed the results by interviewing system users as well as system consultants.

Selection Criteria: We evaluated the processes based on the main building blocks of existing PrMS. The latter comprise business processes, business data, business functions, and users (cf. Fig. 2.1). In particular, we focussed on the fundamental interdependencies that exist between these building blocks. Finally, to set a focus, we restricted ourselves to properties related to process modeling, execution and monitoring.

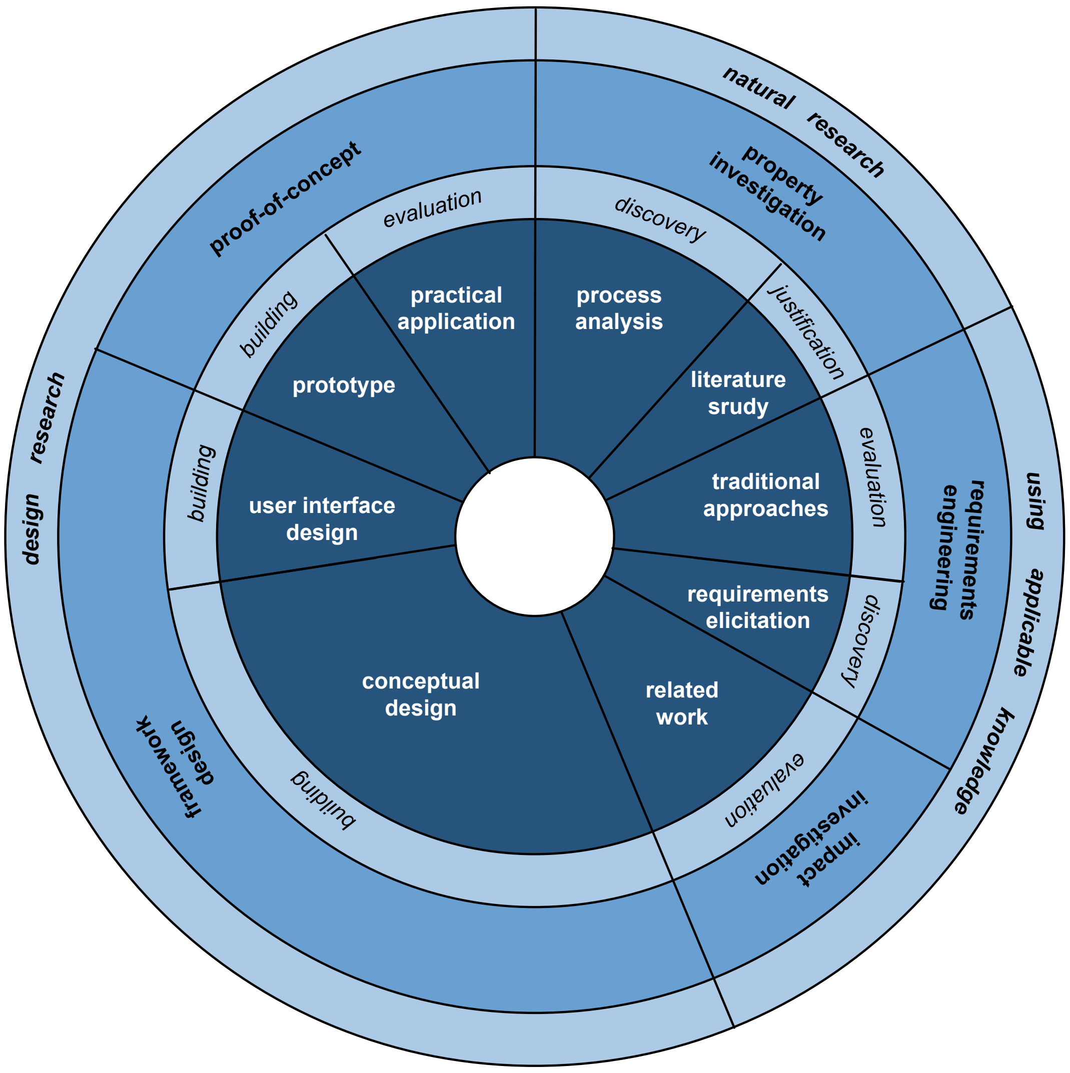


Figure 2.2: Research methodology (according to [KÏ1, KR11c])

2.2.2 Literature Study

Ensuring importance: We complemented our process analysis by an extensive literature study. This way, we were able to show that other work also considers the identified properties as relevant.

Ensuring completeness: In order to not exclude important properties, we compared our analysis results with existing literature. Though we were able to identify additional properties (e.g., relating to process change and process evolution), they did not directly relate to process modeling, execution, and monitoring. To set a focus, we omit them in the context of this thesis.

Ensuring generalisation: In our literature study, we identified several approaches that target at a tight integration of business processes and business data. Interestingly, existing work partially refers to similar application scenarios as considered by this thesis, while addressing only selected properties. Based on these insights, we contrasted the different application scenarios with the total set of identified properties. This way, we were able to demonstrate two things: first, the properties are related to each other. Second, broad support for them is required by a variety of processes from different application domains.

2.3 Using Applicable Knowledge: Requirements Elicitation

To elicit the basic requirements for supporting object-aware processes, we compared the identified properties with the main process characteristics addressed by traditional PrMS [KWR10b, KWR10a]. More precisely, we evaluated which properties cannot be directly supported when applying traditional imperative, declarative and data-driven approaches [KWR10b, KWR10a]. Though the identified requirements are not complete in the sense that they cover all aspects one can imagine, their fulfilment is indispensable for realizing the fundamental properties for modeling and executing object-aware processes.

2.4 Doing Design Research: Framework Design and Proof-of-Concept

Hevner et al [HMPR04] consider solution design as search process being inherently iterative. This has been confirmed by other authors [Sim96, Boe86]. In turn, Simon [Sim96] describes the nature of the design process as a Generate/Test Cycle. The spiral model [Boe86] defines an approach in which one and the same step is repeated several times, each time improving the results of the previous outcome. For this purpose, we performed iterative walkthroughs. In particular, we revised our solution and improved it step by step. This has led to different development versions. Additionally, we investigated in user interface design [Sch10, Wag10]. This way, shortcomings concerning the usability of the framework design were identified at an early project stage and considered in subsequent iterative revisions.

In order to evaluate the PHILharmonicFlows framework, we developed a powerful proof-of-concept prototype realizing its build-time and run-time environment. We applied this prototype to a real-world case. Additionally, we evaluated the developed concepts using other process scenarios; e.g., order handling, house building, vacation requests, and patient treatment. In particular, these scenarios are different from the ones we considered in the context of our process analyses. Finally, we elaborated the benefits of our approach when applying it to these processes.

Part II

Object-Aware Processes

3 Properties of Object-Aware Processes

To better understand why existing PrMS lack an adequate support for many of the processes which are hard-coded in existing application systems, we thoroughly investigate these processes and perform a systematic analysis of their properties.[1] To gain empirical evidence, we evaluate the properties of data- and process-oriented application software (with hard-coded process logic) in several case studies. For example, we study widely used information systems from the areas of human resource management and document reviewing (cf. Chap. 2). In this context, we analyze the nature of *data*, *activities*, *processes*, and *users*, as well as the complex interdependencies between these entities. As our major finding, we observe that many business processes require *object-awareness*; i.e., they focus on the processing of business data, which is represented in terms of business objects. Based on the insights gained during our case studies, we derive propositions in respect to the fundamental properties of *object-aware processes*. This chapter summarizes the findings we gathered during these case studies.

In the following, we discuss basic properties of object-aware business processes along a characteristic example for recruiting people (cf. Fig. 3.1).

[1]The chapter is based on the following referred paper:
[KR09c] V. Künzle and M. Reichert. Towards Object-aware Process Management Systems: Issues, Challenges, Benefits. In *Enterprise, Business-Process and Information Systems Modeling (BPMDS'09)*, volume 29 of *LNBIP*, pages 197–210. Springer Berlin Heidelberg, 2009
[KR09b] V. Künzle and M. Reichert. Integrating Users in Object-Aware Process Management Systems: Issues and Challenges. In *Business Process Management Workshops (Proc. BPD'09)*, volume 43 of *LNBIP*, pages 29–41. Springer Berlin Heidelberg, 2009
[KWR10a] V. Künzle, B. Weber, and M. Reichert. Object-aware Business Processes: Fundamental Requirements and their Support in Existing Approaches. *International Journal of Information System Modeling and Design (IJISM)*, 2(2):9–46, 2010

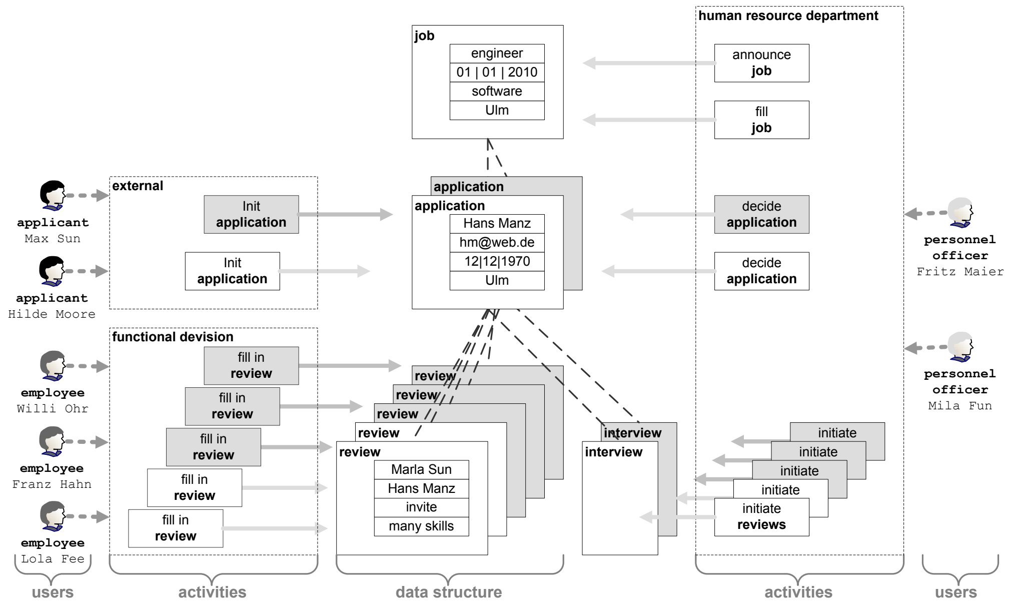


Figure 3.1: Example of a recruitment process (according to [KR11b, KR11d, KWR10a])

Example 3.1 (Example of a recruitment process from the human resource domain):
In the context of recruitment, `applicants` may apply for `job offers` via an online form. Before an `applicant` may send her `application` to the respective company, specific information (e.g., `name`, `e-mail address`, `birthday`, and `residence`) must be provided. Once the `application` has been submitted, the responsible `personnel officer` in the human resource department is notified. The overall process goal is to decide which `applicant` shall get the `job`. Since many `applicants` may apply for a particular `job offer`, usually, different `personnel officers` are involved in handling the `applications`. If an `application` is ineligible, the `applicant` is immediately rejected. Otherwise, `personnel officers` may request internal `reviews` for each `application`. Depending on the functional divisions concerned, the concrete number of `reviews` may differ from `application` to `application`. Corresponding `review` forms must be filled by `employees` from functional divisions who make a `proposal` on how to proceed; i.e., they indicate whether the `applicant` shall be invited for an `interview` or be rejected. If the `reviewer` proposes to invite the `applicant`, an additional `appraisal` is needed. In turn, if the `reviewer` proposes to reject the `applicant`, a `reason` or an `alternative job offer` must be provided. After the `employee` has filled the `review` form, she submits it back to the `personnel officer`. In the meanwhile, additional `applications` might have arrived; i.e., `reviews` relating to the same or different `applications` may be requested or submitted at different points in time. In this context, the `personnel officer` may flag the `reviews` he already evaluated. The processing of the `application` proceeds, while corresponding `reviews` are created; e.g., the `personnel officer` may check the `CV` and study the `cover letter` of the `application`. Based on the incoming `reviews`, he makes his `decision` on the `application` or initiates further steps (e.g., `interviews` or additional `reviews`). Finally, he does not have to wait for the arrival of all `reviews`; e.g., if a particular `employee` suggests hiring the `applicant`, he may immediately follow this recommendation.

3.1 Property Identification

This section discusses fundamental properties of object-aware business processes along the introduced example. Related to the different business perspectives (cf. Chapt. 1), we focus on *data integration*, *process granularity*, *process modeling* and *execution*, *activities*, and *user integration*.

3.1.1 Data Integration

Generally, we figured out that many limitations of existing PrMS can be traced back to the unsatisfactory integration of processes and data. In particular, the scenarios we analyzed in our case studies are all characterized by a tight integration of business data, which is represented in terms of business objects. Respective business data should be manageable and accessible based on complex objects rather than atomic data elements. Usually, application systems manage data in terms of different object types. Each object type comprises a set of attributes describing different properties of the object type. For example, consider the `review` object type as illustrated in Fig. 3.2. A `review` comprises attributes like `urgency`, `proposal`, `appraisal`, `remark`, and `comment`. At run-time, for each object type, a varying number of object instances may be created. Object instances of the same type usually differ in the values of their attributes (cf. Fig. 3.2). While a particular `review` object instance propose to invite the `applicant` (i.e., attribute `proposal` has value "invite"), another one might suggest rejecting the candidate (i.e., attribute `proposal` has value "reject").

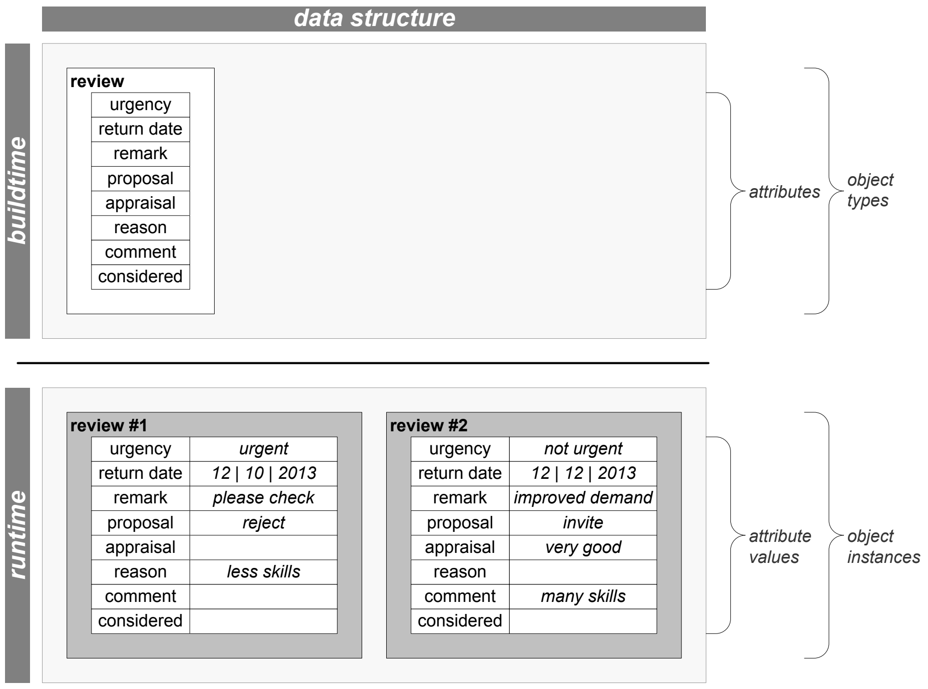


Figure 3.2: Example of an object type and related object instances

Property 1 (Object Types)
Data is managed in terms of object types of which each comprises a number of attributes.

In general, object types are inter-related. Since this thesis focuses on *relational databases* [Cod90], each relation constitutes a *one-to-many relationship*. For example, consider the relation between object types `application` and `review` as illustrated in Fig. 3.3; i.e., object type `review` refers to object type `application`. Hence, at run-time, an `application` object instance may be referenced by number of `review` object instances; i.e., for a particular `application` several `reviews` exist. In the following, we denote an object instance, which is directly or transitively referenced by another object instance o, as *higher-level object instance* of o; e.g., an `application` is a higher-level object instance of a set of `reviews`. By contrast, an object instance directly or transitively referencing another object instance is denoted as *lower-level object instance*; e.g., `reviews` are lower-level object instances of an `application` object instance.

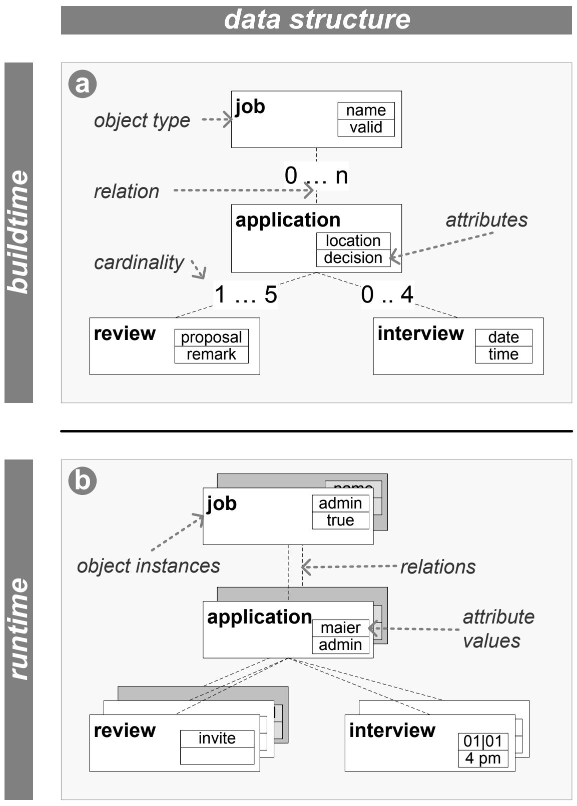


Figure 3.3: Data structure at build- and run-time (according to [KR11b, KR11d, KWR10a])

Property 2 (Object Relations)
Object types are inter-related.

Overall, object types, object attributes, and object inter-relations form a *data structure*. At run-time, object types may have a varying number of inter-related *object instances*. In this context, it is possible to restrict the concrete number of object instances referring the same higher-level object instance by lower and upper bounds (i.e., *cardinalities*). For example, consider Fig. 6.6a: for each `application`, at least one and at most five `reviews` are required. Consequently, object instances of the same object type may not only differ in their attribute values, but also in their inter-relations (cf. Fig. 6.6b); e.g., for a particular `application` two `reviews` might be requested, while for another one three `reviews` are needed. A challenge in this context is to cope with this varying and dynamic number of object instances to be handled at run-time. Thereby, the different relations between the object instances must be considered as well.

Property 3 (Cardinalities)
At run-time, for each object type a varying number of object instances may exist. This number may be restricted in different way through cardinality constraints.

3.1.2 Process Granularity

As a major finding we learned that many business processes focus on the processing of business data represented in terms of business objects. Each business object comprises a set of attributes. Furthermore, business objects may be related to each other. In this context, processes require *object-awareness*, which means that the overall process model shall be structured and divided according to the object types involved. For each object type, therefore, a particular process type exists. Thus, the modeling of processes and data constitute two sides of the same coin, and should therefore correspond to each other. In accordance to data modeling, the modeling and execution of processes is based on two levels of granularity: *object behavior* and *object interactions*.

The first granularity level concerns the behavior of object instances, which describes the processing of an individual object instance. Usually, for each object type, a separate process definition exists [MRH07]. The creation of a process instance is then directly coupled with the one of an object instance; i.e., for each object instance, exactly one process instance exists.

Fig. 3.4 illustrates the mapping between object and process types and between object and process instances respectively; e.g., object type `job` has its own process type. At run-time, exist multiple instances of a `job` object. Accordingly, for each of them a separate process instance is created.

Property 4 (Object Behavior)
For each object type, a specific process type exists.

The second granularity level comprises the interactions taking place between the instances of different object types. More precisely, whether a particular process instance may proceed also depends on the progress of other process instances. Generally, for each object type, multiple object instances may exist (cf. Fig. 3.4b). In particular, these may be created or deleted at arbitrary points in time; i.e., the corresponding data structure dynamically evolves depending on the type and number of created object instances and their relations. Furthermore, the creation of an object instance is directly coupled with the one of the corresponding process instance. These individual process instances are executed asynchronously to each other as well as asynchronously to higher- and lower-level instances. In particular, they may be instantiated simultaneously or at different points in time. Consequently, individual object instances may be in different processing states at a certain point in time. For example, several `reviews` might have been requested for a particular `application`. While a particular one might have just

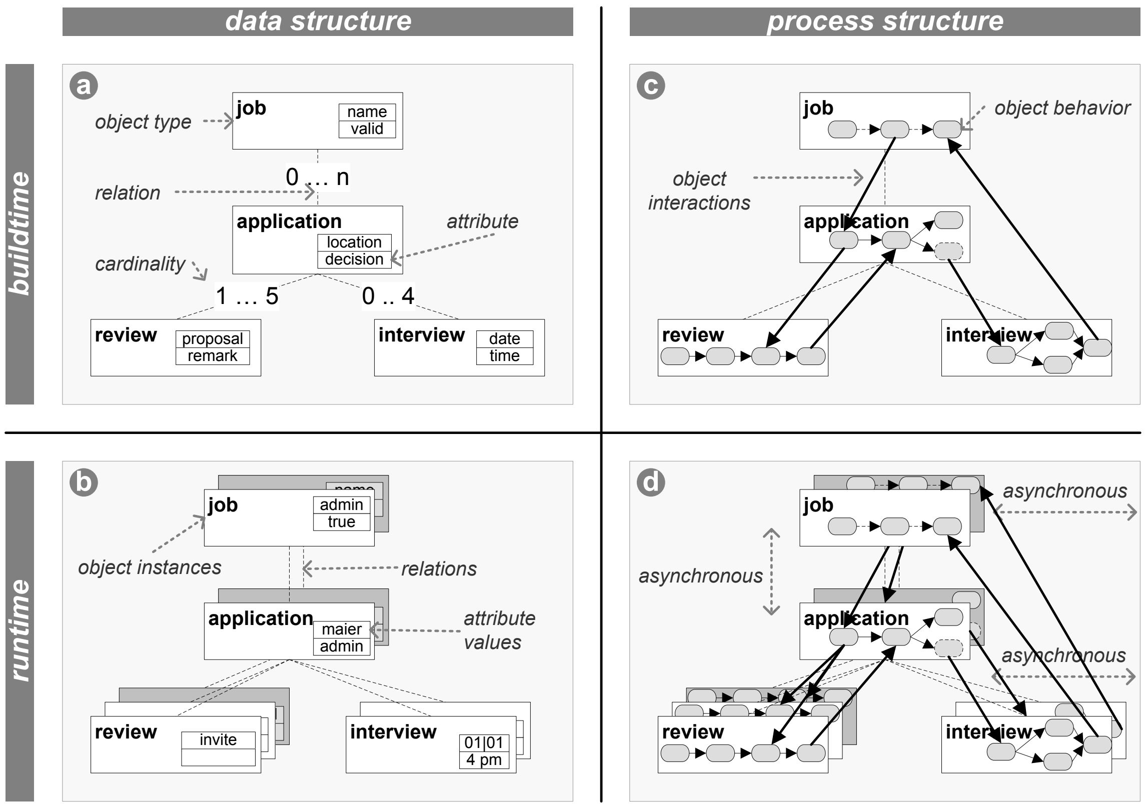


Figure 3.4: Process structure at build- and run-time

been initiated, others might have been already submitted back to the personnel officer who must inspect them. Taking the behavior of individual object instances into account, this results in a *complex process structure* in accordance with the given data structure (cf. Fig. 3.4d). For example, when executing a particular process instance, related subordinate process instances may be triggered. In turn, results collected during the execution of these subordinate process instances are relevant for executing the super-ordinate process instance as well.

When executing particular process instances, it should be possible to synchronize them whenever needed (cf. Fig. 3.5). For example, `applications` may only be filled in and sent as long as the corresponding `job` is published. Here, a number of related object instances depend on one higher-level instance. We denote this as *top-down dependency*. Consider that synchronization must be possible in an asynchronous way rather than be based on specific (i.e., pre-specified) points within the higher-level instance. For example, this means, that even if activity `evaluate` has been already activated, it is still possible to fill in an `application`.
Opposed to this, an `application` may only be rejected if all corresponding `reviews` propose the rejection. In this context, one higher-level instance depends on a number of lower level ones. We denote this as *bottom-up dependency*. For this purpose, it must be possible to aggregate several lower-level instances if needed. Moreover, such dependencies do not necessarily coincide to object relations. For example, consider the initiation of an `interview`. This is only possible if the `job` has been published and at least one `review` proposes to invite the applicant. Thus, *transitive* as well as *transverse dependencies* must be considered as well.

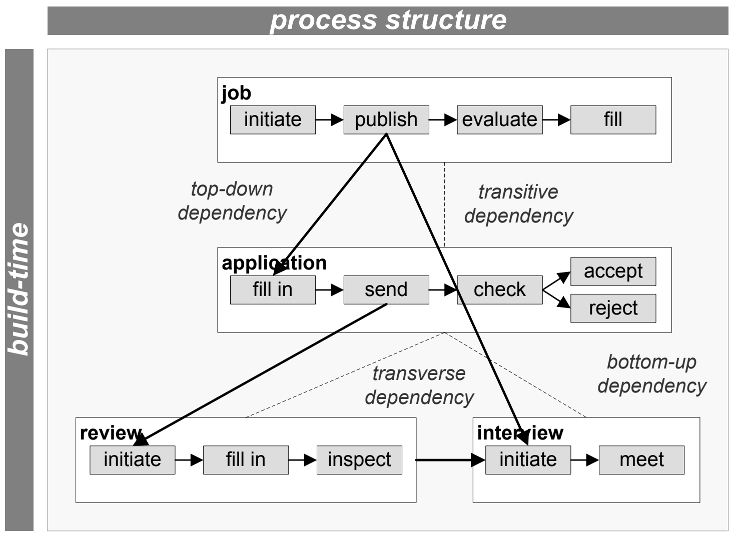


Figure 3.5: Process structure at build-time

Property 5 (Object Interactions)
Individual process instances are executed in a loosely coupled manner; i.e., concurrently to each other. Process synchronization needed in this context, must cope with a varying number of inter-related process instances. Additionally, synchronization should not be based on specific points during process execution, but follows a more asynchronous and flexible process coordination. In order to adequately consider the one-to-many relationships, which exist between the object instances corresponding to the respective process instances, it must be possible to aggregate sets of lower-level instances. Overall, top-down, bottom-up, transverse, and transitive dependencies must be considered (cf. Fig. 3.5).

3.1.3 Process Modeling and Execution

As another basic insight, we learned that the progress of a process often depends on available business objects and their attribute values. Hence, process execution should be accomplished in a *data-driven* manner. For object-aware processes, this means that they need to be defined in terms of data conditions rather than in terms of black-box activities.

Looking at an individual process type in more detail, usually, the creation of a new object instance is directly coupled with the one of the corresponding process instance. When executing activities, certain attribute values are mandatorily required to proceed with the flow of control. In this context, *object behavior* determines in which order object attributes must be written and what valid attribute settings are. Corresponding to this, the steps of a process are less defined in terms of black-box activities, but are rather defined based on *explicit data conditions*. Note that this is a fundamental difference compared to contemporary PrMS.

When defining *object behavior*, for each process step, it must be defined when it may be enabled. This should be accomplished by specifying pre-conditions on the attribute values of the object instance (cf. Fig. 3.6). Regarding our example, the data conditions used to describe the

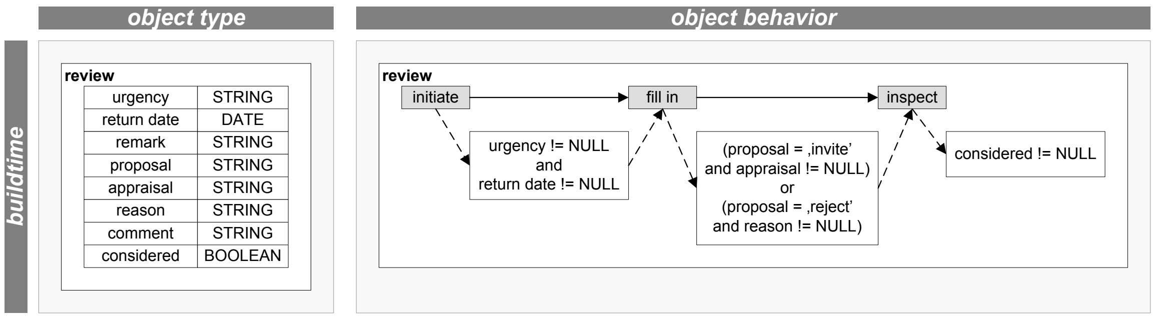


Figure 3.6: Object behavior

object behavior of the `review` object type are related to its attributes (cf. Fig. 3.6).
In turn, regarding *object interactions*, synchronization constraints between the instances representing object behavior may depend on the number of related object instances as well as on respective attribute values. Thus, for process synchronization, corresponding data conditions turn out to be more complex; i.e., varying numbers of related process instances as well as their asynchronous execution must be taken into account. For example, consider the creation of new `review` object instances, which is only possible as long as no decision for the corresponding `application` object instance has been made. Moreover, an `application` may only be rejected (i.e., assign value "rejection" to attribute `decision`) if all `review` object instances referencing the `application` have proposed rejection (i.e., value "reject" is assigned to attribute `proposal`).

Property 6 (Mandatory Information)
During the processing of individual object instances, certain attribute values are mandatorily required before proceeding with process execution. In turn, in the context of interactions between object instances, process instance execution may further depend on the attribute values of related process instances and object instances respectively.

To proceed with the execution of a particular process instance, *mandatorily required data* must be provided. For this purpose, activities for creating object instances as well as for editing object instance attributes must be performed. Respective activities are usually realized as *user forms* that provide input fields (e.g., text-fields, checkboxes, etc.) for writing selected attribute values and data fields (i.e., read-only input fields) for reading them. In particular, there exists one action (i.e., input or data field) for each attribute to be read or written in the context of a particular activity. Furthermore, each activity consists of at least one action. Consider the example from Fig. 3.7: before the `employee` may fill in the `review`, values for attributes `urgency` and `return date` must be provided by the personnel officer.

In the context of a particular activity, users may optionally read or write additional attribute values, which are not mandatorily required to proceed with process execution. For example, input fields corresponding to mandatorily required attributes may be marked using a red star. As illustrated in Fig. 3.8, in addition to the input fields for attributes `urgency` and `return date`, the personnel officer may optionally set a value for attribute `remark`.

Which object attributes are mandatory and which are not, may depend on the values of other object attributes as well. If an `employee` proposes to invite an applicant (i.e., value "invite" is

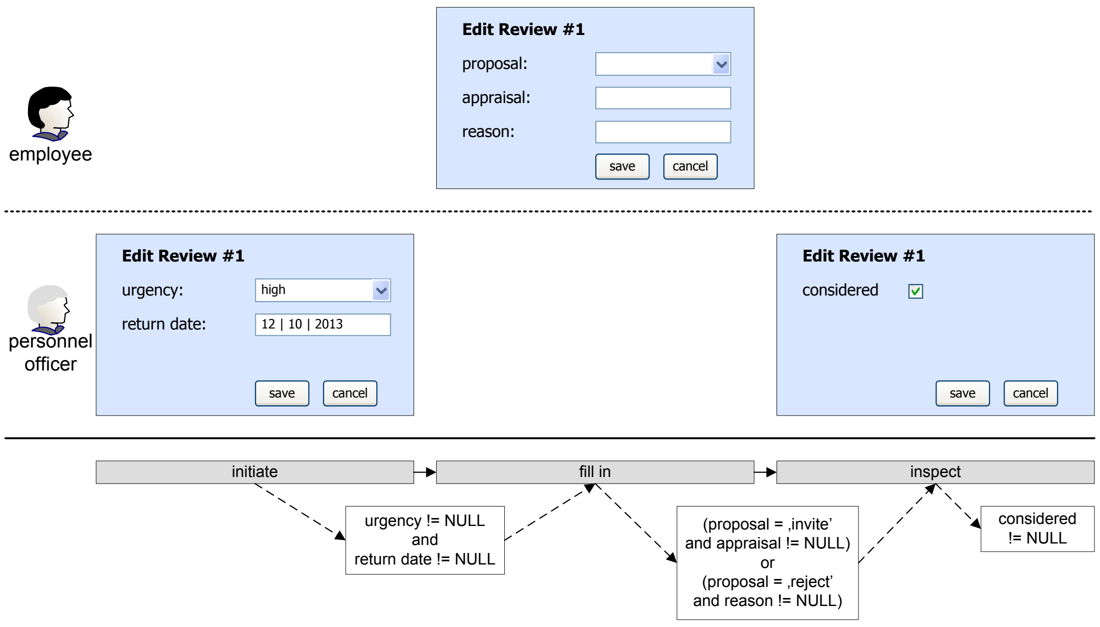


Figure 3.7: Mandatory activities comprising solely mandatory actions

assigned to attribute `proposal`), he must additionally provide a value for attribute `appraisal` (cf. Fig. 3.9a). Opposed to this, if he proposes to reject the applicant (i.e., value "reject" is assigned to attribute `proposal`) he must provide a `reason` instead (cf. Fig. 3.9b). Consequently, when filling a form, certain input fields might become mandatory on-the-fly. Such control flow, which is specific to a particular form, is common in existing application systems.

Property 7 (Control flow within User Forms)
Whether certain object attributes are mandatory when processing a particular activity might depend on certain other object attribute values. In particular, when filling a form, certain attributes might become mandatory on-the-fly.

In addition, users currently not executing a mandatory activity, may execute optional activities if desired; i.e., write certain attribute values even if these are not required at the moment. More precisely, certain activities might be optional, while others are mandatory to proceed with process execution. In turn, other form-based activities may be optionally executed to gather object information at any point in time regardless of the progress of the corresponding process instance. Opposed to this, other activities are mandatory and comprise actions changing the values of the object attributes that are referred by the data conditions of one or multiple process steps. Regarding the example from Fig. 3.10, an employee may fill the `review`, while a personnel officer may edit his `remark` and read attributes `urgency` and `return date`.

Property 8 (Optional activities)
Depending on the state of object instances, certain activities are mandatory for progressing with process execution. At the same time, users should be allowed to optionally execute activities (e.g., to write certain attributes even if they are not required at the moment).

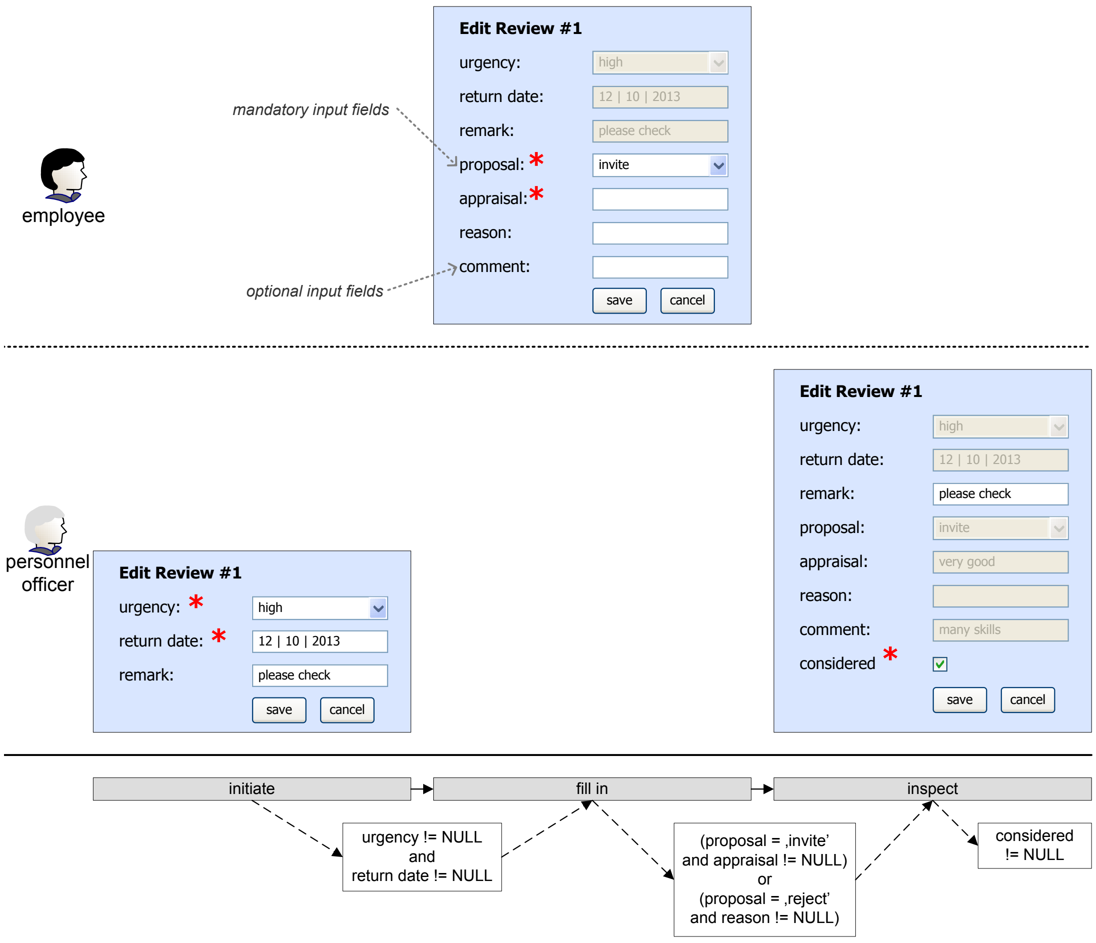
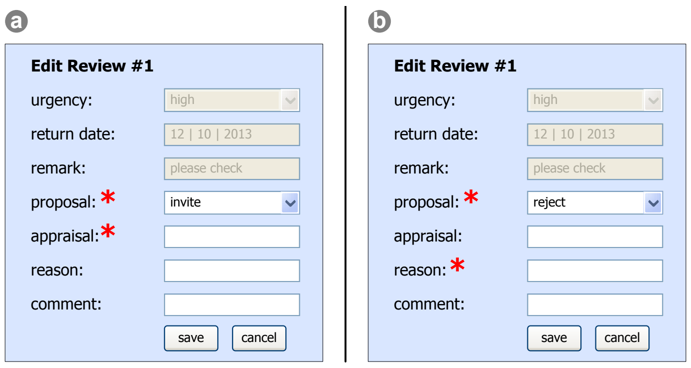


Figure 3.8: Mandatory activities comprising mandatory as well as optional actions

Note that when executing such *optional activities*, the attribute changes required to fulfill the data condition of a particular process step may be realized as well. For example, the employee may fill in the `review` (i.e., edit the input fields corresponding to attributes `proposal`, `appraisal`, and `reason`) before the personnel officer has completed the initiation of the `review` (i.e., before values for attributes `urgency` and `return date` are available). This way, it becomes possible to set respective attributes up-front; i.e., before the mandatory activity usually writing them becomes activated. Since this can be done asynchronously at arbitrary point in time, a high process flexibility can be realized. More precisely, if required data becomes available early, *mandatory activities* to be executed later may then be automatically skipped; i.e., the employee does not need to execute activity `fill in` when required attribute values are already available. Consequently, for any object-aware process, the progress of a corresponding process instance correlates with the attribute values of the associated object instance. This way, process state and object state are in sync at any point in time.

Property 9 (Flexible process execution)
Mandatory activities, which are no longer needed due to the early availability of the required attributes, may be automatically skipped.


a

Edit Review #1

urgency:	high
return date:	12 | 10 | 2013
remark:	please check
proposal: *	invite
appraisal:*	
reason:	
comment:	

b

Edit Review #1

urgency:	high
return date:	12 | 10 | 2013
remark:	please check
proposal: *	reject
appraisal:	
reason: *	
comment:	

Figure 3.9: Control flow within user forms

In certain situations, users may want to re-execute activities, even if all mandatory object attributes have been already set. For example, the employee may want to change the value of attribute `proposal` later on. In such cases, whether or not an activity is considered as completed depends on an *explicit user commitment*. More precisely, the employee explicitly decides when to submit the `review` back to the personnel officer. The latter, in turn, must not inspect the `review` before the employee has submitted it.
Moreover, users decide about the concrete number of object instances created for a particular object type. For example, the personnel officer decides about the concrete number of `reviews` required for each `application`. In general, as long as cardinality constraints are met, users should be free to decide whether or not additional object instances (and process instances respectively), shall be created. Consequently, the concrete data and process structure emerges dynamically at run-time and is determined by the users.

Property 10 (Re-executing activities)
Users should be allowed to re-execute a particular activity (i.e., to update its attributes), even if all mandatory object attributes have been already set.

Property 11 (User decisions)
The progress of a process instance does not only depend on available data. In certain situations, in addition, a commitment of the responsible user is required to proceed with the execution of the process instance. Moreover, users may decide about the concrete number of object instances created at run-time (as long as all cardinality constraints are met).

3.1.4 Activities

Providing access to business data and processing this data, constitute important features of any application system. Typically, particular activities are more related to the involved object instances rather than strictly be assigned to single process steps.

Basically, activities can be categorized into *form-based* and *black-box activities*. In object-aware processes, usually, the values of object attributes may be accessed during activity execution. More precisely, when executing activities, certain attribute values are mandatorily

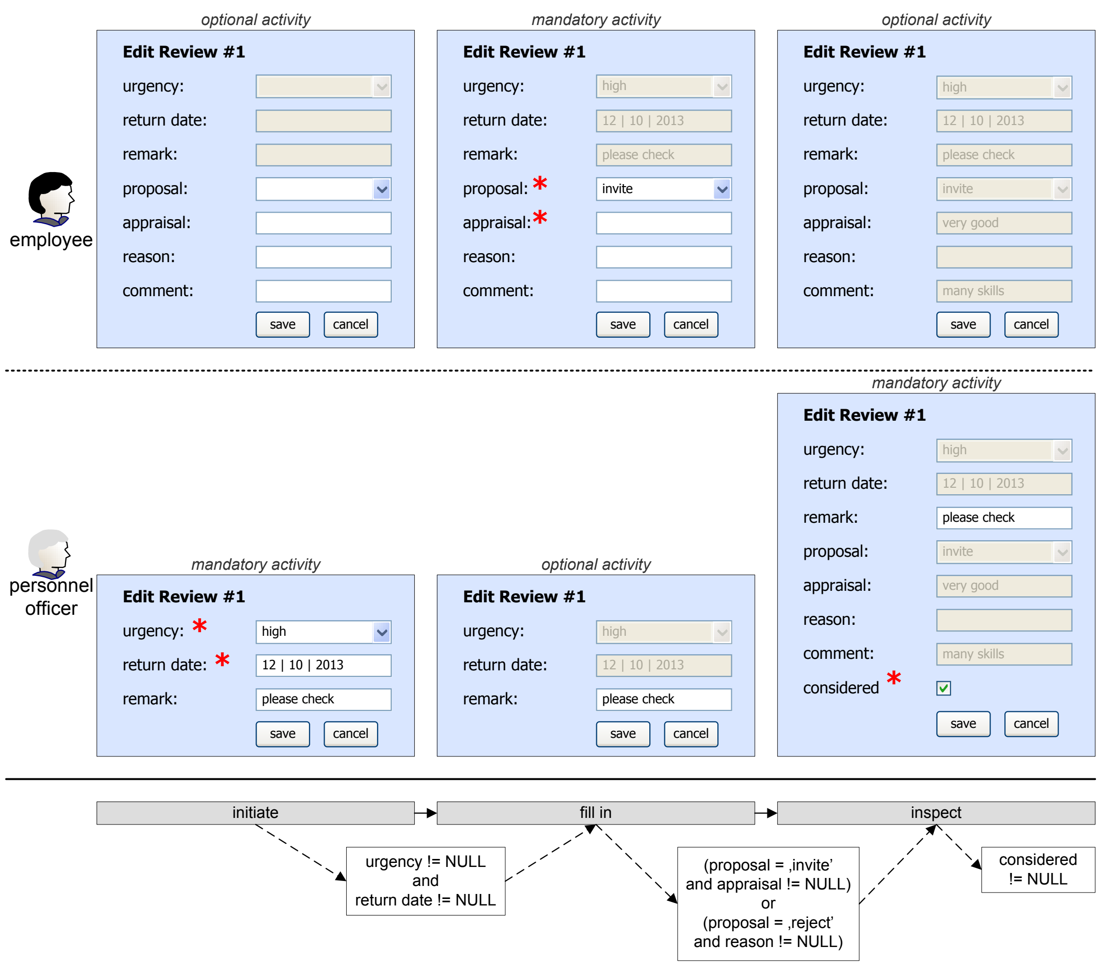


Figure 3.10: Mandatory and optional activities

required to proceed with the flow of control. Usually, activities requiring user input are realized as *user forms*. In addition, black-box activities enable complex computations or allow integrating advanced functionalities (e.g., sending e-mails or invoking web services). Regarding form-based activities, whether or not an input field is displayed for a particular user depends on his data authorizations.

Property 12 (Form-based activities)
A form-based activity comprises a set of atomic actions. Each of them corresponds to either an input field for writing the value of an object attribute or a data field for reading it.

Property 13 (Black-box activities)
Black-box activities enable complex computations or allow integrating advanced functionalities. Hence, for each black-box activity, a corresponding implementation is required.

Activities can be further categorized as *instance-specific* or *context-sensitive*. *Instance-specific activities* correspond to exactly one object instance (cf. Fig. 3.11a). Usually, when executing

such an activity, selected attributes of this object instance may be read, written, or updated. This is accomplished using a form (e.g., the form an applicant may use for entering his application data). However, black-box activities may be also instance-specific. Such activities only require input parameters referring to the attributes of one particular object instance. For example, consider an activity for sending an info-mail to an applicant.
In turn, a *context-sensitive activity* additionally changes attribute values of higher- or lower-level object instances (cf. Fig. 3.11b). Context-sensitive, form-based activities may comprise input fields corresponding to attributes of several object instances; e.g., when an employee is filling in a `review`, additional information about the corresponding `application` should be provided (i.e., attributes belonging to the `application` the `review` refers to). When integrating lower-level object instances, usually, a collection of object instances is considered. When a personnel officer edits an `application`, for example, all corresponding `reviews` should be visible. In turn, a context-sensitive, black-box activity requires attribute values of several object instances as input parameters. For example, consider an activity comparing the skills of an applicant with the requirements of the respective job.

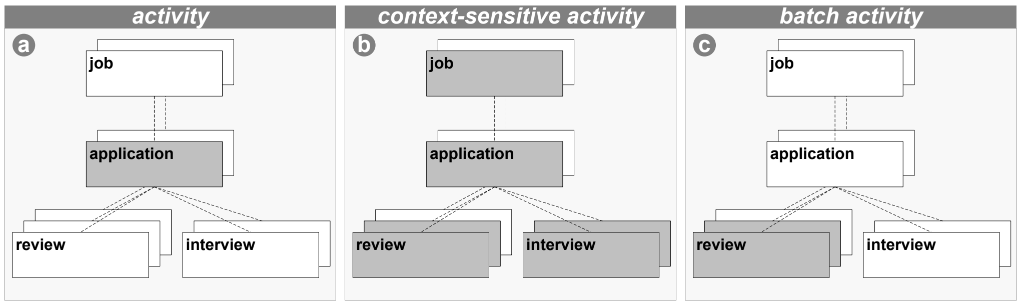


Figure 3.11: Activity types (according to [KR11b, KR11d, KWR10a])

Using context-sensitive activities, several process instances may be processed in the context of one and the same activity. Consequently, activities do not necessarily coincide to single process steps as in traditional process support paradigms (cf. Chapt. 4).

Property 14 (Varying activity granularity)
Depending on their preference, users should be allowed to freely choose the most suitable activity type for achieving a particular goal. Regarding instance-specific activities, all actions refer to attributes of one particular object instance and therefore belong to exactly one process instance (i.e., object behavior). In turn, context-sensitive activities comprise actions referring to different, but related object instances (of same or different type). Since each object instance is coupled with exactly one process instance (describing the behavior of the object instance), a context-sensitive activity belongs to several process instances.

Generally, many object instances may exist for a particular object type. In this context, a *batch execution* of activities allows users to change a collection of selected object instances in one go (cf. Fig. 3.11c). Using batch execution in connection with form-based activities, attribute values may then be set using one form and then be assigned to all object instances; e.g., a personnel officer might want to flag a collection of `reviews` as "evaluated" in one go, or, as soon as an applicant is hired for a `job`, for all other `applications` value "reject" shall be set for attribute `decision` by filling one form. In turn, regarding black-box activities one and the same activity may be invoked for a number of instances in one go. As example consider the activity used to send an info-mail to an applicant. Using batch execution, it becomes possible to send the info-mail to a number of applicants in one go.

Property 15 (Batch execution)
Activities shall be executable for a number of object instances (of the same type) in one go. Using form-based activities, attribute values shall be set based on one particular form. Respective values are then assigned to a number of selected object instances. In turn, regarding black-box activities one and the same activity may be executed for a number of object instances in one go.

Altogether, there exist several possibilities to reach a process goal. Depending on their preference, users should be free to select the most suitable activity type in order to fulfill a specific task; i.e., there may exist alternative ways of achieving a particular goal. More precisely, it is up to the user whether he selects an instance-specific activity, context-sensitive activity or batch activity in order to provide the required attribute values. In addition, users may terminate an activity, even if required data is missing and re-execute it afterwards. This shall be also possible if all mandatorily required attribute values have been already set.

3.1.5 User Integration

In existing PrMS, human activities are usually associated with actor expressions at build-time (e.g., user roles). We denote this as *type-specific authorization* (cf. Fig. 3.12). Respective expressions are then applied to all process instances of the respective type. Users who may work on an activity are then determined during run-time based on the actor expression. Regarding our example (cf. Fig. 3.12), each user owning role `personnel officer` may check `applications`. More precisely, each `personnel officer` may check any `application`.

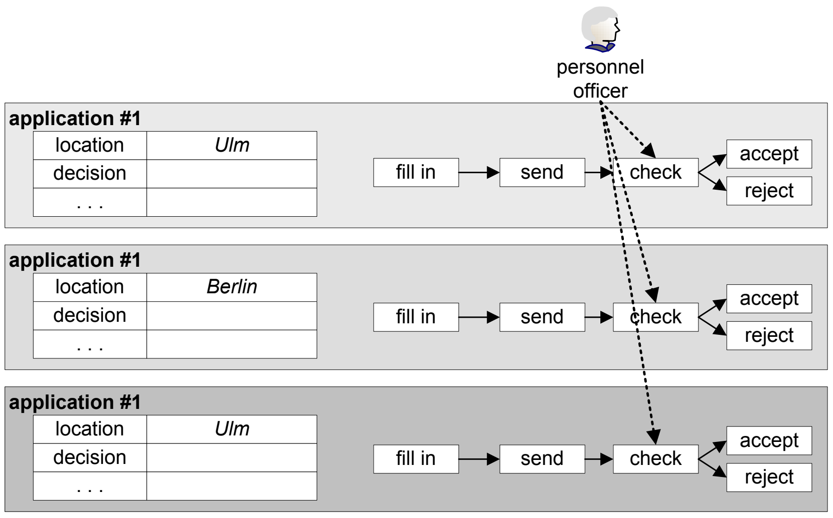


Figure 3.12: Type-specific authorization

Regarding object-aware processes, a type-specific authorization alone is not sufficient. In particular, the selection of potential actors not only depends on the activity itself, but also on the object instance processed by this activity [RzM98, RzM04]. We denote this as *instance-specific* *authorization*. As example consider Fig. 3.13, which depicts different process instances handling `applications`. While one `personnel officer` might be responsible for `applications` that apply for a `job` at a `location` whose name starts with a letter between 'A' and 'L', another one may only have access to `applications` referring to a `job` at a `location` whose name starts with a letter between 'M' und 'Z'. Existing PrMS lack support for such data-dependent, instance-specific authorizations.

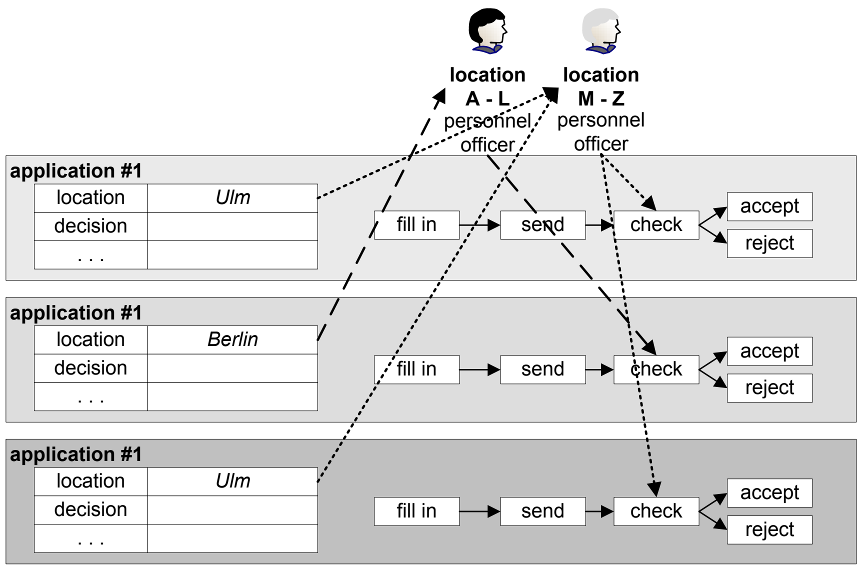


Figure 3.13: Instance-specific authorization

Property 16 (Instance-specific authorization)
The selection of potential actors for executing an activity does not only depend on the activity and its associated user role itself, but also on the object instance processed by this activity.

Regarding instance-specific authorizations, user permissions have to be restricted to a set of object instances or to a specific object instance [HW04, KKC02]. In general, the mapping between users and object instances (i.e., the regulation which user shall have which access rights for which object instances) is not arbitrary, but must obey certain constraints [BBU99]. For example, an `applicant` may only scan his own `application`, but must not access the ones of other `applicants`. Thus, the relationships between users and object instances must be taken into account when defining data and process authorizations.

Property 17 (Authorizations considering relationships to users)
The selection of potential actors for executing an activity may also depend on the relationship between the user and the object instance to be processed by the activity.

To proceed with process execution, certain attribute values must be set. For this purpose, authorized users must execute mandatory activities. In this context, form-based activities comprise input fields corresponding to the required attributes. Consequently, not only the actor expressions, but also permissions for accessing data must be taken into account when assigning activities to users. In particular, it is very crucial that process authorization (i.e., actor expressions) complies with data authorization.

Property 18 (Compliance between data and process authorization)
Each user who must execute a process-relevant activity, needs to own respective data permissions for setting required attribute values.

When working on mandatory (form-based) activities, users may optionally edit additional attribute values together with the ones mandatorily required. In particular, based on optional activities authorized users may access or change data at any point in time. However, undesired manipulations of object attributes, which are relevant for process execution, should be prevented. For example, an `applicant` must write the values of attributes `CV` and `cover letter` before he may submit his `application`. However, if the `applicant` has already sent his `application`, he must no longer change the values of these two attributes. Hence, data permissions must be granted taking the progress of the corresponding process instance into account as well. Which input and data fields, shall be displayed not only depends on the user working on this activity, but also on the *processing state of the respective process instances*. Note that this resulty in a multiplicity of different user forms whose manual implementation would be cumbersome and error-prone.

Property 19 (Data authorization depending on the progress of the process)
Which input and data fields shall displayed not only depends on the user executing a (form-based) activity, but also on the processing state of the respective process instance.

Despite the dependencies existing between user assignment and data permission, we need to differentiate between them. In particular, a user normally not involved in process execution, may have the data permission to set or change attribute values solely based on optional activities. For example, a `manager` is allowed to read `applications`, make a `comment`, and change the `decision`. However, the workitem for the corresponding mandatory activity is assigned to an `employee`, but not to the `manager`.

Property 20 (Differentiating between data and process authorization)
There is a difference between data and process authorization. While data authorization allows for the execution of optional activities, process authorization controls who is responsible for executing mandatory activities that are usually required to proceed with the flow of control.

Note that one and the same user may have both data and process authorizations. Then, the respective form-based activity should comprise both mandatory input fields and optional ones.

3.2 Property Verification

The previous section has given insights into processes currently not adequately supported by process management technology. In particular, it has identified basic properties of these processes. This section backs up our findings based on an *extensive literature study*. In particular, we compare our analysis results with existing literature in order to ensure the *relevance* and *completeness* of the identified properties. In addition, we confirm that a support of the properties is needed in a variety of application scenarios. Finally, we add important properties identified by other researchers, but not yet contained in our property list.

We focus on process modeling, execution and monitoring, but exclude properties related to the ad-hoc change and evolution of business processes [RW12]. Further, we contrast different application scenarios with the total set of properties. This way, we can demonstrate that the properties are *related to each other* and a broad support is required by a variety of processes from different application domains; i.e., *generalization* is possible.

According to the insights gained in our analysis, advanced process support necessitates *object-awareness* in many application cases; i.e., business processes and business objects cannot be treated independently from each other. Note that this has been confirmed by other work as well [LBW07, RL03, VRv08, MRH07, vWG05, vBEW00, RDtI07, RDtI09a, KG07], although a holistic support and understanding of object-awareness is still missing. In particular, we must understand the inherent relationships that exist between the different business perspectives and the aspects they cover (cf. Chapt. 1).

An aggregation of the different properties into more abstract categories results in five major *characteristics* of object-aware process support (cf. Fig. 3.14): First, object-aware processes shall be based on *two levels of granularity*. On one hand, *behavior* of individual object instances needs to be considered during process execution; on the other, *interactions* among different object instances must be taken into account as well. Since the progress of a process depends on available object instances (and their attribute values), in addition, process execution should be *data-driven*. Finally, a *flexible activity execution* is crucial. In particular, activities do not have to coincide with particular process steps.

Object-aware processes must consider all elements of the underlying data structure, which comprises *objects*, *object attributes*, and *object relations*. More precisely, objects and object relations also constitute proper guidelines for choosing the granularity of processes and sub-processes respectively; i.e., the behavior of the business objects involved in a process must be taken into account during process execution and the interactions between business objects must be adequately covered; i.e., the behavior of individual objects must be coordinated with the one of related objects. Thus, any process structure (cf. Sect. 3.1) should comply with the corresponding data structure.
In addition to user worklists, which provide access to the process activities to be executed, *integrated access to related objects* is another fundamental characteristic of object-aware processes. In particular, authorized users should be allowed to access and manage business data (i.e., business objects) at any point in time. In turn, this requires advanced support with respect to the integration of users.

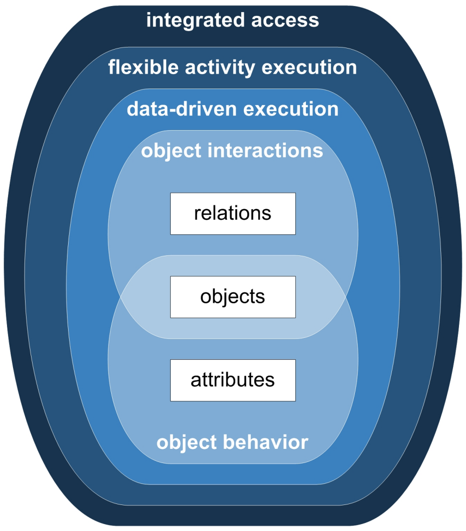


Figure 3.14: Main characteristics of object-aware processes (according to [KR11b])

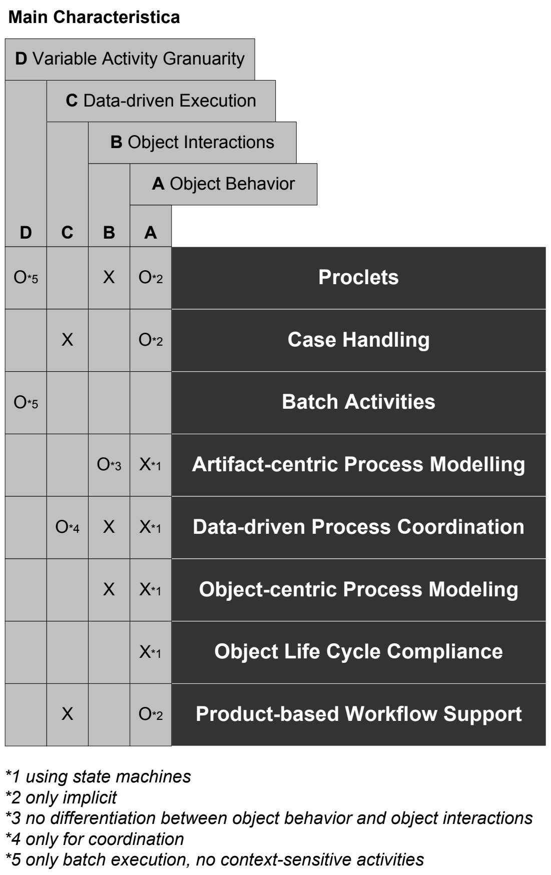
3.2.1 Relevance and Completeness

Besides PHILharmonicFlows, there exists other pioneering work targeting at object-aware process support (cf. Fig. 3.15): artifact-centric modeling [LBW07], product-based workflow support [RL03, VRv08], data-driven process coordination (Corepro) [MRH07, MRH08a], case handling [vWG05], Proclets [vBEW00], object-centric processes [RDtI07, RDtI09a], and object life cycle compliance [KG07]. Although these approaches show several limitations, four characteristics of object-aware processes are addressed by at least one of these approaches. This section summarizes these approaches, while details are provided in Chapt. 4.

The approaches depicted in Fig. 3.15 confirm the relevance of a tight integration of processes and data. In the following, we discuss how the characteristics object behavior, object interactions, data-driven execution, and varying activity granularity are addressed by them.

Object Behavior

To ensure consistency between process and object states, imperative approaches have been enriched with *object life cycles (OLC)*. Examples of such approaches include artifact-centric modeling, data-driven coordination, object-centric processes, and object life cycle compliance. An OLC defines object behavior; i.e., the processing states of an object and the transitions between them. The latter are enabled after executing corresponding activities, which are therefore associated with pre-/post-conditions on objects states; i.e., the execution of an activity depends on the current state of an object and its completion triggers a subsequent state.
None of these approaches considers object attributes. Hence, it is unclear which attribute values must be available when a subsequent state becomes activated; i.e., mandatory information is not considered. Consequently, if certain pre-conditions cannot be met during run-time, it is not possible to dynamically react on this; i.e., a data-driven OLC execution is supported by none of the approaches.
Proclets, case handling, and product-based workflows allow for another kind of object behavior. Although business objects are not explicitly considered, each process may be aligned with an



Main Characteristica

D Variable Activity Granuarity
C Data-driven Execution
B Object Interactions
A Object Behavior

D	C	B	A	
O*5		X	O*2	**Proclets**
	X		O*2	**Case Handling**
O*5				**Batch Activities**
		O*3	X*1	**Artifact-centric Process Modelling**
	O*4	X	X*1	**Data-driven Process Coordination**
		X	X*1	**Object-centric Process Modeling**
			X*1	**Object Life Cycle Compliance**
	X		O*2	**Product-based Workflow Support**

**1 using state machines*
**2 only implicit*
**3 no differentiation between object behavior and object interactions*
**4 only for coordination*
**5 only batch execution, no context-sensitive activities*

Figure 3.15: Characteristics supported by existing approaches

object; i.e., it is up to the process designer to decide for an object-specific process granularity. Proclets are based on black-box activities, whereas case handling and product-based workflows allow defining activities in tight relation to atomic data elements (that may be considered as object attributes). This way, it becomes possible to determine the order in which object attributes shall be written and to define what valid attribute settings are.

Object Interactions

An asynchronous coordination of process instances is enabled; i.e., lightweight processes communicate with each other via messages. Although this is not necessarily based on the underlying data structure (i.e., processes may be defined at an arbitrary level of granularity), processes may be aligned according to object types.
Using artifact-centric process modeling, explicit relations between artifacts can be defined. However, their life cycles are treated independently from each other. In addition to the support of object behavior, pre-conditions for activity execution allow coordinating artifact lifecycles. Consequently, no separation between object behavior and object interactions is made.
Object-centric process modeling as well as data-driven process coordination allow defining object interactions based on OLCs. Thus, it becomes possible to asynchronously execute object-specific process instances and to coordinate them at certain points during their execution; i.e., process synchronization complies with the underlying data structure. Thereby, a varying num-

ber of instances can be created.

Data-driven Execution

Regarding Case Handling, activities are realized as forms processing atomic data elements. The latter may be free, mandatory, or optional. An activity is completed if all of its mandatory data elements are set. Thereby, a particular data element may be read or written in the context of several activities. Since a data element, mandatory for a particular activity, may be optional for a preceding one, values may be assigned before they become mandatory for process execution. This way, a data-driven process execution is enabled.
Product-based workflows use a product data model, which is a tree-like structure comprising atomic data elements and related operations. The latter may comprise input data elements and refer to exactly one output data element. An operation will be executable if all input data elements have assigned values. Finally, the product is considered as processed as soon as a value for the top element of the product data model (i.e., the root element) has become available.
Regarding data-driven process coordination, processes themselves are activity-driven. More precisely, activities are executed in the context of OLC state transitions. The activation of a subsequent state then depends on the completion of the process associated with the respective state transition. Opposed to this, process synchronization relies on a data-driven approach: Whether or not a particular state of an object is activated may depend on the activation of specific states of other (related) objects.

Varying Activity Granularity

[SOSS05] allows for a batch execution of activities; i.e., the same activities of different process instances may be grouped and then processed together. A similar approach is followed by Proclets. In particular, it is possible to create several Proclet instances in one go. The execution of a set of instance-specific activities, however, is not possible.
To our best knowledge, currently, there exists no approach that allows for the flexible composition of activities relating to different process instances (i.e., it is not possible to support context-sensitive activities involving a number of (selected) process instances).

3.2.2 Relatedness and Generalisation

To emphasize the practical benefit of the identified characteristics, Fig. 3.16 shows their relevance in the context of the *application scenarios* discussed in literature. Note that existing approaches partially consider the same scenarios, but address different characteristics. Hence, this indicates that the characteristics are related to each other and required in the context of a variety of processes from different application domains. For each application domain, the relevance of the characteristics is illustrated in the rows on the bottom of Fig. 3.16. For example, consider application scenario *order processing* as illustrated in Column 5. Order processing has been used as illustrating scenario in the context of case handling, batch activities, and business artifacts. While case handling addresses the need for data-driven process execution, business artifacts emphasize object behavior and interactions. Finally, [SOSS05] discusses the need for executing multiple activities in one go. Altogether, Fig. 3.16 indicates that integrated and consistent support is needed for the identified characteristics.

According to Fig. 3.16, each characteristic of object-aware processes has been addressed by at least one existing approach. However, note that Fig. 3.16 does not distinguish between

Main Characteristica

D Variable Activity Granuarity
C Data-driven Execution
B Object Interactions
A Object Behavior

Application Scenarios

D	C	B	A		Human Resources (Hiring People)	Academic(ConferenceManagement)	Insurance (Claim Processing)	Development Processes	Order Processing	Education (Course Management)	Health Care (Patient Treatment)	Hotel Business (Guest Check)	Inspections
O*5		X	O*2	**Proclets**	X	X	X	X					
	X		O*2	**Case Handling**			X		X	X	X		
O*5				**Batch Activities**					X	X			
		O*3	X*1	**Artifact-centric Process Modelling**		X			X		X	X	
	O*4	X	X*1	**Data-driven Process Coordination**				X					
		X	X*1	**Object-centric Process Modeling**							X		X
			X*1	**Object Life Cycle Compliance**			X						
	X		O*2	**Product-based Workflow Support**			X	X					
					X	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
					X	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
					X	**X**	**X**	**X**	**X**	**X**	**X**		
					X	**X**	**X**	**X**	**X**				

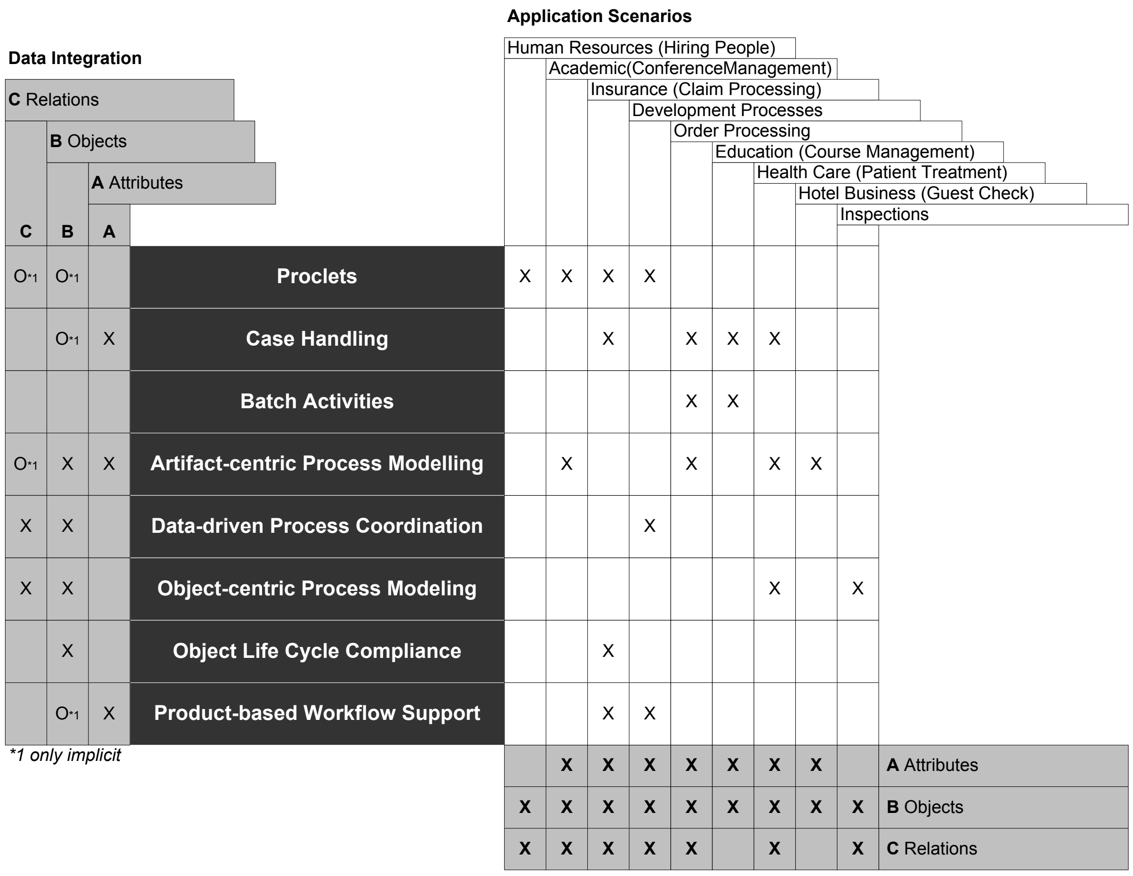
	Human Resources (Hiring People)	Academic(ConferenceManagement)	Insurance (Claim Processing)	Development Processes	Order Processing	Education (Course Management)	Health Care (Patient Treatment)	Hotel Business (Guest Check)	Inspections
A Object Behavior	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
B Object Interactions	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
C Data-driven Execution	**X**	**X**	**X**	**X**	**X**	**X**	**X**		
D Variable Activity Granularity	**X**	**X**	**X**	**X**	**X**				

**1 using state machines*
**2 only implicit*
**3 no differentiation between object behavior and object interactions*
**4 only for coordination*
**5 only batch execution, no context-sensitive activities*

Figure 3.16: Application scenarios discussed by existing approaches (cf. [KR11b, KR11d])

process modeling and execution.
Besides the discussed approaches, there exist systems supporting of single characteristics. As examples consider *project management systems* or *enterprise resource planning systems* having an integrated workflow component (e.g., SAP Business Objects) [SSB09]. However, these systems lack a comprehensive support of object-aware processes.
Finally, note that there is no approach integrating application data in a consistent way; i.e., objects, attributes and relations must be considered (cf. Fig. 3.17).

Providing integrated access to business data on one hand and business processes on the other requires advanced concepts for *user integration*. In particular, the assignment of users to processes and activities must not be solely based on user roles, but take the processed application data into account as well. This has been recognized in [RzM98, RzM04] which allow for *activity-based object-individual resolution of roles*. In this context, the relations between users and object instances are considered as well (see also [BBU99, HW04, KKC02]). Finally, [ST97] and [Bot02] emphasize that user assignments should consider the permissions for accessing the data processed by an activity as well. Whether or not a permission for accessing data or executing a function is granted, depends on the specific use case; i.e., the context in which it is accessed or executed.



Data Integration: **C** Relations, **B** Objects, **A** Attributes

Application Scenarios

C	B	A		Human Resources (Hiring People)	Academic(ConferenceManagement)	Insurance (Claim Processing)	Development Processes	Order Processing	Education (Course Management)	Health Care (Patient Treatment)	Hotel Business (Guest Check)	Inspections
O*1	O*1		**Proclets**	X	X	X	X					
	O*1	X	**Case Handling**			X		X	X	X		
			Batch Activities					X	X			
O*1	X	X	**Artifact-centric Process Modelling**		X			X		X	X	
X	X		**Data-driven Process Coordination**				X					
X	X		**Object-centric Process Modeling**							X		X
	X		**Object Life Cycle Compliance**			X						
	O*1	X	**Product-based Workflow Support**			X	X					
			A Attributes		**X**	**X**	**X**	**X**	**X**	**X**	**X**	
			B Objects	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
			C Relations	**X**	**X**	**X**	**X**	**X**		**X**		**X**

*1 only implicit

Figure 3.17: Data integration in existing approaches

The need for accessing process-related application data at any point in time during process execution is emphasized in [WSML02]. *Instance-based user groups* are suggested, which grant access to all data of the process instances a particular user is involved in. A similar concept is presented in [vWG05].
A more flexible execution of activities is provided by declarative approaches [vPS09, PWZ$^+$12, WRZW09], which do not enforce users to work on activities in a strict execution order (as imposed in traditional imperative approaches). Instead, processes are defined in terms of a set of activities and constraints prohibiting undesired execution orders of these activities. However, declarative approaches provide only limited support for object-aware processes [KWR10b]. Finally, [SOSS05] emphasizes the need for batch activities; i.e., activities of different process instances may be grouped. However, the grouping only considers activities but does not take data into account.

4 State-of-the-Art

This chapter discusses issues that emerge when applying existing process management paradigms to implement the properties introduced in Chapt. 3. First, we focus on imperative and declarative process support paradigms (cf. Sects. 4.1 and 4.2), and then consider extensions of these two fundamental paradigms (cf. Sect. 4.3). Finally, we evaluate specific work for user integration, relevant in the context of this thesis (cf. Sect. 4.4).[1]

4.1 Imperative Process Support Paradigms

There has been a long tradition of modeling business processes in an imperative way [DR09, RW12]. Process languages supporting this paradigm, for example, include BPMN, EPC and BPEL. Imperative processes are specified in terms of directed process graphs [WRRM08]. Process steps then correspond to activities that are connected by arcs expressing precedence relations (cf. Fig. 4.1) [TRI09]. For control flow modeling, different patterns exists, e.g., sequential, alternative and parallel routing, or loop backs [vtKB03].

Imperative approaches only provide limited support for the properties of object-aware processes. Compared to the main characteristics of object-aware processes (cf. Sect. 3.2), the imperative approach can be characterized by *hidden information flows (A)*, *flow-based triggering of activities (B)*, *actor expressions (C)*, *fixed activity granularity (D)*, and *arbitrary process granularity (E)*. In the following, we evaluate to what extend the imperative approach is able to support object-aware processes showing these characteristics (cf. Fig. 4.2).

[1]The chapter is based on the following referred paper:
[KWR10b] V. Künzle, B. Weber, and M. Reichert. Object-aware Business Processes: Properties, Requirements, Existing Approaches. *Technical Report*, 2010
[KWR10a] V. Künzle, B. Weber, and M. Reichert. Object-aware Business Processes: Fundamental Requirements and their Support in Existing Approaches. *International Journal of Information System Modeling and Design (IJISM)*, 2(2):9–46, 2010
[KR11d] V. Künzle and M. Reichert. Striving for Object-Aware Process Support: How Existing Approaches Fit

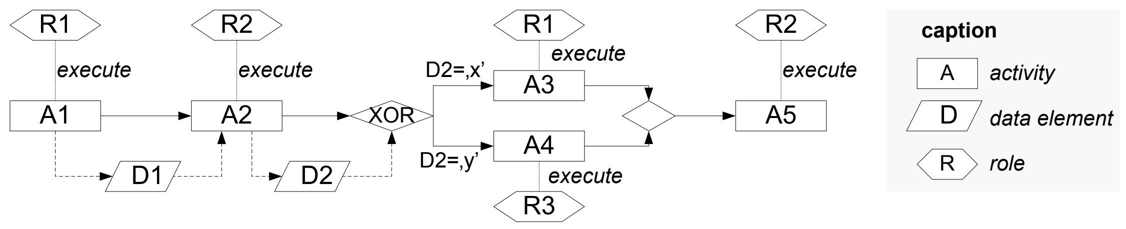
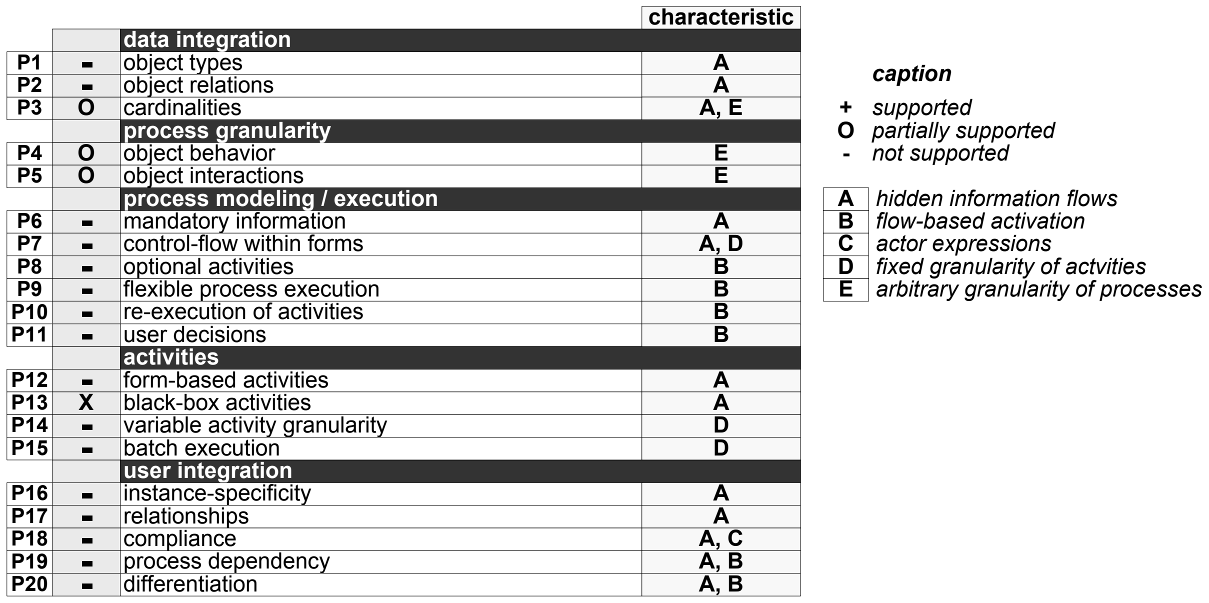


Figure 4.1: Imperative process modeling (according to [KWR10a])



			characteristic
		data integration	
P1	-	object types	**A**
P2	-	object relations	**A**
P3	O	cardinalities	**A, E**
		process granularity	
P4	O	object behavior	**E**
P5	O	object interactions	**E**
		process modeling / execution	
P6	-	mandatory information	**A**
P7	-	control-flow within forms	**A, D**
P8	-	optional activities	**B**
P9	-	flexible process execution	**B**
P10	-	re-execution of activities	**B**
P11	-	user decisions	**B**
		activities	
P12	-	form-based activities	**A**
P13	X	black-box activities	**A**
P14	-	variable activity granularity	**D**
P15	-	batch execution	**D**
		user integration	
P16	-	instance-specificity	**A**
P17	-	relationships	**A**
P18	-	compliance	**A, C**
P19	-	process dependency	**A, B**
P20	-	differentiation	**A, B**

caption

+	*supported*
O	*partially supported*
-	*not supported*
A	*hidden information flows*
B	*flow-based activation*
C	*actor expressions*
D	*fixed granularity of actvities*
E	*arbitrary granularity of processes*

Figure 4.2: Evaluation of the imperative approach (according to [KWR10a])

4.1.1 Hidden Information Flows

As first characteristic of the imperative process support paradigm consider *hidden information flows (A)*. Usually, the imperative approach allows for the explicit definition of the data flow between activities based on atomic data elements [RM09]. The latter are connected with activities (and their input/output parameters) through data arcs or with routing conditions (cf. Fig. 4.1). Activities themselves are regarded as black boxes; i.e., application data comprising object types and object relations is usually managed within the applications invoked by activities during run-time. In particular, there exists no explicit link between activities and the object instances (and object attributes) processed by them. Hence, which data is actually accessed or changed during activity execution does not become transparent. Consequently, it is not possible to define mandatory information and to enable an automatic generation of form-based activities. Altogether, this characteristic affects properties related to the business perspectives *data*, *processes*, *activities*, and *users* (cf. Chapt. 1).

Together. In *1st Int'l Symposium on Data-driven Process Discovery and Analysis (SIMPDA'11)*, 2011

Data. Imperative PrMS are unaware of the object instances actually accessed during process execution. Thus, data integration based on object types, attributes and relations is not supported (i.e., Props. 1 and 2 are not met). In addition, cardinalities between object instances are not transparent. However, as we will discuss, it is up to the process modeler to create a workaround by defining a separate sub-process for each object type involved (i.e., Prop. 3 is not fully met).

Processes. A particular activity usually accesses data provided by preceding activities according to the modeled data flow. However, the PrMS has no control on whether required data changes are accomplished; i.e., mandatory information in the context of an activity execution can neither be ensured nor monitored by the PrMS (i.e., Prop. 6 is not met). Since mandatory information is not transparent, the internal control-flow (i.e., Prop. 7) of a form-based activity cannot be expressed.

Activities. If a data element, an activity wants to read, is not written by preceding activities, process execution might be blocked or even crash during run-time. In turn, if the flow of data between activities (and the application services invoked by them) is not explicitly specified within a process model, corresponding process instances might proceed although required data is missing. Consequently, it is not possible to automatically invoke a form-based activity for requesting missing data from users (i.e., Prop. 12 is not met). Finally, black-box activities and hence application integration are supported (i.e., Prop. 13 is met).

Users. Regarding user integration, it does not become transparent, which object instances are accessed when executing a particular activity. Hence, authorization may only be related to activities. As a consequence, instance-specific authorization, enabling different permissions for the same activity depending on the processed object instance, is not supported (i.e., Prop. 16 is not met). Since the organizational model is usually strongly separated from application data, in addition, the relationships between users and processed objects cannot be considered (i.e., Prop. 17 is not met). Moreover, using an imperative PrMS, integrated access to application data is not possible. For this reason, it cannot be guaranteed that a user who owns the permission for executing an activity is also authorized to access attributes of the object instances processed by this activity; i.e., compliance between data and process authorization is not considered (i.e., Prop. 18 is not met). The same applies to properties Prop. 19 (process dependency; i.e., data authorization dependent on the progress of the process) and Prop. 20 (differentiation; i.e., differentiation between data and process authorization).

4.1.2 Flow-based Triggering of Activities

Another characteristic of the imperative process support paradigm is its *flow-based triggering of activities (B)*. Each process step corresponds to an activity, which is mandatory for process execution (except it is contained in a conditional execution path skipped during run-time). Moreover, whether or not an activity becomes activated depends on the state of preceding activities, i.e., a particular activity may be enabled if its preceding activities are completed or cannot be executed anymore (except loop backs). No direct support exists for verifying whether the (semantic) goals of a process can be achieved [RKG06, RDtI07, GS07]. In this context, some approaches allow checking the compliance of imperative process models with global regulations or semantic constraints [LKRM$^+$10, LRMD10]. For example, they define pre- and post conditions for activities in relation to application data [HTG12]. If the pre-conditions of an activity cannot be met during run-time, process execution will be blocked. In this context, it is no longer sufficient to only postulate certain attribute values for executing a particular activity. In addition, it must be also possible to dynamically react on emerging attribute values; i.e., to trigger activities based on data available rather than on the activities already executed. Altogether, activity execution is exclusively flow-based. This characteristic affects properties belonging to the business categories *processes* and *users*.

Processes. Data may only be accessed when executing activities according to the defined control flow; i.e., data must not be accessed asynchronously to process execution. Consequently, there exists no explicit support for *optional activities* that allow for a data access at any point in time (i.e., Prop. 8 is not met). However, this can be simulated using the workaround described in Ex. 4.1.

Example 4.1 (Workaround 1: Optional activities):
Optional activities may be added as conditional branches in different regions of a process model (cf. Fig. 4.3).

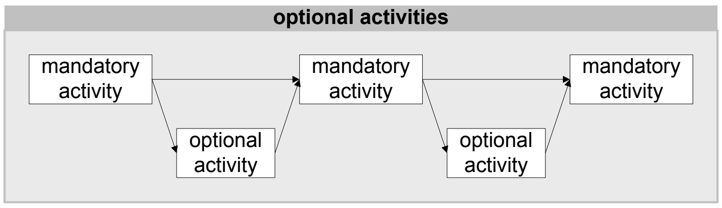


Figure 4.3: A workaround "simulating" optional activities (according to [KWR10a])

As a drawback of this workaround, spaghetti-like process models containing a high number of redundant activities might result. Besides this, optional activities cannot be distinguished from mandatory ones. Note that without such a workaround, required changes of application data would have to be accomplished directly within the application systems. When bypassing the PrMS or application system, however, inconsistencies with respect to attributes, redundantly maintained in both systems, might occur. Worst case, this might result in run-time errors or inconsistent process executions.

Regarding process flexibility provided by imperative approaches [RRD09], *flexible process execution* as required in the context of object-aware process management is not explicitly supported. More precisely, since the activation of an activity depends on the completion of other activities, skipping an activity if required output data is already available is not possible (i.e., Prop. 9 is not met). Again, a workaround exists (cf. Ex. 4.2).

Example 4.2 (Workaround 2: Flexible process execution):
Consider Fig. 4.4a. Using XOR-Splits, process data elements may be evaluated before and after activity execution. If required object attribute values have already been made available before triggering the activity, which usually writes them, the skipping of this activity is simulated by not choosing the corresponding branch.

Note that there exist approaches like ADEPT2 [RHD98, Rei00, RRKD05, RD09], which enable process flexibility by supporting dynamic process changes (e.g., to add or move activities)

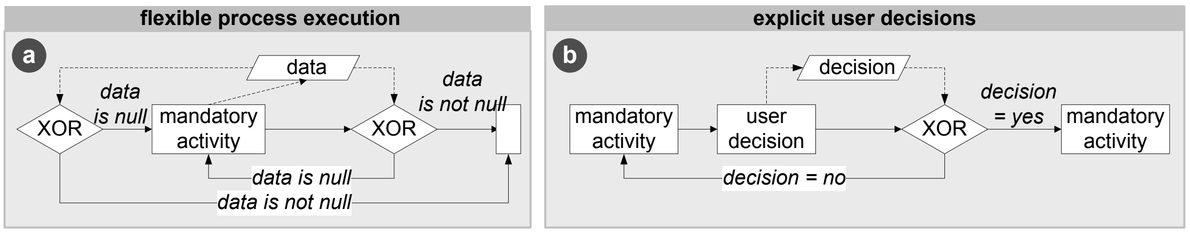


Figure 4.4: Workarounds for flexible execution and user decisions (according to [KWR10a])

during run-time. However, to set a focus, we exclude issues related to process changes and process evolution (see [RRD09, WRRM08, WSR09, RW12] for respective surveys). Instead, this thesis focuses on process modeling and execution.

Activity activation solely depends on the completion of other activities. Hence, there is no explicit support for *user decisions* (i.e., Prop. 11 is not met). Further, there is no inherent support for *re-executing an activity* as long as the user does not commit its completion (i.e., Prop. 10 is not met).

Example 4.3 (Workaround 3: Activity re-execution and user decisions):
Consider Fig. 4.4b. User decisions are encapsulated by black-box activities, which write data elements according to the defined data flow. These data elements are then evaluated using an XOR-split to decide whether to proceed with process execution or to initiate a backward jump (using a loop).

Users. Since data may only be accessed when executing mandatory activities, imperative approaches lack a support for synchronizing data processing and process execution. Thus, data authorization is not properly taken into account; i.e., data authorizations do not depend on process states. Note that this is outside the scope of PrMS and therefore implemented in the underlying application systems. Accordingly, it is not possible to ensure that data authorization depends on the progress of the corresponding process. (i.e., Prop. 19 is not met). Finally, no *differentiation between process and data authorization* can be made (i.e., Prop. 20 is not met). In particular, users may only execute mandatory activities added to their worklist.

4.1.3 Actor Expressions

As another characteristic, human activities are associated with *actor expressions*; e.g., user roles (C) [RMR08, RMR09]. Based on such expressions, activities will be assigned to worklists of authorized users during run-time. In particular, this enables process coordination among users. Furthermore, when a human activity becomes enabled, a corresponding work item is added to the worklists of authorized users; i.e., a process-oriented view allows executing activities by the right users at the right point in time. However, this affects the *compliance between process and data authorization*. In particular, it is not possible to ensure that each user, who must execute a particular activity in order to proceed with process execution, may also change the data processed during the execution of this activity (i.e., Prop. 18 is not met).

4.1.4 Fixed Activity Granularity

As another characteristic of imperative processes consider *fixed activity granularity* (D). In particular, activities are associated with a specific business function implemented at build-time, and thus having a fixed granularity. Hence, an activity must be executed in a defined context. However, such a rigid execution is not always adequate. In certain cases, an activity needs to be repeated in an ad-hoc manner, or it has to be executed in advance or first be stopped and then be caught up at a later point in time [vWG05]. Conventional PrMS do not allow for such kind of flexibility. Furthermore, users are usually involved in the execution of multiple instances of a particular process type. Thus, work items of their worklist may refer to activities of the same type. Each of them must then be processed separately in the PrMS, which does not always comply with common work practice. In summary, the isolated execution of process instances in existing PrMS is by far too inflexible [SOSS05]. This characteristic affects properties related to the business perspectives *processes* and *activities*.

Processes. All activities are considered as black box; i.e., it does not become transparent which object instances are processed by a particular activity. Furthermore, it is not possible to control the flow logic within a form; i.e., it is not possible to influence the order in which data elements shall be processed during the execution of an activity (i.e., Prop. 7 is not met).

Activities. Each process step refers to exactly one activity that is pre-defined at build-time. Hence, different work practices cannot be supported. By contrast, for object-aware processes, activities may have *varying granularity*; i.e., it shall depend on the user, executing an activity, to decide which and how many object instances shall be processed. In this context, data authorization does not only depend on the user executing this activity, but on the processing states of the involved object instances as well. The resulting high number of different activities is not supported (i.e., Prop. 14 is not met); i.e., *context-sensitive activities* are not supported. Since it is not possible to execute several activities in one go, in addition, *batch execution* is not supported (i.e., Prop. 15 is not met).

4.1.5 Arbitrary Process Granularity

Granularity issues are not properly addressed by imperative PrMS; i.e., processes, sub-processes, and activities may be modeled at *arbitrary levels of granularity* (E). While certain activities are only processing one object instance, others may process several object instances of the same or different type. Neither a consistent methodology nor practical guidelines exist for process modeling [RL03], often resulting in inconsistent or non-comparable models. Furthermore, when modeling and executing processes in PrMS, no direct support for considering the underlying data structure is provided (i.e., the objects and their relations). This has effects on properties of the business perspectives *processes* and *data*.

Processes. Imperative approaches do not distinguish between the behavior of individual object instances and the processes coordinating them. Generally, there exists no elaborated modeling methodology giving advice on the number of object types to be handled within one process definition. A process is either defined at a *coarse- or fine-grained level*. When choosing a *fine-grained modeling style*, each process definition is aligned with exactly one object type. This way one can ensure that corresponding process instances access one particular object instance of the respective object type at run-time. For this purpose, either one data element for routing the object-ID or several data elements (corresponding to the object attributes) are added to the process model. The activity-centred paradigm of imperative approaches, however, is not appropriate for supporting *object behavior* (i.e., Prop. 4 is only partially met). In particular, hidden information flows and the flow-based activation of activities inhibit the dynamic adaptation of the control-flow based on available data. Furthermore, process instances are executed in isolation to each other [vBEW00]. Neither dependencies between instances of different process types nor between instances of the same process type can be defined at a reasonable semantical level. Often, the modeling of subordinate processes serves as a workaround. However, in existing PrMS the execution of subordinate process instances is tightly synchronized with their superordinate process instance; i.e., the latter is blocked until the sub-process instances are completed. Thus, in current PrMS, neither aggregated activities nor more complex synchronization dependencies can be adequately handled [vBEW00] (i.e., Prop. 5 is not fully met).

A *coarse-grained modeling style*, prohibits *fine-grained control* regarding *object behavior* (i.e., Prop. 4 is not met). Processes are only defined in terms of activities and *interactions between object instances* are not considered (i.e., Prop. 5 is not met). Process support involving different object instances is provided using sub-processes. Thereby, a sub-process is associated with an activity of the higher-level process instance. However, it is not possible to define relations and synchronization dependencies between different sub-process definitions. Consequently, processes comprising the explicit definition of *object interactions* are not supported (i.e., Prop. 5 is not met). Note that this limitation can be addressed by "multiple-instantiation patterns" [vtKB03, RR06], which allow specifying the number of instances for a respective activity either at build- or run-time. Regarding *multiple-instance activity patterns*, new sub-process instances may only be created as long as subsequent activities have not been started. Thus, lower-level process instances (i.e., sub-process instances) can only be created at a specific point during the execution of the higher-level process instance. Furthermore, except for one variant of the multiple-instantiation pattern, sub-process instances cannot be executed asynchronously to the higher-level process instance. Using *multiple-instantiation patterns with synchronization* (cf. Fig. 4.5a), each sub-process instance must either be completed or skipped before subsequent activities of the higher-level process instance may be triggered. In turn, using *multiple-instantiation without synchronization*, the results of these sub-process executions are not relevant for progressing the higher-level process instance (cf. Fig. 4.5b). Finally, interdependencies between sub-processes, executed asynchronously to each other (cf. Fig. 4.5c), cannot considered.

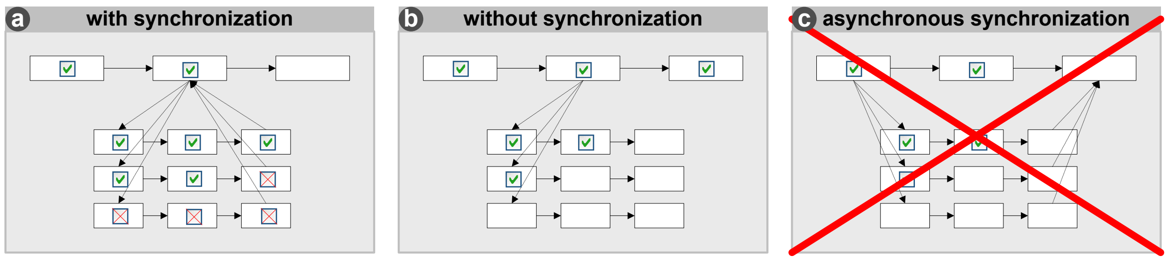


Figure 4.5: Sub-process execution based on multiple-instantiation

Data. Since an activity may be linked to several object types (and object flows are hidden), it is a difficult task to ensure consistency between process and data modeling. In particular, when modeling a process, the creation of object instances cannot be restricted to a varying and dynamic number of object instances based on *cardinalities*. Using multiple-instantiation patterns, however, it is possible to restrict the number of sub-processes in accordance to the specified cardinality of related object types. Thus, cardinalities may only be manually considered by the process modeler without system support (i.e., Prop. 3 is partially met).

4.2 Declarative Process Support Paradigms

Declarative approaches suggest a fundamentally different way of describing business processes [vP06, vPS09, FLM$^+$09, FMR$^+$10, WRZW09, PWZ$^+$12]. While imperative models specify how things have to be done, declarative approaches focus on the logic that governs the interplay of actions in the process by describing (1) the *activities* that may be performed and (2) the *constraints* prohibiting undesired behavior. In the example from Fig. 4.6, activities `A2` and `A3` may only be executed after finishing `A1`. Furthermore, `A2` and `A3` are mutually exclusive.

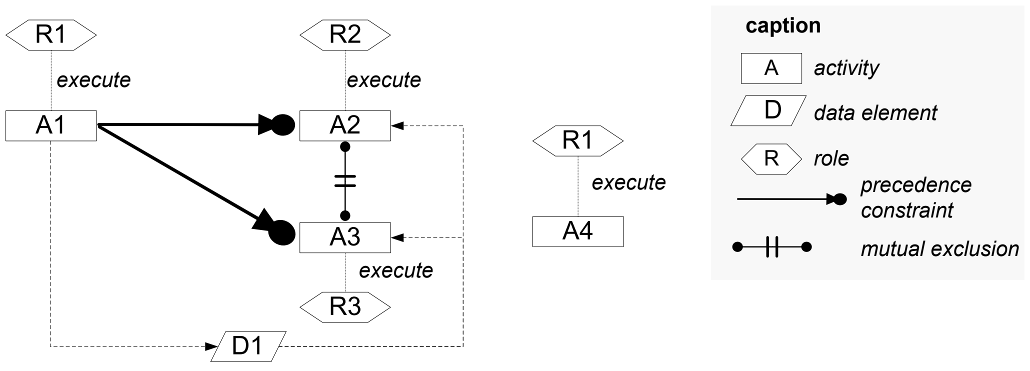


Figure 4.6: Declarative modeling approach [Pes08]

Declarative modeling approaches only provide rudimentary support for object-aware processes. Several of their characteristics correspond to the ones of imperative approaches: *hidden information flows (A)*, *actor expressions (C)*, *fixed activity granularity (D)*, and *arbitrary process granularity (E)*. However, they differ from imperative approaches in respect to activity activation. While imperative approaches pursue a flow-based activation, declarative approaches rely on a *constraint-based coordination (B)* (cf. Fig. 4.7). This leads to a better support of optional activities compared to imperative approaches. However, the extensions introduced for imperative approaches (e.g., [vBEW00]) are not applicable to declarative ones. To avoid redundancies, we only discuss the main differences between imperative and declarative approaches.

4.2.1 Constraint-based Coordination of Activities

Characteristic to declarative processes is the *constraint-based coordination of activities (B)*. Imperative models take an "inside-out" approach by requiring all execution alternatives to be explicitly specified in the model. In turn, declarative models take an "outside-in" approach: constraints implicitly specify execution alternatives as all valid alternatives have to satisfy the

			characteristic
		data integration	
P1	-	object types	A
P2	-	object relations	A
P3	-	cardinalities	A, E
		process granularity	
P4	O	object behavior	E
P5	O	object interactions	E
		process modeling / execution	
P6	-	mandatory information	A
P7	-	control-flow within forms	A, D
P8	+	optional activities	B
P9	-	flexible process execution	B
P10	-	re-execution of activities	B
P11	-	user decisions	B
		activities	
P12	-	form-based activities	A
P13	X	black-box activities	A
P14	-	variable activity granularity	D
P15	-	batch execution	D
		user integration	
P16	-	instance-specificity	A
P17	-	relationships	A
P18	-	compliance	A, C
P19	-	process dependency	A, B
P20	-	differentiation	A, B

caption

+	*supported*
O	*partially supported*
-	*not supported*

A	*hidden information flows*
B	*constraint-based activation*
C	*actor expressions*
D	*fixed granularity of actvities*
E	*arbitrary granularity of processes*

Figure 4.7: Evaluating the declarative approach (according to [KWR10a])

defined constraints [Pes08]. Consequently, adding constraints means discarding some execution alternatives. This results in a coarse up-front specification of a process, which can be refined iteratively during run-time. Generally, constraints can be roughly divided into three classes [SSO05, vP06, vPS09]: constraints restricting the *selection* of activities (e.g., minimum/maximum occurrence of activities, mutual exclusion), the *ordering* of activities, and the use of *resources* (e.g., execution time of activities). This characteristic affects properties related to the business perspective *process*.

Proper support for *optional activities* is provided, i.e., activities may be considered as optional as long as no constraint enforces their execution (i.e., Prop. 8 is met). Mandatory activities cannot be skipped if the required data is already available (i.e., Prop. 9 is not met). However, the following workaround may be applied:

Example 4.4 (Workaround 4: Flexible process execution):
Specific data constraints may be introduced to check whether all required data are available. If this is not the case, respective activity execution is then blocked.

Activities cannot be *re-executed* based on user commitments (i.e., Prop. 10 is not met).

Example 4.5 (Workaround 5: Re-execution of activities):
Using a specific constraint, re-execution of a particular activity is possible as long as no activity providing the user commitment has been executed.

4.2.2 Arbitrary Process Granularity

Like for imperative processes, granularity issues are not properly addressed; i.e., processes, sub-processes, and activities may be modeled at *arbitrary level of granularity (E)*. However, partial support for integrating process instances can be achieved based on sub-processes [ZSPW12]: For declarative approaches not supporting multiple instantiations, *cardinalities* to higher-level process definitions cannot be expressed (i.e., Prop. 3 is not met).

4.3 Extensions of Traditional Approaches

Several approaches have already perceived the inability of current technologies to adequately capture the relation between process and data, and concepts targeting at a better integration of these two perspectives have been suggested: artifact-centric modelling [NC03, LBW07, GS07], product-based workflow-support [RL03, VRv08], data-driven process coordination [MRH07, MRH08a], case handling [vWG05], object-centric processes [RDtI07, RDtI09a], and Proclets [vBEW00].

This section evaluates to what extent these approaches support the properties introduced in Sect. 3.1. As illustrated in Fig. 4.8, each property is addressed by at least one existing approach. Although the approaches we analyzed show limitations, they can be considered as pioneer work towards object-aware process support. However, none of them covers all properties in a comprehensive and integrated way. Further, note that Fig. 4.8 does not make any difference between process modeling and process execution. Though some approaches (e.g., the artifact-centric modeling) provide rich capabilities for process modeling, they do not explicitly address run-time issues (or at least do not treat them explicitly). Since all these approaches correspond to extensions of traditional imperative or declarative process support paradigms, it is impossible to extend them for covering the comprehensive set of the introduced properties.

4.3.1 Case Handling

We first evaluate to what extend Case Handling (CH) [vWG05, GRv08, RRvdA03] and the Flower CH tool [Ath02] cover the properties of object-aware processes. As illustrated in Fig. 4.9, CH is a data-driven process support paradigm for which activities are represented in terms of *user forms* that comprise a number of input fields. The latter refer to *atomic data elements* which are either defined as mandatory, restricted, or free. Compared to imperative and declarative approaches, the main differences lie in the *integration of application data*, the *data-driven execution paradigm*, and an *advanced role concept*.

Processes. Processes may be defined at *arbitrary level of granularity*; i.e., in a coarse- or fine-grained manner. A *coarse-grained* definition includes data elements corresponding to different objects. Alternatively, at a more *fine-grained* level, a case is treated in tight accordance with an object; i.e., each case then refers to an object type. This enables support of *object behavior* (i.e., Prop. 4 is met). In turn, when modeling processes at a *coarse-grained* level, similar restrictions for the asynchronous coordination of sub-process instances hold as for imperative approaches. Hence, we presume a *fine-grained modeling style* in the following.

			Case Handling	Artifact-centric Modelling	Product-based Support	Data-driven Coordination	Proclets	Object-centric Processes
data integration	P1	object types	O	X	O	O	-	O
	P2	object relations	-	X	-	X	-	O
	P3	cardinalities	O	X	-	O	O	X
process granularity	P4	object behavior	X	X	X	X	X	X
	P5	object interactions	-	O	-	O	O	O
process modeling/ execution	P6	mandatory information	X	X	X	-	-	-
	P7	control-flow within forms	-	-	-	-	-	-
	P8	optional activities	O	-	-	-	-	O
	P9	flexible process execution	X	-	-	-	-	-
	P10	re-execution of activities	X	-	-	-	-	-
	P11	user decisions	-	-	-	-	-	O
activities	P12	form-based activities	X	-	-	-	-	-
	P13	black-box activities	X	X	X	X	X	X
	P14	varying activity granularity	-	-	-	-	-	-
	P15	batch execution	-	-	-	-	-	-
user integration	P16	instance-specificity	-	-	-	-	-	-
	P17	relationships	-	-	-	-	-	-
	P18	compliance	X	-	-	-	-	-
	P19	process dependency	-	-	-	-	-	-
	P20	differentiation	-	-	-	-	-	-

Figure 4.8: Evaluation of existing approaches (according to [KWR10a])

Object interactions can then be defined as sub-cases. However, it is not possible to execute them in the loosely coupled manner required (i.e., asynchronously to the higher-level case). It is further not possible to adequately handle the one-to-many relationships that exist between different cases; e.g., to aggregate sets of lower-level cases. Thus, object interactions are not supported as required in the context of object-aware processes (i.e., Prop. 5 is not met).

Data. CH solely provides *atomic data elements*; data integration based on object types and their inter-dependencies is not considered. However, using a fine-grained modeling style (i.e.,

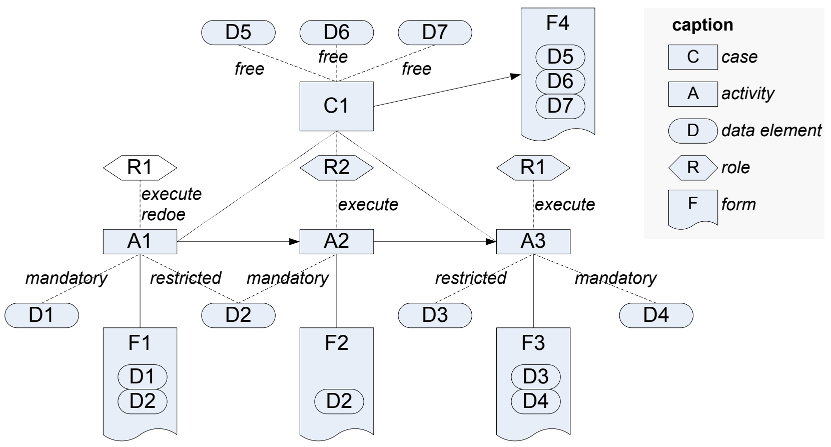


Figure 4.9: Case Handling Modeling Approach [vWG05]

to treat a case in tight accordance with an object), the case as a whole is considered as object. Thus, object types are partially supported (i.e., Prop. 1 is partially met). Even if interdependencies between different cases can be defined as sub-cases, corresponding relationships (at data level) are not explicitly defined. Thus, it is not possible to consider transitive and transverse relationships (i.e., Prop. 2 is not met). Further, Case Handling supports multiple-instantiation patterns through dynamic sub-plans (i.e., sub-process instances) [Ath02]. This allows creating a dynamically specified number of sub-process instances. Consequently, using multiple-instantiation patterns as extension, *cardinalities* between higher- and lower-level process instances can be taken into account. However, it cannot be ensured that the correct number of sub-process instances is actually created at run-time; i.e., to ensure that their quantity lies between the minimum and maximum cardinality (i.e., Prop. 3 is not fully met).

Processes. Opposed to activity-centric approaches, CH enables a *tighter integration* of processes, activities and data [MWR08, WMR10]. Thereby, CH differentiates between free, restricted, and mandatory data elements. For *mandatory data elements*, a value is required to complete the activity these data elements belong to. This way, mandatory information can be defined (i.e., Prop. 6 is met). However, since dependencies between input fields cannot be expressed, no support for controlling the *control-flow within a user form* exists (i.e., P7 is not supported). Opposed to imperative and declarative approaches, the activation of an activity does not depend on the completion of preceding activities, but rather on data changes. In the context of CH, an activity is considered as completed if its mandatory data elements all have an assigned value; i.e., *mandatory information* is provided. Thus, activities can be automatically skipped at run-time if their data elements are provided by other activities. This enables a *flexible process execution* (i.e., Prop. 9 is met). In this context, one must consider that *user commitments* can not be defined. As a consequence, it is not possible to intervene process execution in order to revise previously filled information (i.e., Prop. 11 is not met). However, besides defining who shall work on an activity, CH allows defining who may *redo* an activity or (manually) *skip* it. Despite the introduction of the redo-role, *re-executing activities* arbitrarily often is not possible (i.e., Prop. 10 is not fully met).

Example 4.6 (Re-executing activities in CH):
As illustrated in Fig. 4.10a, role `R1` may execute or redo activity `A1`. If all mandatory data elements of a particular

activity are available, subsequent activities become enabled immediately. Regarding this example (cf. Fig. 4.10b), as long as `A2` is not completed (i.e., a value for data element `D2` is not set), `R1` may redo activity `A1`. However, after completing subsequent activity `A2`, a redo only becomes possible if the user is authorized to redo `A2` as well (cf. Fig. 4.10c). Otherwise, `A1` cannot be redone any longer.

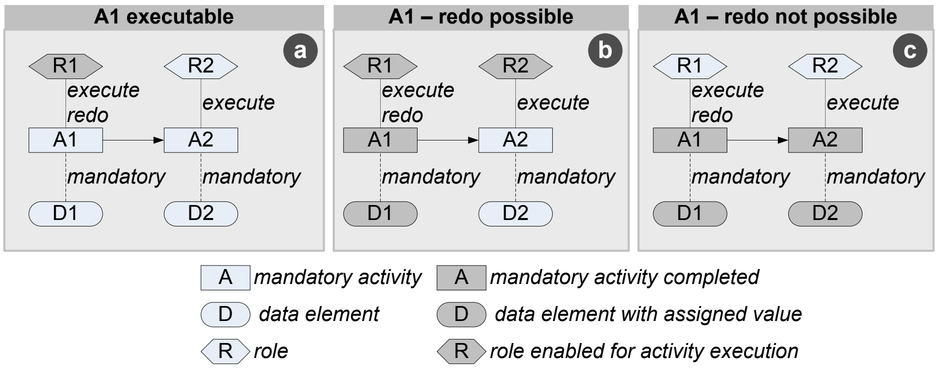


Figure 4.10: Re-execution of activities in Case Handling

Consider *optional activities*. First, all users involved in a case may read its data elements. Based on a query mechanism, users may access both active and completed cases. This allows accessing data at any point in time. Second, *free data elements* may be used to write data, independent from the normal flow of control. For this purpose, free data elements are assigned to the case as a whole rather than to specific activities. However, this way the same optional activity is offered to all users; i.e., it is not possible to offer different optional activities to different users depending on the current progress of the case instance (i.e., Prop. 8 is not fully met).

Activities. CH supports both *black-box* and *form-based activities* (i.e., Props. 12 and 13 is met). However, for both kinds of activities the *granularity is fixed* at build-time and a concrete activity is always executed in the context of a particular case instance. Consequently, users cannot access data elements of other relating cases. Thus, there is no variable granularity of activities supporting preferred work practices; e.g., context-sensitive activities cannot be realized (i.e., Prop. 14 is not met). Finally, since each case instance is executed in isolation, batch execution of activities is not supported (i.e., Prop. 15 is not met).

Users. Since the data elements processed during the execution of an activity are known, fine-grained process authorization at the level of single data elements becomes possible. Hence, it can be ensured that each user owning the execution role of an activity owns respective data permissions as well (i.e., Prop. 18 is met). Note that any user owning the execution role of an activity must execute it mandatorily; i.e., no differentiation between authorization and user assignment is made (i.e., Prop. 20 is not supported). Regarding data authorization, it is further not possible to define different access rights for a particular user depending on the progress of the case (i.e., Prop. 19 is not met). Finally, instance-specific authorization, considering the relationships between users and application data, is not provided (i.e., Props. 16 and 17 are not met).

For a detailed comparison of CH with the imperative approach see [MWR08, WMR10]. Finally, an approach presenting more flexible processes using adaptive workflows and CH is presented in [GRv08].

4.3.2 Artifact-centric Modeling

We now consider artifact-centric process modeling [NC03, LBW07, GS07, Hul08] and evaluate to what degree it covers the properties of object-aware processes. The basic concepts of this approach are illustrated in Fig. 4.11. *Business artifacts* are similar to object types; i.e., a business artifact comprises atomic as well as structured *attributes*, *related business artifacts*, and a *lifecycle* [Hul08, BHS09]. The latter is defined using a finite-state machine capturing the main processing stages and the transitions between them. Transitions may be associated with conditions defined in terms of attribute values or relationships to other business artifacts. In turn, services are executed to evolve business artifacts through their entire lifecycle. For this purpose, associations (i.e., ECA-rules) specify how services are linked with artifacts. Note that artifacts (including their informational structure and lifecycles), services, and ECA-rules only constitute a logical representation of business processes and business data. A formal specification of the semantics of artifact-based process models is presented in [HDD$^+$11, DHV11, DHV13]. In particular, an incremental as well as a fixpoint semantics is provided for business artifacts. However, note that this semantics focuses on the life cycle of a particular artifact, but does neither consider a data-driven execution paradigm nor interactions with other artifacts during run-time.

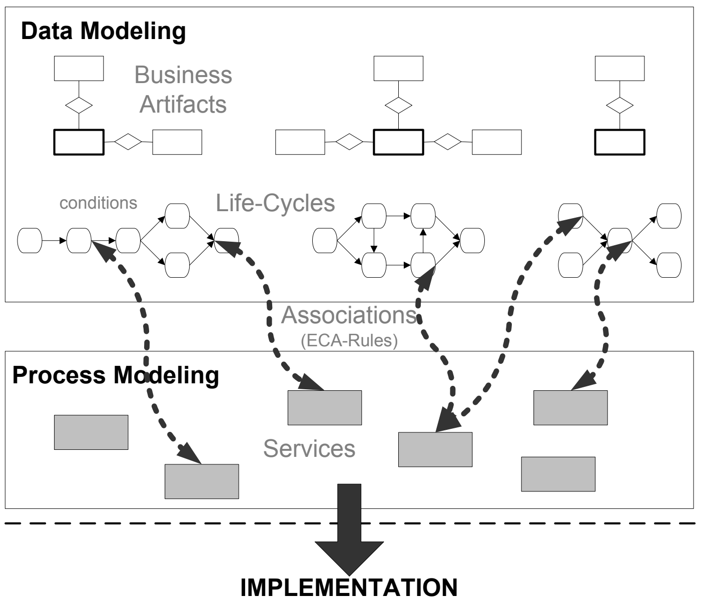


Figure 4.11: Artifact-centric modeling [Hul08]

Data. Like objects, artifacts consist of different attributes and refer to related artifacts. Thus, there exists no comprehensive data structure. Instead, attributes and relations may be redundantly defined within different artifacts (i.e., artifact definitions are treated independent from each other). The emerging data structure is redundantly distributed among several data models. This makes it hard to comprehend and difficult to maintain. Altogether, data modeling based on object types, relations and cardinalities is supported (i.e., Props. 1, 2, and 3 are met), even through it has no impact on the process structure.

Process Granularity. In addition to the definition of the information structure, for each artifact, a life-cycle model exists. The latter is used to describe artifact behavior (i.e., Prop. 4

is supported). During process execution, artifacts are processed. Thereby, they change to different life-cycle stages. However, granularity issues in respect to activities (and processes respectively) are not considered; i.e., it does not become transparent which artifacts are actually processed when a particular activity is executed; i.e., each service processes one or more artifacts (of same or of different type). Further, ECA-rules are used for coordinating artifact life cycles based on quantifiers (e.g., $\forall$, $\exists$). Consequently, there is no clear separation between behavior and interactions. Finally, aggregations are not taken into account and neither transitive nor transverse relationships between business artifacts are considered (i.e., Prop. 5 is not fully met).

Process Modeling and Execution. Services are defined in terms of black-box activities. Thus, it is neither possible to determine the order in which object attributes shall be written nor to define what valid attribute settings shall be (i.e., Prop. 7 is not supported). However, for each transition of an artifact life-cycle, a (data-) condition (describing mandatory information) may be specified (i.e., Prop. 6 is met). Based on this condition, it becomes possible to synchronize data state and process state. However, it is a tedious task to ensure that these conditions are consistent with the ECA-rules specified for service invocations. The latter enable the activation of services based on data conditions. Hence, activities may be re-executed as long as required data is not available (i.e., Prop. 10 is met).
User commitments are not explicitly considered in this context (i.e., Prop. 11 is not supported). Since ECA-rules constitute pre-conditions rather than post-conditions, it is not possible to dynamically skip services (i.e., Prop. 9 is not supported). Although optional activities may be realized using ECA-rules, the business artifacts framework does not aim at an integrated access to application data. Thus, for users it is not possible to distinguish between optional and mandatory activities in their worklists (i.e., Prop. 8 is not supported).

Activities. All services are defined in terms of black-box activities (i.e., Prop. 13 is supported). Form-based activities, which may be automatically generated based on the data conditions, are not considered (i.e., Prop. 12 is not met). Since each service requires its own implementation, the granularity of activities is fixed at build-time. Consequently, it is neither possible to provide a dynamically varying granularity of activities nor batch execution (i.e., Props. 14 and 15 are not met).

Users. Regarding user integration, the approach relies on similar concepts as applied in traditional PrMS (e.g., role-based access control [FK92]). A more advanced user integration, as required for object-aware process support, is not provided (i.e., Props. 16, 17, 18, 19, and 20 are not met).

For more details see [NC03, LBW07, GS07, Hul08]. Further, [BGH^{+}07] and [GBS07] discusses a formal analysis of artifact-centric business process models. For a comparison of the artifact-based approach with the imperative one see [KLW08]. Finally, modeling artifact-centric processes in a declarative way is introduced in [VHH^{+}11].

4.3.3 Product-based Workflow Support

The core of product-based workflows [vdA97, RL03, Van09, VRv11] is a *Product Data Model* described in terms of a tree-like structure [RL03]. Such a structure comprises *atomic data elements* as well as *operations* (cf. Fig. 4.12); i.e., processes are designed in respect to a given product structure. The latter may have several input data elements and exactly one

output data element. An operation is executable if (specific) values are available for all input data elements. Finally, the product is fully processed as soon as a value for the top element of the product data model (i.e., the root element) becomes available.
In particular, the basic idea is to model a sequence of activities based on a bill-of-material and design criteria like costs, quality, and time. Process models can then be manually derived [RL03] or automatically generated [Van09] based on the product data model. In addition, it is possible to directly execute the product data model [Van09, VRv11].

Data. Product-based modeling only considers atomic data elements; data integration in terms of object types and their inter-relations are not considered. However, it is possible to choose a *fine-grained modeling style* as well; i.e., one particular product data model may be related to one object. Thus, object types (i.e., Prop. 1) are partially supported. Since sub-processes are not considered, object relations and cardinalities cannot be taken into account (i.e., Props. 2 and 3 are not met).

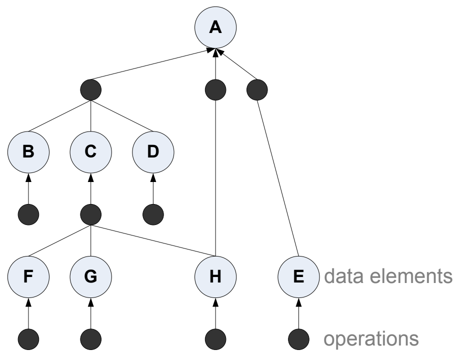


Figure 4.12: Product data model [VRv11]

Process Granularity. Regarding product-based modeling, dependencies between atomic data elements (that can be considered as attributes) are established. Nevertheless, processes (as well as activities) may be defined at an arbitrary level of granularity; i.e., while some activities may process data elements belonging to one object instance, executing another one may involve data elements belonging to several object instances (of same or different type). In this context, it does not become transparent how the various data elements belong together (i.e., to the same or to a different object instance). Taking the before mentioned fine-grained modeling approach, however, it is possible to ensure that all data element involved in the process model belong to one and the same object instance. This way, object behavior is provided (i.e., Prop. 4 is met). Since each process instance is executed in isolation, however, object interactions are not supported (i.e., Prop. 5 is not met).

Process Modeling and Execution. Input data elements are explicitly specified. Hence, mandatorily required information becomes transparent (i.e., Prop. 6 is met). However, each process model solely contains data elements mandatorily required for process execution. Additional data is not taken into account. Consequently, optional activities are not considered; i.e., it is not possible to write certain data elements even if they are not required at the moment (i.e., Prop. 8 is not supported). What is done during activity execution, however, is still out of the control of the PrMS; i.e., activities constitute black-boxes. Hence, it is not possible to capture the internal flow logic of a user form (i.e., Prop. 7 is not met). The run-time interpretation of the product data model itself enables data-driven process execution. Since the activation of an activity depends on pre-conditions, however, an activity cannot be automatically skipped if required output data element is already available (i.e., Prop. 9 is not supported). Finally, activities cannot be re-executed and user decisions are not explicitly considered (i.e., Props. 10 and 11 are not met).

Activities. Only black-box activities are considered (i.e., Prop. 13 is supported). As extension, however, user forms may be automatically generated based on the respective input and output data elements. Since only one particular output data element is allowed, however, each form may only comprise one input field. Concerning object-aware processes, providing required attribute values is not as strict as for product-based processes. Consequently, form-based activities are not supported by the latter (i.e., Prop. 12 is not met). Further, each activity requires its own implementation and each process instance is executed in isolation. Finally, the granularity of activities is fixed at build-time; i.e., it is not possible to provide a dynamically varying granularity of activities as well as batch execution (i.e., Props. 14 and 15 are not met).

User Integration. Concerning user integration, product-based processes apply similar concepts as traditional PrMS (i.e., role-based access control). More advanced concepts, required in the context of object-aware processes are not considered (i.e., Props. 16, 17, 18, 19, and 20 are not met).

For more detailed information see [vdA97, RL03, Van09, VRv11]. [Rei02] illustrates a use case applying the product-based approach within the financial services. Further, a first approach towards integrating this approach with case handling is presented in [vB01, VRvdA08]. Finally, [CV11] uses this approach for monitoring collaborative business processes and [RVV10] addresses challenges for dealing with schema evolution in this context.

4.3.4 Data-driven Process Coordination

Corepro [MRH07, MRH08a, MRH^{+}08b, Mö9] and its data-driven coordination framework enable the coordination of multiple processes based on objects and object relations. Single objects are defined in terms of states and (internal) transitions between them. The latter can be associated with complex actions (i.e., processes) that must be completed before the subsequent state may be entered. According to the relations between objects, external transitions connect the states of different objects to coordinate their processing (cf. Fig. 4.13).

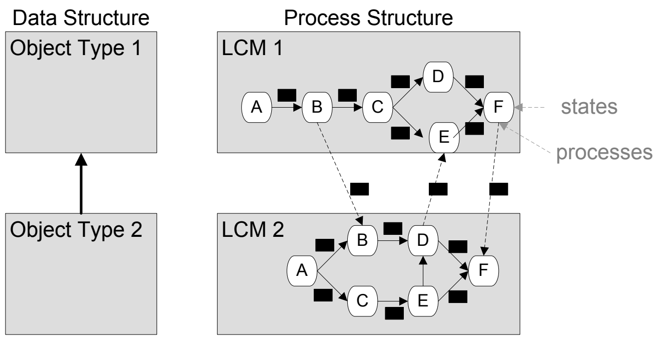


Figure 4.13: COREPRO modeling approach

Data. Object types are explicitly defined. However, the respective definition does not cover object attributes (i.e., Prop. 1 is not fully met). Further, it is possible to define object relations. In this context, several object instances may be handled at run-time. However, their concrete number must be fixed at build-time. Cardinality constraints are not taken into account (i.e., Prop. 3 is not fully met).

Process Granularity. For each object type, a process model describing object behavior is defined in terms of a state chart (i.e., Prop. 4 is supported). At run-time, corresponding process instances are executed concurrently. Based on the aforementioned external transitions, the execution may be synchronized where needed. Although the approach considers multiple object instantiation, one-to-many relationships between object types cannot be properly handled. In particular, it is not possible to aggregate sets of lower-level object instances. Moreover, process coordination is solely aligned with object relations. Transitive or transverse relationships cannot be specified. Hence, object interactions are not supported as required in the context of object-aware processes (i.e., Prop. 5 is not fully met). In summary, it is possible to coordinate individual (i.e., object-specific) processes based on the underlying data structure while the granularity of processes (and activities) can be freely chosen.

Process Modeling and Execution. Since object attributes are not considered, mandatorily required information cannot be defined (i.e., Prop. 6 is not met). In addition, access to data is only possible when executing activities specified within the process model; i.e., optional activities are not supported (i.e., Prop. 8 is not met). Activities are treated as black-boxes; i.e., it is out of control what is done during their execution. Hence, it is not possible to define the internal flow logic of a form (i.e., Prop. 7 is not met). Processes themselves are still activity-driven, only process synchronization follows a data-driven approach. In particular, the activation of a subsequent state depends on the completion of a process associated with the corresponding state transition. Consequently, a flexible data-driven execution, allowing for the re-execution of activities and the explicit consideration of user decisions, is not supported (i.e., Props. 9, 10, and 11 are not met).

Activities. Process execution is solely based on black-box activities (i.e., Prop. 13 is met). Since object attributes are not defined, form-based activities cannot be automatically generated at run-time (i.e., 12 is not met). Neither a variable granularity of activities nor batch execution are supported (i.e., Props. 14 and 15 are not met).

User Integration. The integration of users is not addressed. Since processes are considered as black-boxes, standard concepts for user integration may be applied (e.g., role-based access control). Advanced concepts, as required in the context of object-aware processes, are not considered (i.e., Props. 16, 17, 18, 19, and 20 are not met).

4.3.5 Proclets

Proclets [vBEW00, vBEW01, vMR09] constitute small processes that interact with each other through the *exchange of messages*. The latter is accomplished through ports and channels (cf. Fig. 4.14). Channels may be customized to support different kinds of interactions (e.g., push/pull, synchronous/asynchronous). A naming service can be used to find other Proclets. Each message sent or received is stored in the knowledge base of the respective Proclet.

Data Integration. Object types are not explicitly managed by this framework. However, the definition of a proclet may be aligned with a specific object type (i.e., to use a *fine-grained modeling style*). Object attributes are not considered (i.e., Prop. 1 is not fully supported). Since each proclet may communicate with any other proclet, coordination is not based on object relations (i.e., Prop. 2 is not met). Since for each proclet type a dynamic number of instances can be handled at run-time, a complex process structure emerges and evolves during run-time. Finally, cardinality constraints can be handled (i.e., Prop. 3 is partially met).

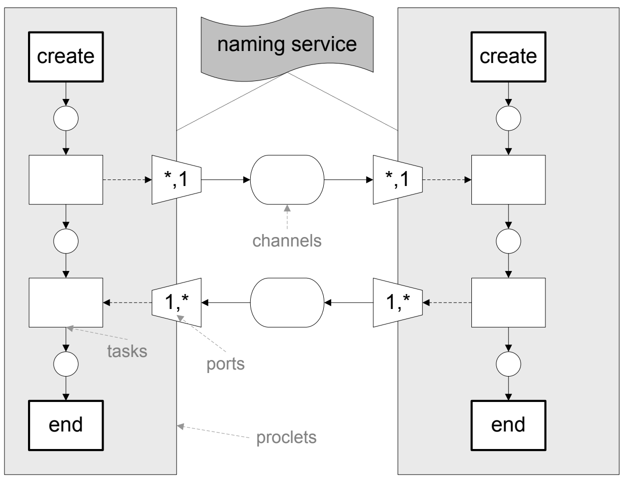


Figure 4.14: Proclet framework [vBEW00]

Process Granularity. The granularity of a proclet is not explicitly defined. However, using a *fine-grained modeling style* (i.e., each proclet is defined in close relation to an object type), a certain kind of object behavior can be expressed (i.e., Prop. 4 is met). At run-time, for each proclet type a dynamic number of instances is handled. This way, a complex process structure emerges, for which the individual proclet instances may be executed asynchronously to each other and synchronized where needed. It is possible to send a message to multiple proclets or to restrict the number of recipients based on cardinality constraints. In addition, activity pre- and post-conditions may be used to define creation and execution dependencies as well as aggregations. However, since object relations are not taken into account, transitive or transverse relationships between proclet instances cannot be handled (i.e., Prop. 5 is not fully met).

Proclets are defined using Petri nets [vBEW00]. Thus, black-box activities are supported (i.e., Prop. 13 is met). In turn, all other properties belonging to the business perspectives *processes*, *activities*, and *users* are not supported (i.e., Prop. 6 - 12 and Prop. 14 - 20 are not met).

For more detailed information see [vBEW00, vBEW01, vMR09]. Finally, a use case from the healthcare domain is introduced in [MRv^{+}10].

4.3.6 Object-centric Processes

Object-centric business process modeling [RDtl07, RDtl09a, Red09, RDtl09b] allows coordinating object-specific processes, which are defined in terms of finite state machines. As illustrated in Fig. 4.15, the latter are coordinated using different signals. To each state, several atomic tasks may be assigned.

Data Integration. Object types may be explicitly defined. However, similar to the data-driven process coordination approach, object attributes are not explicitly considered (i.e., Prop. 1 is not fully met). As opposed to the data-driven process coordination approach, object relations are not explicitly defined. Instead, coordination dependencies are directly defined using various signal types. This way, process coordination may not necessarily coincide with object relations (i.e., Prop. 2 is not fully met). In this context, cardinality constraints may be considered as well (i.e., Prop. 3 is met).

Process Granularity. For each object-type, a finite state-machine is defined to describe its behavior (i.e., Prop. 4 is supported). At run-time, for each object type a varying number of object instances (with corresponding state machines) may then be executed concurrently to each other. However, it is not possible to specify asynchronous object interactions as required in the context of object-aware processes. In particular, the spawn signal (cf. Fig. 4.15) should not be restricted to specific points during process execution. Instead, it should be possible to assign them to a number of states. Moreover, aggregation conditions (cf. the finished signal in Fig. 4.15) should be definable using more complex constraints rather than solely declaring minimal and maximal counts. Consequently, object interactions cannot be handled as flexible as required (i.e., Prop. 5 is not fully met).

Process Modeling and Execution. Tasks are assigned to particular states to activate them. Since object attributes are not considered, mandatory information does not become transparent (i.e., Prop. 6 is not met). For this reason, a flexible (data-driven) process execution is not possible (i.e., Prop. 9 is not met). However, [RDtI09a] introduces an approach enabling more flexible task execution. This way, unplanned tasks may be invoked allowing for some kind of optional activities. When using these tasks, however, integrated access to involved business data is not possible (i.e., Prop. 8 is not fully supported). Since tasks are considered as black-box, the internal flow logic of a user form cannot be specified (i.e., Prop. 7 is not met). Finally, neither re-execution of activities nor the integration of user decisions are considered (i.e., Props. 10 and 11 are not met).

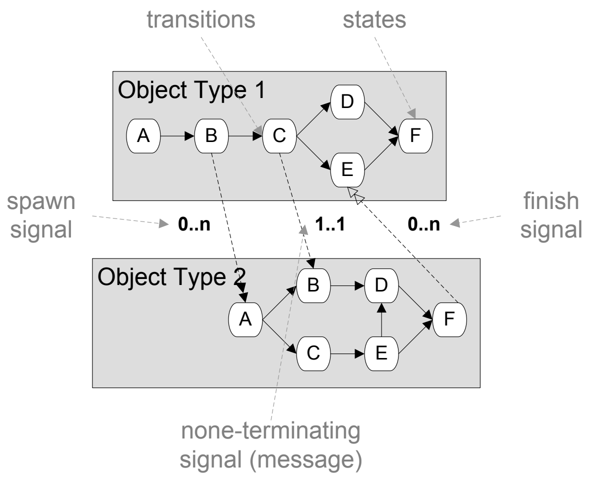


Figure 4.15: Object-centric modeling approach [RDtI09a]

Regarding activities and the integration of users, the same conclusions can be as for imperative process support paradigms (i.e., Props. 12 - 20 are not met).

4.3.7 Other Approaches

In the previous sections, we presented approaches dealing with traditional process management. Another interesting concept from this area is discussed in [SOSS05]. It deals with the *grouping and ungrouping of activities* in worklists in order to enable batch execution (cf. Prop. 15).

Object Behavior Diagrams [KS91, PS98] provide a graphical notation for describing the behavior of objects in object-oriented software systems. They comprise states and activities triggering state transitions. Since it is possible to transform object behavior diagrams into a corresponding Petri Net representation based on synthesis techniques [CKLY98, ER89], an operational semantics for executing them is provided. It is further possible to synchronize the execution of behavior diagrams corresponding to different objects [M09]. However, no formal operational semantics is provided; further the synchronization support is not sufficient to fully met the properties of object-aware processes.

Another approach dealing with object behavior is provided by UML [BRJ98]. In particular, *UML Activity Diagrams* provide an activity-centered approach for modeling object behavior comprising control flow logic, events, and user decisions. In turn, with *UML State Charts*, object behavior can be defined in terms of states and state transitions. However, both diagrams do not explicitly consider object attribute values, and hence a data-driven execution is not addressed. Furthermore, UML does not provide a well-defined operational semantics for executing respective diagrams at run-time (although there exist proposals for this [PDG$^+$11, Ges10, Esh02]). Finally, UML lacks mechanisms for synchronizing the execution of different models [Red09].

Data-driven programs [DH12] address the problem of exchanging data between different information systems using interface protocols. However, data semantics is not considered. Further, the communication between information systems becomes complex. For this reason, [DH12] proposes to design software applications operating with defined master data rather than individual 1:1 interfaces. Accordingly, software applications should be coordinated based on a common basis of data, which relies on concepts like grouping, cardinalities, polymorphism, and a "case-of" mechanism.

In the context of *model-driven software architectures* [Fra10], application code is automatically generated based on a collection of different models. Like PHILharmonicFlows, this approach aims at increasing development times and managing the complexity of large software systems. Opposed to PHILharmonicFlows, however, the models are used to generate corresponding application code rather than to interpret this code directly. As a consequence, a lot of effort is required to keep models and application code consistent.

Finally, *scientific workflows* [TDGS07] address a tighter integration of processes and data as well. In this context, different data sources are used to perform extensive calculations and evaluations requiring distributed resources. However, they are mainly based on an activity-centric execution paradigm.

4.4 Approaches for User Integration

Existing approaches are provided by process management and application systems. Regarding the latter, both consistency issues and instance-specific authorization have been considered. However, user assignment based on data permissions and considering the relationships between users add data are not taken into account by any approach. To our best knowledge, only approaches from the area of application systems consider relations between users and object instances. Regarding process management, a set of approaches addressing specific strategies for user assignment are available. Altogether, a comprehensive solution for generic access control, as required for object-aware process management, is still missing.

4.4.1 Enabling Instance-specific Authorization

Instance-specific authorization (i.e., Prop. 16) means that the selection of potential actors does not only depend on the activities performed, but also on the object instances processed by the respective activity. For this purpose, [ST97] describes an approach that groups permissions for accessing data and functions. Whether a user may perform a particular activity depends on the agreement of another user at run-time. This enables a manual approval of the object instances relevant in the given context. However, it is not possible to access data outside the scope of a specific activity; i.e., optional activities are not considered. Similarly, current PrMS focus on actor assignments for controlling activity executions. By contrast, permissions for accessing data and functions are mostly managed within the invoked application systems. [WSML02] describes the concept of "instance-based user group". Actors get access to all object instances associated with at least one process instance they have been involved in. The opposite direction (i.e., user assignments depending on permissions for data access) is not considered. [RzM98] allows managing specific properties for each data element relevant for the process. In addition to actor expressions, for each activity, relevant properties of the used data elements are defined. Obviously, this is accomplished in a redundant manner and therefore data inconsistencies might arise.

4.4.2 Ensuring Consistency between Data and Process Authorization

To ensure consistency between data and process authorization, each user who shall execute a process-relevant activity must own corresponding data permissions as well (cf. Prop. 18). In this context, Case Handling [vWG05] distinguishes between mandatory and optional data elements of an activity; however, no differentiation between mandatory and optional activities is made. Similar to [WSML02], users get access to all data elements of the process instances they are involved in (i.e., the permission to read / write data is assigned implicitly). [ST97] and [Bot02] define permissions for accessing data and functions in the context of a specific task.
The opposite direction must be taken into account as well. More precisely, the assignment of users to tasks may depend on data permissions (cf. Prop. 19). To our best knowledge, this is not considered by any approach. Regarding [ST97] and [Bot02], for example, all permissions are defined at the level of object types. Thus, it is not possible to assign different permissions for object instances of the same type.

4.4.3 Considering Relations between Users and Object Instances

In some approaches [LS97, KKC02, Tho97, HW04, BBU99], it is possible to restrict permissions to a selected set of object instances. However, only in few cases [KKC02, HW04, BBU99] these restrictions can be defined depending on the relationships between users and object instances. In particular, it is not possible to consider relationships already defined in the data structure. Instead, they must be defined redundantly based on the permissions.

4.4.4 Differentiating User Assignment and Authorization

Except few approaches (e.g., Case Handling), PrMS do not support optional activities as described in Sect. 3.1. Hence, it is not possible to differentiate between tasks users must execute and tasks they may execute. In [RtE05], various possibilities for assigning and activating activities are described. [BFA99] takes the hierarchy of roles into account and [WBK03] allows defining different priorities for assigning users to activities. However, for all approaches, always at least one user must execute an activity. Opposed to this, [vWG05] focuses on data access rather than on assigning users to activities. Finally, for each activity it is possible to differentiate between optional and mandatory data elements.

5 Requirements

Chapt. 4 has revealed characteristic problems that occur when using existing process management technology for implementing object-aware processes and their properties. Tab. 5.1 illustrates major differences between object-aware process support on one hand and traditional (i.e., imperative and declarative) process support on the other:

	Object-Aware Process Support	**Traditional Process Support**
Data Integration	integrated data access	hidden information flows
Modeling	object behavior and interactions	arbitrary process granularity
Execution	data-, user-, and activity-driven	activity-driven / constraint-based
Activities	flexible granularity	fixed granularity

Table 5.1: Comparing object-aware with traditional process support

Basically, any PrMS targeting at the support of object-aware processes must satisfy a number of fundamental requirements. This thesis differentiates between end-user requirements, process support requirements, and system requirements. Note that we summarized the requirements in a compact and unstructured way considering the fact that we already systematically presented the properties of object-aware processes, to be supported by the generic solution framework we target at, in Chapt. 3.

As emphasized, we aim at effective business-IT-alignment, rapid application development and improved maintenance. General requirements like scalability, traceability, quality, and reusability apply to all applications and PrMS, and are therefore not discussed in detail. Instead, we focus on requirements for the modeling and execution of object-aware processes. In this context, we must ensure correct and robust process execution during run-time. Regarding real-world scenarios, however, it is not always possible to completely exclude deadlock situations. For example, reviewers might not be able to agree on whether or not to hire a particular applicant; or it might be not possible to recruit an appropriate candidate for a job. Consequently, not only rules for structural correctness are required, but also extensive mechanisms for detecting incorrect situations at run-time. For this purpose, it is essential to keep track of relevant run-time information (e.g., the current process state).

5.1 End-User Requirements

Traditional PrMS are unable to provide data- and function-oriented views as known from many application systems (i.e., accessing business information is only possible when executing a process activity and its related application function). In turn, regarding the end-user view on object-aware processes, it will be crucial that all business perspectives can be accessed using one and the same system; i.e., an integrated access to processes and data must be provided. In particular, proper business-IT-alignment necessitates an integrated access to data, functions, and processes. Finally, available data and running processes should be transparent to authorized users.

5.1.1 Integrated Access to Processes, Data and Functions

To provide effective support for object-aware processes, both a data- and a process-oriented view need to be provided for end-users. In particular, hidden information flows, as can be found in traditional PrMS, are no longer acceptable. Applying a *data-oriented view*, users must be able to access and manage data at any point in time. In turn, using a *process-oriented view*, it should be possible to assign upcoming tasks to the right users at the right point in time. However, it is not sufficient to only provide strictly separated views on processes and data. In particular, a process instance is tightly coupled with its corresponding object instance; i.e., attribute values reflect the progress of its related process instance. In turn, data authorization (i.e., read and write permissions on object attributes) depends on the progress of process execution. As illustrated in Fig. 5.1, these mutual dependencies between object and process instances should be made transparent to end-users. On one hand, this means that during process execution access to relevant context information is needed. For this purpose, in addition to upcoming mandatory activities, the process-oriented view should display the objects referred by these activities. On the other hand, business data should include information about corresponding processes. Thus, within a data-oriented view, the progress of the corresponding process instances must be displayed. Activities may then be invoked starting from both, the data- and the process-oriented view.

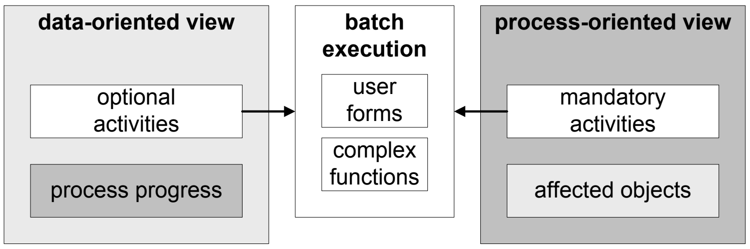


Figure 5.1: End-user view - abstract model

5.1.2 Monitoring

When executing object-aware processes (i.e., object behavior), upcoming mandatory activities should be immediately assigned to responsible users. For this purpose, worklists (i.e., adequate process-oriented views) must be provided. Generally, for an object type, hundred or thousands of object instances might exist. Such huge data sets are usually handled within existing application systems. For example, consider a human resource management system, which concurrently processes thousands of job applications; i.e., system performance is crucial. Furthermore, object instances may be created or deleted at arbitrary points in time; i.e., the corresponding data structure dynamically emerges and evolves depending on the type and number of created object instances as well as on their relations. In addition, for each object instance (e.g., job application), a corresponding process instance must be handled and its manifold inter-dependencies with other process instances be controlled. Considering this, we obtain a complex and potentially large process structure referring to a multitude of object instances and reflecting their inter-dependencies.
Consequently, in addition to the processing state of individual process instances (i.e., object instances), it must be possible to track the various object interactions (i.e., the overarching process logic). For this, proper monitoring concepts are required. In particular, the processing state of the whole process structure must be displayed at each point in time during process execution. In this context, it is crucial to adequately handle one-to-many relationships and to consider the asynchronous execution of the individual process instances. Finally, inconsistencies emerging during the execution of a complex process structure must be detected and pproperly handled.

5.2 Process Support Requirements

Obviously, the missing link between application data and business processes prohibits an integrated access. Further, it disallows defining processes in close relation to data. In order to realize the properties of object-aware processes, a new process modeling paradigm and methodology is required. In particular, process execution should no longer be solely activity-driven (i.e., the execution of activities depends on the completion of other activities). Instead, a data-driven execution paradigm, combined with activity-centric aspects and user decisions, is required. Since activities may have flexible granularity, individual activities do not longer coincide to particular process steps as in existing process support paradigms. Note that this is a fundamental difference.

5.2.1 Process Modeling Methodology

Object-aware processes are usually structured according to business objects involved. To align data and process modeling in this context, these must not be treated independent from each other; i.e., processes need to be defined in close relation to the underlying data model and data must no longer be treated as second-class citizens within the PrMS [vWG05]. Instead, process modeling must be based on a clear methodology differentiating between object behavior and object interactions (cf. Props. 4 and 5). Further, it is no longer recommendable to define processes at an arbitrary level of granularity as in existing PrMS (cf. Fig. 5.2).

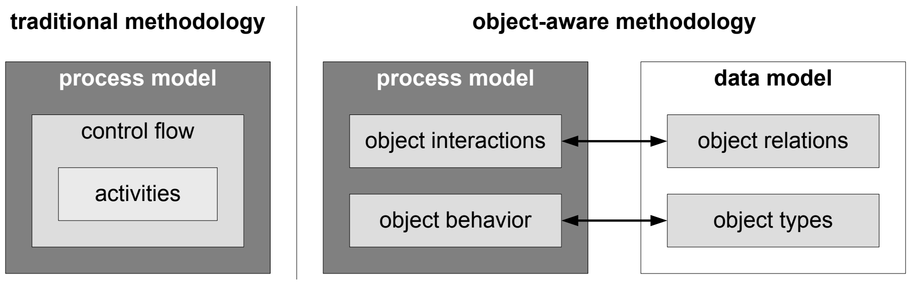


Figure 5.2: Object-aware process modeling methodology

5.2.2 Process Modeling Paradigm

As discussed, users should be able to choose the work practice they prefer. Therefore, a flexible execution of activities is required; i.e., activities must not follow a pre-defined execution order. For this reason, a strict activity-based process modeling and process execution paradigm, as supported in traditional PrMS, is too rigid.

First of all, consider the execution of an activity in the context of a particular process instance reflecting object behavior. Using optional activities (cf. Prop. 8), it should become possible to execute activities before and after their (mandatory) activation during process execution. We denote this as *horizontal flexibility* (cf. Fig. 5.3). In turn, an activity may also process attribute values of several object instances; i.e., an activity execution may refer to multiple object instances. We denote this as *vertical flexibility* (cf. Fig. 5.3). Finally, *batch execution* of activities (cf. Prop. 15) might be required; i.e., attributes of several instances of the same type may be processed in one go. Altogether, the granularity of an activity should not be fixed at build-time, but be dynamically selectable depending on the needs of the respective user. Therefore, it is no longer appropriate that activities coincide to individual process steps as in existing applications. In addition, process instances can no longer be treated independent from each other.

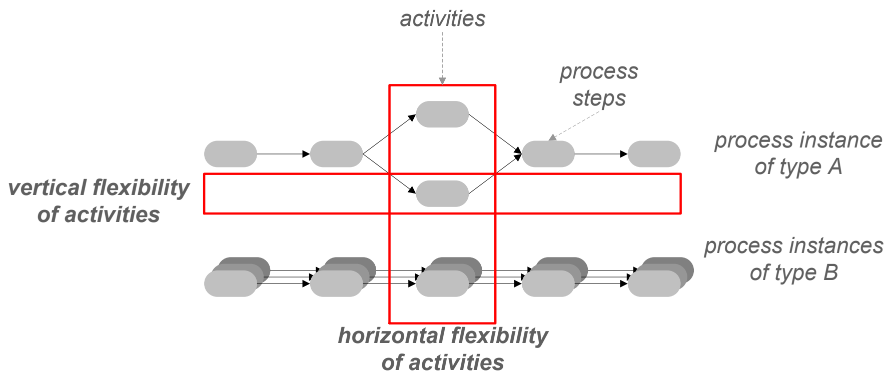

activities
process steps
process instance of type A
vertical flexibility of activities
process instances of type B
horizontal flexibility of activities

Figure 5.3: Horizontal and vertical flexibility of activities

5.3 System Requirements

To ensure rapid development and run-time flexibility (e.g., in respect to the activity granularity chosen), an object-aware process management should comprise generic techniques for automatically generating worklists, overview tables, and user forms at run-time. In particular, these components should enable an integrated access to data, functions, and processes. In addition, the structure of data and process models should correspond to each other; i.e., data and processes are two sides of the same coin. For this purpose, the traditional architecture of process-aware information systems, as introduced in Chapt. 1, is no longer applicable.

5.3.1 Generic End-User Components

Process definition must be accomplished at two levels of granularity; i.e., object behavior and object interactions. Opposed to this, however, activity execution should be of flexible granularity; i.e., an activity may comprise input fields corresponding to attributes of several object instances and process instances respectively (cf. Fig. 5.3). When initiating a `review`, for example, it should be possible to simultaneously change attribute values of the `application` and the `job offer`. Which input and data fields shall be displayed then, not only depends on the user performing this activity, but also on the state of the respective process instances. Note that this might require a multiplicity of different user forms. Manually implementing all these forms would be a cumbersome and error-prone task. To enable rapid development, instead, it should be possible to automatically and dynamically generate user forms based on a generic component at run-time. The latter should automatically determine required input fields (for writing object attributes) and data fields (for reading object attributes) at run-time. In addition, it should be possible to automatically generate data-oriented views (i.e., overview tables) as well as process-oriented views (i.e., worklists) during run-time.

5.3.2 New Architecture

To overcome fundamental limitations of traditional PrMS, a tight integration of data and process models becomes necessary (cf. Fig. 5.4a). Such a tight integration provides the foundation for realizing the discussed generic techniques for automatically generating user interfaces. In particular, permissions for accessing data depend on the progress of corresponding process instances. In turn, process execution is primarily data-driven; i.e., the activation of mandatory activities depends on already assigned attribute values. More precisely, to automatically generate overview tables, worklists, and user forms, both information about data and corresponding processes is required (cf. Fig. 5.4b). Finally, user integration is not only required for process support, but also in the context of data and functions (cf. Fig. 5.4b). In particular, a tight integration of data functions and processes is indispensible for enabling integrated access to them during run-time.

Finally, it is not appropriate to treat user roles and application data independent from each other as in existing PrMS. Instead, data structures (i.e., object types and their inter-relations) should embed organizational entities as well; i.e., roles should be explicitly defined in terms of object types. Otherwise, it will not be possible to utilize the relationships existing between users and object instances (cf. Prop. 17).

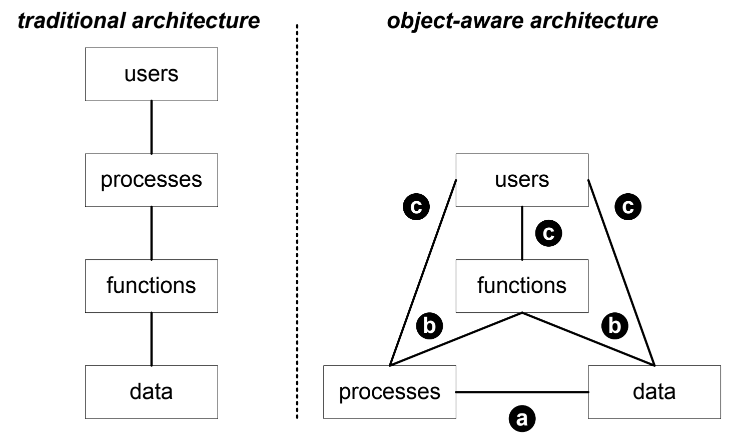


Figure 5.4: Traditional architecture and object-aware architecture (according to [KR11b])

Altogether, a tight integration of all business perspectives (cf. Chapt. 1) is required. Nevertheless, separating concerns still must be ensured to allow for a good system maintenance. In particular, data and processes must be defined in separate, but well-integrated models.

Part III

PHILharmonicFlows

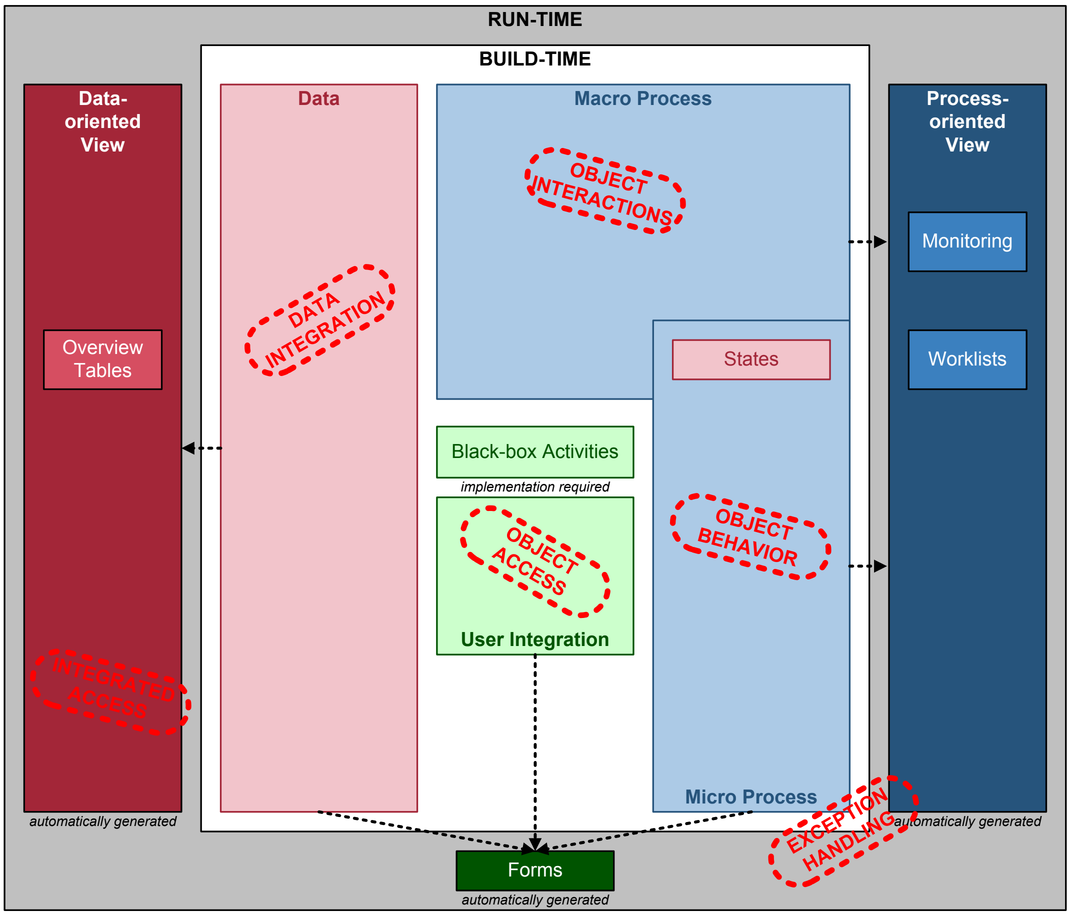


Figure 5.5: Main components of the PHILharmonicFlows framework

The overall research goal of this thesis is to provide a comprehensive framework for *object-aware process support* that meets all the properties we identified in Chap. 3. In part III of this thesis, we incrementally introduce PHILharmonicFlows[1] – a framework providing a *generic solution approach* for enabling an *integrated access* to business data, business processes, and business functions (cf. Fig. 5.5) [KR11b, KR11a]. Based on the various models provided by this framework and their generic implementation, an automatic and dynamic generation of data- as well as process-oriented views becomes possible at run-time (e.g., overview tables, worklists, and user forms). Regarding activities, PHILharmonicFlows differentiates between a *form-based* and *black-box* implementation. The latter allow for more complex business functions accomplishing computations or integrating legacy applications. Hence, for each black-box activity, a specific implementation is required. In turn, form-based activities (i.e., user forms), are automatically generated taking both user permissions and the progress of the respective process instance into account. Finally, PHILharmonicFlows provides advanced support for *user integration*, *process monitoring*, and *exception handling*.

Fig. 5.5 summarizes the main components of the PHILharmonicFlows framework. Basically, it comprises a *modeling as well as a run-time environment* enabling full lifecycle support for object-aware processes. As opposed to activity-centric process modeling approaches which explicitly require specifying activities and their execution constraints (e.g., precedence relations), PHILharmonicFlows allows defining processes in tight integration with data. As a fundamental prerequisite, a *data model* of the respective domain needs to be defined. In most existing process management systems (cf. Chap. 4), the data and process perspectives are integrated in one and the same artifact resulting in complex models that are difficult to comprehend and maintain. In turn, PHILharmonicFlows allows for defining data and processes in separate, but well integrated models (cf. Fig. 6.1). Thus, it retains the well established *principle of separating concerns* [Dij76].

The *modeling environment* of PHILharmonicFlows enforces a well-defined modeling methodology that governs the definition of processes at different levels of granularity. In this context, the framework differentiates between *micro* and *macro processes* capturing both *object behavior* and *object interactions*.

Process and data authorization is based on *user roles*. Data may be accessed optionally and at any point during run-time. In turn, process execution is based on permissions for creating and deleting object instances as well as for reading or writing their attributes. Furthermore, the access to object attributes must take the current progress of the process into account. In order to enable access control at such a fine-grained level, PHILharmonicFlows maintains a comprehensive *authorization table*. Finally, besides form-based activities, PHILharmonicFlows supports black-box activities (e.g., to invoke a web service or send an e-mail).

The *run-time environment* provides *data-* and *process-oriented views* to end-users; i.e., authorized users may invoke activities for accessing data at any point in time as well as activities needed to proceed with the flow of the process. Moreover, PHILharmonicFlows is based on a well-defined formal semantics, which allows for the automatic generation of end-user components corresponding to the run-time environment (e.g., tables giving an overview of object instances, user worklists, and form-based activities). Consequently, implementation efforts are significantly reduced and only become necessary for realizing black-box activities.

[1] Process, Humans, and Information Linkage for harmonic Business Flows

6 Data Integration

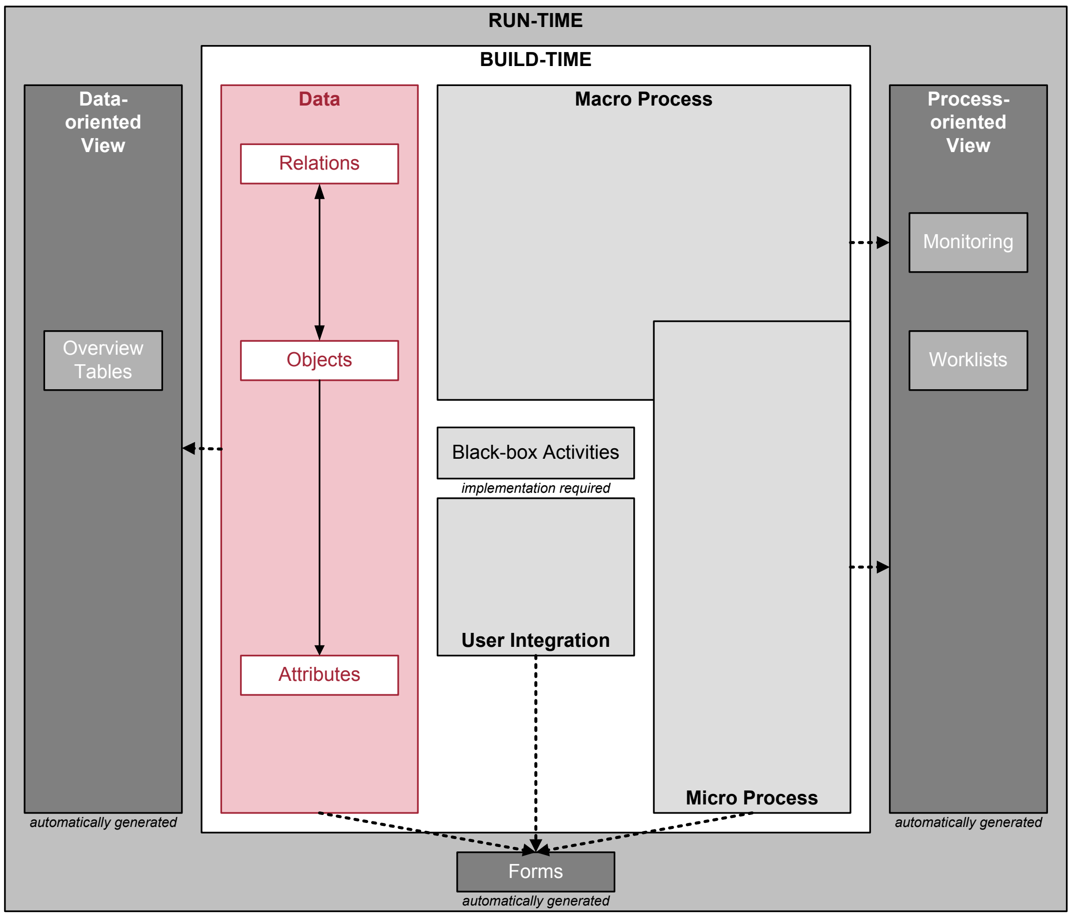


Figure 6.1: Data modeling in PHILharmonicFlows

A DBMS is usually based on a specific *data meta-model* (e.g. relational, hierarchical, or object-relational models) to define the logical *data structure*. Our goal is to cover the fundamental relationships that exist between the different business perspectives, especially the ones between business processes and business data. Hence, PHILharmonicFlows relies on a *relational data model*, which comprises object types, object attributes, and object relations [Cod90]. Note that the framework does not consider object-oriented data models. On one hand, the latter are not common in practice; on the other, their use would distract us from the basic properties of object-aware processes.[1]

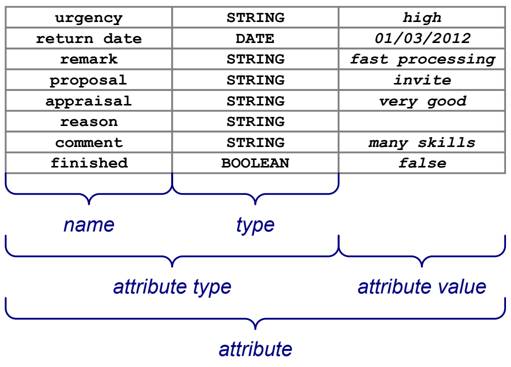
RDBMS enable the definition of business data in terms of relations (i.e., tables). In particular, for each business object type (e.g., order) a corresponding table is defined representing the attributes of the business object as columns (cf. Fig. 1.4b). For each of these attributes, a corresponding data type as well as additional properties are then specified (e.g., the maximal length of its value). At run-time, individual object instances (e.g., Order 1, Order 2, and Order 3) are represented as rows in the corresponding table; each column may contain a value corresponding to the data type of the respective attribute (cf. Fig. 1.4c). At least one of the attributes is defined as primary key. Using its value during run-time, any object instance represented as row in the respective table can be uniquely identified. Relationships between object types are modeled in terms of foreign key attributes. More precisely, a table may contain a column storing values of primary key attributes belonging to any referenced object instance (i.e., row in a table). Thus, individual object instances differ in the values of their attributes and their relationships among each other. Since normalization constitutes an integral part of the relational data model, all relations form 1-to-many relationships; i.e., many-to-many-relationships are dissolved using an intermediate relation. In order to control data access, each registered user can get the permissions to read and write data (i.e., object instances) from several relations. As illustrated in Fig. 1.4d, these permissions may be restricted to particular rows (i.e., individual object instances). For this purpose, several views can be defined, each of them containing a subset of the available rows that fulfill a certain constraint. In addition, permissions can be restricted to particular columns (i.e., individual attributes) as well.

6.1 Object Attributes

Attribute types describe the individual properties of business objects. As illustrated in Fig. 6.2, each attribute type is defined by its *name* and *type* (cf. Def. 1a). To set a focus, according to the traditional relational model, only atomic data types are considered. The latter include *integer*, *decimal*, *string*, *boolean*, and *date*. In turn, complex (i.e., aggregated) data types (e.g., lists or arrays), are captured in terms of relating object types (which are then referenced by the object type they belong to).
When generating user forms at run-time, for each attribute the respective user owns write (read) permissions for, an input field (data field) is automatically created.

In a relational DBMS, attribute types may be optionally flagged as "not null". For them, a value is required when creating an object instance. Whether or not a value is required for a specific attribute at run-time, depends on the course of process execution; i.e., "not null" does not require to write the attribute value when creating the object instance, but necessitates that this is done at some point during process execution. To ensure this, we introduce *micro*

[1]Note that many-to-many relationships between objects can be dissolved by using one-to-many relations.



urgency	STRING	*high*
return date	DATE	*01/03/2012*
remark	STRING	*fast processing*
proposal	STRING	*invite*
appraisal	STRING	*very good*
reason	STRING	
comment	STRING	*many skills*
finished	BOOLEAN	*false*

Figure 6.2: Attribute types and attribute values

processes in Chap. 7. Finally, default attribute values may be used which will then be assigned to respective input fields as initial values.

At run-time, attributes are represented in terms of name-value-pairs. Thereby, any attribute value must belong to the domain of the respective attribute type (cf. Fig. 6.2 and Def. 1a).

Definition 1 (Attribute types and attributes):
In the following, let **Identifiers** be the set of all valid identifiers over a given alphabet.

a) An **attribute type** is a tuple **attrType = (name, type)** where

- name $\in$ Identifiers is an identifier.
- type $\in$ {INTEGER,DECIMAL,STRING,BOOLEAN,DATE} is a basic data type.

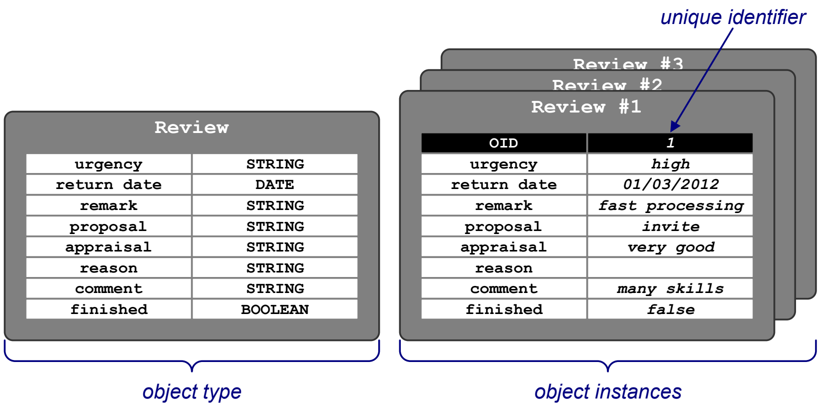
AttrTypes denotes the set of all definable attribute types and **AttrValues** the set of all possible attribute values corresponding to any of the above data types.

b) An **attribute** is a tuple **attr = (attrType, attrValue)** $\in$ AttrTypes $\times$ AttrValues with attrValue denoting a valid value of attrType; i.e., attrValue belongs to the domain of attrType.type. **Attr** denotes the set of all attributes.

6.2 Object Types and Instances

When defining a business object, attribute types may be grouped to a respective object type. As illustrated in Fig. 6.3, each object type comprises a set of attribute types that define its properties. In addition, an object type has a unique name (cf. Def. 2).

At run-time, for each object type a varying number of object instances may be created (cf. Fig. 6.3). Object instances usually differ in the values of their attributes. Note that we do not enforce the definition of primary keys as known from DBMS. Instead, for each object instance, PHILharmonicFlows automatically generates an object identifier (OID) to uniquely identify it at run-time. How to represent such an OID depends on the technical implementation chosen.



Review	
urgency	STRING
return date	DATE
remark	STRING
proposal	STRING
appraisal	STRING
reason	STRING
comment	STRING
finished	BOOLEAN

Review #1

OID	*1*
urgency	*high*
return date	*01/03/2012*
remark	*fast processing*
proposal	*invite*
appraisal	*very good*
reason	
comment	*many skills*
finished	*false*

Figure 6.3: Object types and object instances

Definition 2 (Object types and instances):

a) An **object type** is a tuple **oType = (name, AttrTypeSet)** where

- name $\in$ Identifiers is an identifier.
- AttrTypeSet $\subset$ AttrTypes is a finite set of attribute types.

$\forall$ attrType$_i$ $\in$ AttrTypeSet, i=1,2: attrType$_1$.name = attrType$_2$.name $\Rightarrow$ attrType$_1$ $\equiv$ attrType$_2$.
OTypes corresponds to the set of all definable object types.

b) An **object instance** is a tuple **o = (oid,oType,attrval)** where

- oid is the unique identifier of o; i.e., it holds: for o$_i$ = (oid$_i$, oType$_i$, attrval$_i$), i=1,2 being two instances of a particular object type with oid$_1$ = oid$_2$, $\Rightarrow$ o$_1$ $\equiv$ o$_2$;
  i.e., every object instance is uniquely identified by its object identifier.
- oType = (name, AttrTypeSet) is the object type of o.
- attrval: AttrTypeSet $\mapsto$ AttrValues $\cup$ {NULL} assigns to each attribute type attrType = (name, type) $\in$ AttrTypeSet an attribute value.
  Thereby, attrval(attrType) is compatible with the data type attrType.type.

OInstances denotes the set of all object instances of any object type ot $\in$ OTypes. Furthermore, **oinstances**: OTypes $\mapsto$ $2^{\text{OInstances}}$ assigns to each object type ot the set of object instances (with oinstances(ot) $\subseteq$ OInstances) currently existing for this object type.

Like in the relational data model, attribute types may be flagged as unique (cf. Def. 3); it is not possible to assign the same value for the respective attribute to more than one object instance.[2]

Definition 3 (Unique attributes):
Let ot = (name, AttrTypeSet) $\in$ OTypes be an object type. Then:

unique$_{\text{ot}}$: AttrTypeSet $\mapsto$ BOOLEAN indicates for each object attribute whether its value is unique within the collection of object instances corresponding to object type ot.

[2]Corresponding validations are automatically applied at run-time when editing the generated user forms (cf. Sect. 9.1.1).

6.3 Predefined Attribute Values

For certain attributes, only specific values from a finite value domain may be assigned at run-time. This is usually accomplished based on combo boxes or radio buttons. For example, consider attributes "country name" and "family status". Regarding the latter, only values "single", "married", "divorced", and "widowed" may be set. As illustrated in Fig. 6.4, in PHILharmonicFlows such predefined value sets can be specified in terms of *value types*. Like object types, each value type may comprise a number of attribute types (cf. Def. 4). Usually, only one attribute type is required. In turn, value types comprising several attribute types, are required for realizing more advanced concepts during user form generation (cf. Chap. 13.4 for more details). Finally, each instance of a value type represents a particular value at run-time.

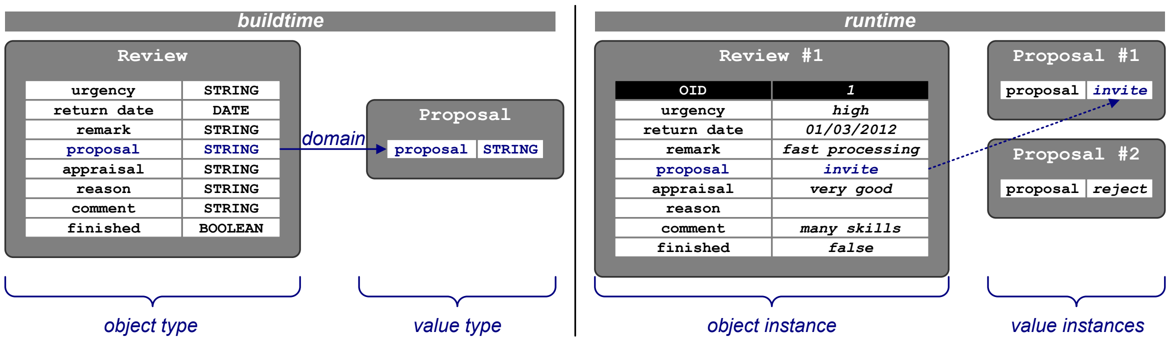


Figure 6.4: Value types and instances

Definition 4 (Value types and instances):

a) A **value type** is a tuple **vType = (name, AttrTypeSet)** where

- name $\in$ Identifiers is an identifier.
- AttrTypeSet $\subset$ AttrTypes is a finite set of attribute types describing vType.

VTypes denotes the set of all definable value types.

b) A **value instance** is a tuple **v = (vType, vattrval)** where

- vType = (name, AttrTypeSet) is the value type of v.
- vattrval: AttrTypeSet $\mapsto$ AttrValues $\cup$ {NULL} assigns to each attribute type attrType $\in$ AttrTypeSet an attribute value vattrval(attrType) compatible with attrType.type.

VInstances denotes the set of all value instances of any type vt $\in$ VTypes.
Finally, **vinstances: VTypes** $\mapsto$ $\mathbf{2^{VInstances}}$ assigns to each value type vt $\in$ VTypes the set of value instances(vt) currently existing for this value type.

Example 6.1 (Value type):
To capture predefined values describing the family status of an `employee`, a value type comprising an attribute type `family status` of data type STRING is defined. Corresponding to this attribute type, instances "single", "married", "divorced", and "widowed" may be created.

A value type is assigned to the respective attribute type(s) of a particular object type. We denote these assignments as *domains*, which are represented in terms of relations (cf. Def. 5).

At run-time, a domain comprises all the values of the value instances the respective attribute type refers to (cf. Fig. 6.4).

Definition 5 (Domain):
Let ot = (name, AttrTypeSet) $\in$ OTypes be an object type. Then:

domain: AttrTypeSet $\mapsto$ **Identifiers** $\times$ **vTypes** $\times$ **(AttrTypes** $\cup$ $\{$**NULL**$\})$
with domain(attrtype) = (label, vType, vattrType)
assigns to the attribute type of the object type ot an attribute type corresponding to a value type vType $\in$ vTypes or NULL (if no such corresponding attribute type exists): vattrType $\in$ vType.AttrTypeSet.

Example 6.2 (Domain):
Attribute type `family status` of the value type introduced in Ex. 6.1 can be defined as *domain*. Hence, corresponding instances "single", "married", "divorced", and "widowed" may then be assignable at run-time.

Like in relational DBMS, object relations are captured "by reference" (i.e., within a relation always one object instance is referenced based on its OID). Opposed to this, instances of value types are referenced "by value"; i.e., the value of the selected instance is directly assigned to the attribute.

6.4 Data Model and Data Structure

As illustrated in Fig. 6.5, a *data model* consists of object types and the relation types between them (cf. Def. 6). Thereby, a *relation type* describes a semantic relationship between a source and a target object type. For each relation type, in addition, a *minimum* and *maximum cardinality* may be specified. In turn, cardinalities set a restriction of the minimum and/or maximum number of instances of the source object type that may reference one and the same target object instance. Since normalization constitutes an integral part of the relational model [KR05, Cod90], all relations form 1-to-many relationships; i.e., many-to-many-relationships must be dissolved in our approach by introducing additional 1-to-many-relations [KR05].

Definition 6 (Data model and data structure):

a) A **data model** is a tuple **dm = (name, OTypeSet, RelTypeSet)** where

- name $\in$ Identifiers is an identifier.
- OTypeSet $\subset$ OTypes is a finite set of object types.
- RelTypeSet $\subset$ Identifiers $\times$ OTypeSet $\times$ OTypeSet $\times$ $\mathbb{N}_0$ $\times$ $(\mathbb{N}_0 \cup \infty)$ is a finite set of relation types with relType = (name, source, target, min, max) $\in$ RelTypeSet having the following meaning:
  - name $\in$ Identifier is an identifier.
  - source $\in$ OTypeSet is the source object type of relType.
  - target $\in$ OTypeSet is the target object type of relType.
  - min $\in \mathbb{N}_0$ corresponds to the minimum cardinality of relType.
  - max $\in \mathbb{N}_0 \cup \{\infty\}$ with max $\geq$ min corresponds to the maximum cardinality of relType.

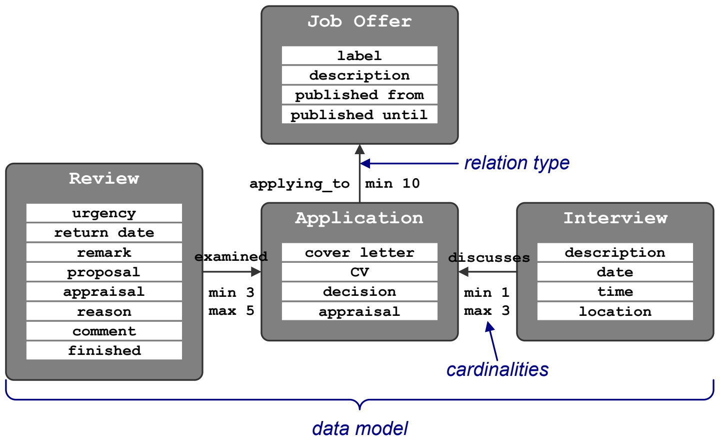


Figure 6.5: Example of a data model

DM denote the set of all definable data models.

b) A **data structure** is a tuple **ds = (dm, OSet, RelSet)** representing an instance of a data model where

- dm = (name, OTypeSet, RelTypeSet) $\in$ DM is the data model ds is based on.
- OSet is a finite set of object instances corresponding to any object type from OTypeSet: OSet $\subseteq$ OInstances $\wedge$ $\forall$ o $\in$ OSet: o.oType $\in$ OTypeSet.
- RelSet $\subseteq$ RelTypeSetType $\times$ OSet $\times$ OSet
  $\wedge$ $\forall$ rel = (relType, soid, toid) $\in$ RelSet:
  rel.relType $\in$ RelTypeSet $\wedge$ soid.oType = rel.relType.source
  $\wedge$ toid.oType = rel.relType.target;
  i.e., RelSet is a finite set of instances of relation types from RelTypeSet.

DS describes the set of all definable data structures.

For each object type, a number of *object instances* exists at run-time. These usually differ in their attribute values and relations among each other (cf. Fig. 6.6).

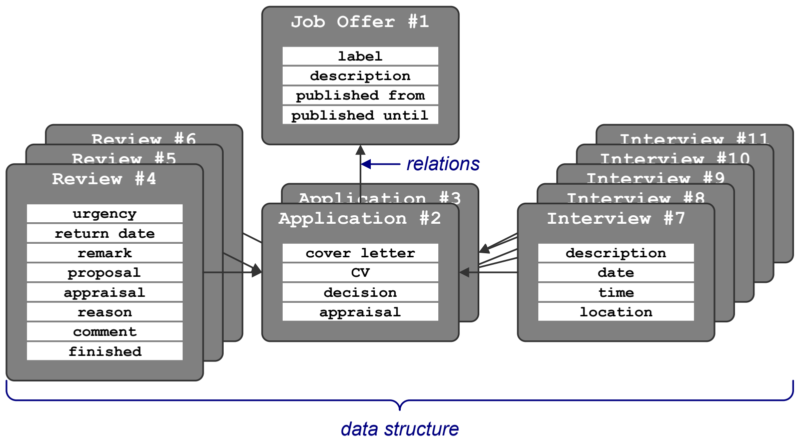


Figure 6.6: Example of a data structure

The number of object instances referencing one and the same target object instance must be between the minimum and maximum cardinality (cf. Def. 7). In this context, note that minimum cardinalities may initially be not fulfilled. Hence, mandatory activities for creating respective object instances are assigned to the worklists of responsible users (cf. Chap. 8.8).

Definition 7 (Cardinality compliance):
Let ds = (dm, OSet, RelSet) $\in$ DS be an instance of data model dm = (name, OTypeSet, RelTypeSet) $\in$ DM. Then:

$\forall$ relType $\in$ RelTypeSet, $\forall$ o $\in$ OSet:
RelSet* := { rel $\in$ relSet | rel.relType = relType $\wedge$ rel.toid = o }, $\Rightarrow$
relType.min $\leq$ | RelSet* | $\leq$ relType.max.

6.5 Data-oriented User View

Based on object types, object attributes and object relations, PHILharmonicFlows allows for automatically generating *data-oriented views* at run-time (cf. Fig. 6.7).

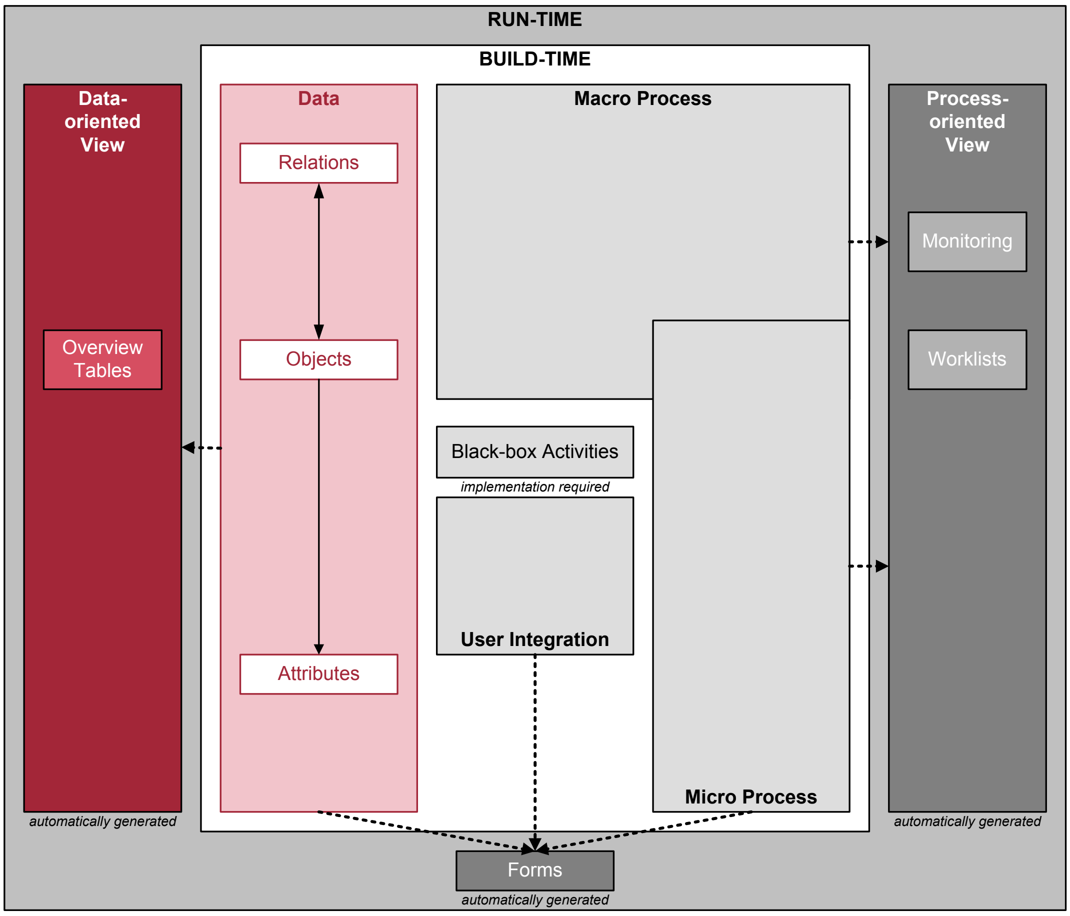
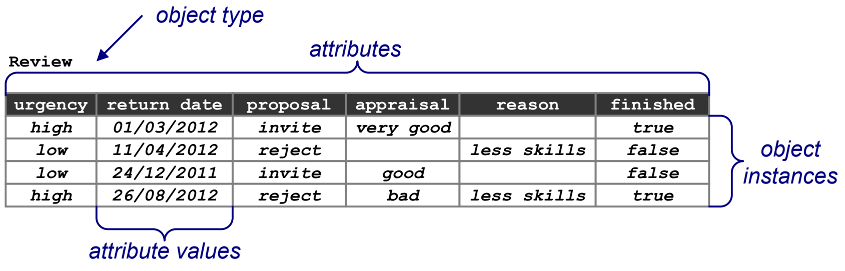


Figure 6.7: Data-oriented view in PHILharmonicFlows

To enable both process management and data integration, PHILharmonicFlows provides a data- and task-oriented view to end-users. Based on a *data-oriented view*, users may access and manage business objects at any point in time. More precisely, users may access respective objects also outside the scope of the activities relevant for process execution (i.e., bypassing mandatory activities assigned to the users worklist). However, data- and task-oriented views should not be treated completely independent from each other. On one hand, when accessing data (optionally) corresponding processing states are important as well. On the other, when executing mandatory activities, optional actions (e.g., for reading and writing additional attribute values) should be enabled as well. In this context, *overview tables* constitute a fundamental run-time concept provided by PHILharmonicFlows. Such an overview table is automatically generated for each object type, listing the corresponding object instances and enabling access to several object-related business functions. The latter include optional activities (e.g., for editing related attribute values) as well as mandatory activities needed to proceed with the flow of the processes to which respective objects correspond to (cf. Chapt. 9 for details). Based on these data- and function-oriented views, the task-oriented view works as filter selecting exactly those object instances for which the respective must execute a particular business function; i.e., overview tables may be used in different context providing the object instances required in the particular situation.

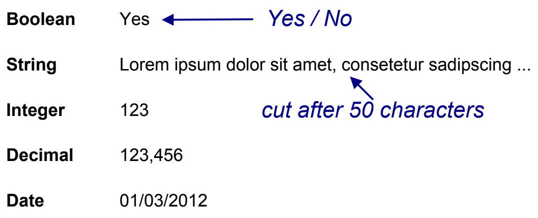
As illustrated in Fig. 6.7, a data model constitutes the basis for generating overview tables at run-time. For each object type, a separate overview table exists. From a user perspective, the instances of a particular object type correspond to rows in a table (cf. Fig. 6.8). In turn, columns relate to selected attributes of the object type or - more precisely - to attribute values of the respective object instances. The name of the attribute type is then used as label for the heading of the corresponding column.


object type
Review
attributes
object instances
attribute values

urgency	return date	proposal	appraisal	reason	finished
high	*01/03/2012*	*invite*	*very good*		*true*
low	*11/04/2012*	*reject*		*less skills*	*false*
low	*24/12/2011*	*invite*	*good*		*false*
high	*26/08/2012*	*reject*	*bad*	*less skills*	*true*

Figure 6.8: Overview table comprising attributes

The way how respective attribute values are displayed depends on the type of the corresponding attribute (cf. Fig. 6.9).



Boolean	Yes
String	Lorem ipsum dolor sit amet, consetetur sadipscing ...
Integer	123
Decimal	123,456
Date	01/03/2012

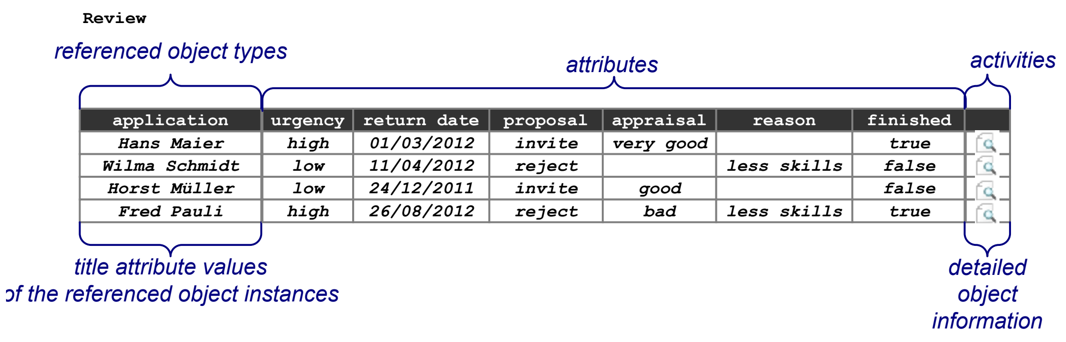
Figure 6.9: Displaying attribute values

As illustrated in Fig. 6.10, in addition to attributes, references to other object instances are displayed in terms of columns as well. At run-time, such relations are expressed based on

OIDs of the referenced object instances (cf. Def. 6). Thus, each relation must be dissolved in order to display the object instance referenced; i.e., the values of the relations are substituted by (meaningful) attribute values of the related object instance.[3] For this purpose, PHILharmonicFlows allows defining *label attributes* (cf. Def. 8), which are displayed in the respective overview tables (instead of the OID itself).

Definition 8 (Label attribute):
Let OTypes be the set of all definable object types and AttrTypes be the set of all definable attribute types (corresponding to any object type). Then:

label: OTypes $\mapsto$ **2**$^{\textbf{AttrTypes}}$ with label(ot) $\subseteq$ ot.AttrTypeSet denotes the attribute types to be used to label a particular object instance at run-time.



application	urgency	return date	proposal	appraisal	reason	finished	
Hans Maier	*high*	*01/03/2012*	*invite*	*very good*		*true*	🔍
Wilma Schmidt	*low*	*11/04/2012*	*reject*		*less skills*	*false*	🔍
Horst Müller	*low*	*24/12/2011*	*invite*	*good*		*false*	🔍
Fred Pauli	*high*	*26/08/2012*	*reject*	*bad*	*less skills*	*true*	🔍

Figure 6.10: Example of an overview table with attributes and relations

The number of attribute types corresponding to a particular object type is usually too large to display them all in the overview table. Hence, we allow hiding attributes or relations (cf. Sect. 13.4). Additional information on object instances (e.g., attributes not displayed or detailed information about referenced object instances) may be viewed on-demand. For this purpose, an optional activity may be invoked for each object instance listed in the overview table (cf. Fig. 6.10).

To determine which object instances (and corresponding attribute values) may be read or written by a user, fine-grained authorization permissions are required (cf. Chapt. 9). The latter also depend on the current processing state of the considered object instance.

6.6 Summary

As a a prerequisite for any integrated access to business processes and business data, PHILharmonicFlows allows defining a (relational) data model, which consists of object types, object attributes, and object relations. Regarding the latter, in addition, minimum and maximum cardinalities may be additionally defined. In particular, such a data model provides the foundation for automatically generating data-oriented user views and user forms at run-time.

[3]In most cases, one attribute is used to describe a particular object instance.

7 Micro Process Modeling

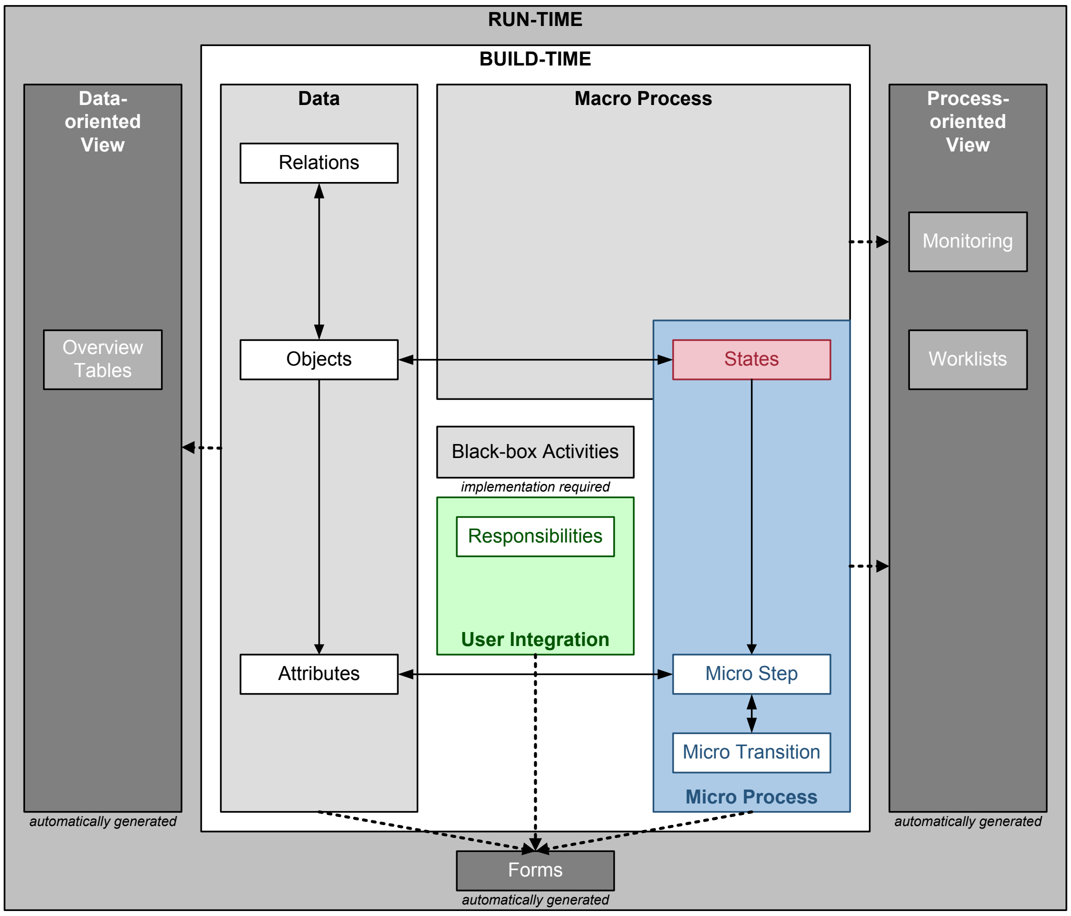


Figure 7.1: Integrating micro process types in PHILharmonicFlows

Currently, two different kinds of process modeling paradigms exist (cf. Chap. 4). While imperative ones enable stringent processes (i.e., guidance how processes shall be executed is provided), declarative modeling paradigms additionally support optional activities which enable loosely structured processes by allowing users to execute activities as long as pre-specified constraints are not violated [vPS09, RW12, FLM^{+}09, FMR^{+}10]. However, neither imperative nor declarative approaches are appropriate to support the properties described in Chap. 3. Exactly for this reason, we introduce a new process modeling paradigm that combines ideas known from declarative approaches (i.e., using conditions) with the ones from imperative approaches.

PHILharmonicFlows enforces a well-defined modeling methodology that governs the process engineer to define processes at different levels of granularity. More precisely, the framework differentiates between *micro* and *macro processes* to be able to capture both *object behavior* and *object interactions*. This chapter introduces the notion of a micro process for capturing and defining object behavior (cf. Fig. 7.1) [KR11a]. In this context, optional as well as mandatory access to data is enabled. More precisely, using *optional activities* (cf. Prop. 8), on one hand, data (i.e., object instances) may be accessed and managed at arbitrary point in time. On the other, certain information (i.e., attribute values) must be available to reach desired processing goals. Therefore, activities for creating object instances as well as for providing required attribute values must be executed. Since these activities are necessary to proceed with the process, we denote them as *mandatory activities*. To realize them, for each *object type* a *micro process type* describing *object behavior* needs to be defined (cf. Prop. 4). As illustrated in Fig. 7.2, the creation of an object instance is coupled with the creation of a respective micro process instance.

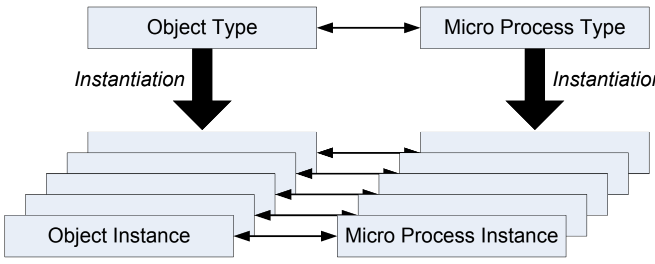


Figure 7.2: Relationship between object and micro process instances

To capture object behavior, the properties defined in Chap. 3 must be realized. First, one must ensure that during the processing of individual object instances certain attribute values are mandatorily set before proceeding with the process (cf. Prop. 6); i.e., the processing state of a particular micro process instance must comply with the attribute values of the corresponding object instance. For this purpose, object behavior is based on data (i.e., object attributes) and thus the data (i.e., attribute values) mandatorily required becomes transparent. In particular, this necessitates a *data-oriented modeling paradigm* rather than an activity-oriented one. Data-oriented process definitions also provide the basis for automatically generating form-based activities during run-time (cf. Prop. 12). A particular requirement is to define the internal flow logic for executing a form-based activity; i.e., the flow between the input fields of the respective form (cf. Prop. 7). For this purpose, modeling for micro processes must be accomplished at a very *fine-grained level* enabling the definition of order dependencies for setting attribute values.
Another challenge is to coordinate the processing of an individual object instance (i.e., assigning its attribute values) among different users. This must be done in a more *coarse-grained*

modeling style rather than be based on particular attribute values. In this context, the modeling paradigm must additionally allow integrating black-box activities (cf. Prop. 13).
To allow for a flexible process execution (cf. Prop. 9), PHILharmonicFlows applies a *data-driven execution paradigm* for micro processes; i.e., the progress of a micro process depends on the availability of certain data. Even though required data might be available, in many cases it is desired that the micro process must not proceed with its execution as long as the user has not explicitly committed his or her work on the respective activity (cf. Prop. 11). In particular, users should be allowed to re-execute activities (cf. Prop. 10). Consequently, process control not only depends on data, but also on the respective responsible user as well.
Another challenge concerns the combinational explosion of the number of user forms to be supported: On one hand, provided input fields not only depend on the permissions of respective user invoking the form, but also on the current state of the process. On the other, varying granularities of user forms must be considered; i.e., user forms may invoke input fields in respect to attribute of several object instances although the micro process instance refers to one particular object instance (cf. Prop. 14).
Finally, the execution of a particular micro process instance does not only depend on the attribute values of the object instance it belongs to, but also on attribute values of related object instances and thereto on the execution of other micro process instances. For this purpose, any micro process should provide proper interfaces for interacting with other micro process instances (i.e., to enable object interactions).

As illustrated in Fig. 7.1, similar to existing approaches capturing object behavior, PHILharmonicFlows provides a *state-based approach*; i.e., (abstract) object states serve as the basis for coordinating the (mandatory) activities of a micro process among different user roles. For this purpose, the latter need to be assigned to the different states of a micro process. Opposed to existing state-based approaches, however, PHILharmonicFlows allows for an explicit *mapping between states and attribute values* ensuring compliance between them; i.e., it is precisely defined which attribute values must be set in order to leave a certain micro process state. To accomplish this, each state comprises a number of (atomic) *micro steps*. Each of these micro steps corresponds to a mandatory write access on a particular attribute of the processed object instance. Note that a single micro step does not represent an activity, but solely refers to one atomic action. By connecting micro steps with *micro transitions*, a default execution order can be expressed. In particular, based on micro transitions, both the internal logic of form-based activities and the coordination of the processing among users can be expressed.

7.1 Micro Process Types

First of all, we introduce *micro process types* defining *object behavior*. Sects. 7.1.1 - 7.1.3 first discuss this kind of process support informally, followed by a formal specification in Sect. 7.1.4. As discussed, for each object type a micro process type must be provided defining the processing of individual object instances at run-time (i.e., object behavior). In this context, one must specify mandatorily required attribute values as well as the order in which they shall be set. As illustrated in Fig. 7.3, each micro process type comprises a number of *micro step types* of which each refers to a particular object attribute type. In turn, micro step types are connected with each other using *micro transition types*. Based on this, their default execution order can be expressed; i.e., the order in which respective attribute values shall be set (cf. Fig. 7.3). At run-time, such a micro step will be reached when a value for its corresponding attribute is set. Hence, the respective attribute is mandatory. To enable the process flexibility required in practice, however, authorized users may provide a value for attributes required later one. For this case, the (subsequent) micro step for writing this attribute will be automatically reached since the value for the respective attribute is already available.

Example 7.1 (Micro steps and micro transitions):
Consider the `review` micro process type as illustrated in Fig. 7.3. Each micro step type refers to a particular attribute type of the respective object type (except the start micro step type); e.g., micro step type `urgency` refers to attribute type `urgency`. In turn, micro transition types define the default execution order of the micro steps; e.g., after assigning a value to attribute `urgency`, a value for attribute `return date` is mandatorily required. However, a value for attribute `return date` may be also set before writing attribute `urgency`. Then, the subsequent micro step `return date` will be automatically reached since the value of attribute `urgency` will be available.

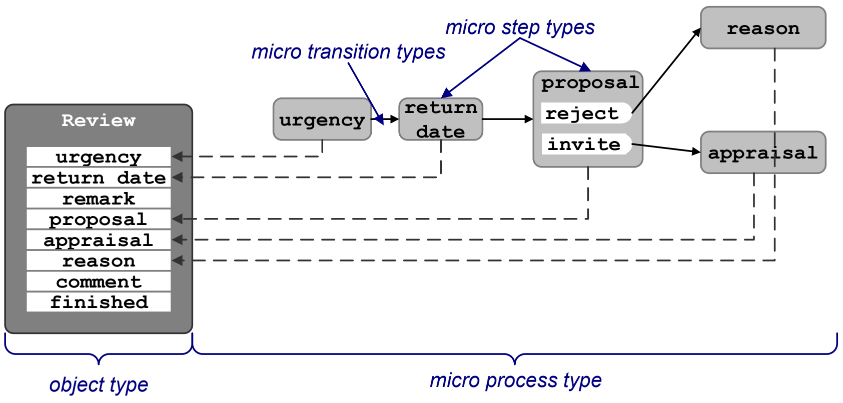


Figure 7.3: A micro process type with micro step and micro transition types

7.1.1 Micro Step Types

As opposed to activity-centric process modeling, where each process step corresponds to a predefined activity, a *micro step type* refers to a particular *attribute type*. In turn, the latter corresponds to the object type the micro process type is defined for. This way, each micro step type describes an atomic action for writing the respective attribute (cf. Ex. 7.1). In addition, the processing of individual object instances not only includes the setting of its attribute values, but also the assignment of relations to other object instances. As illustrated in Fig. 7.4, a micro step type may therefore also refer to a *relation type* (whose source object type corresponds to the object type of the micro process type).

A micro process type may contain micro step types not referring to any attribute or relation type. Respective micro step types usually represent *start* and *end micro step types* (cf. Fig. 7.5).

Generally, there exist different kinds of micro step types (cf. Fig. 7.6). Micro step types not referring to any attribute or relation type are denoted as *empty micro step types*. An empty micro step is immediately reached when one of its incoming micro transitions becomes activated. As example consider the end micro step type in Fig. 7.5. Another example of empty micro steps are start micro steps. These are immediately reached when the corresponding object instance is created; i.e., a value for the oid of the corresponding object instance is set. Further,

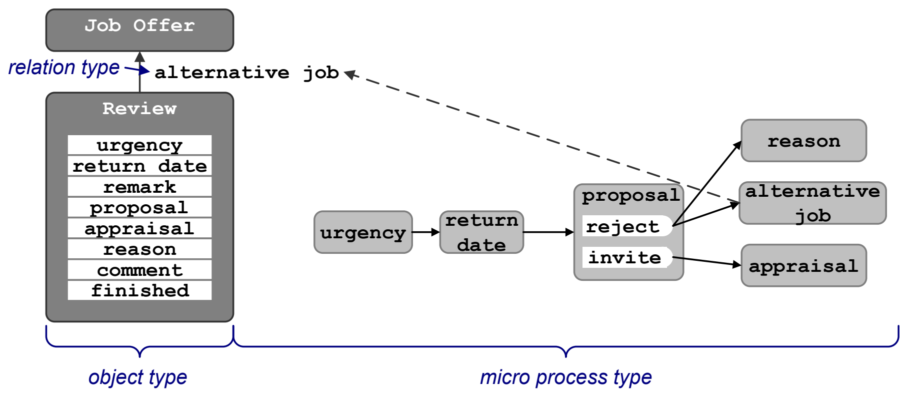


Figure 7.4: A micro process type with a micro step type referring to a relation type

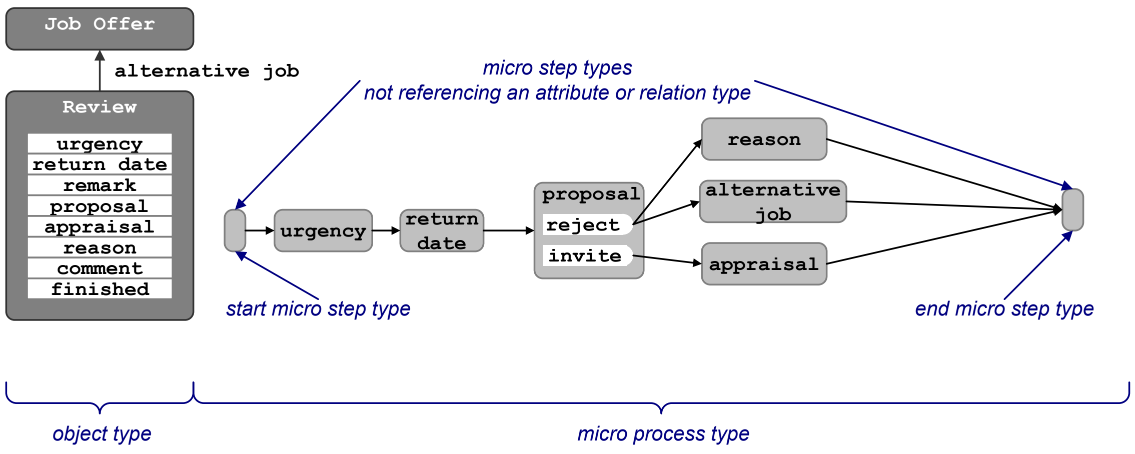


Figure 7.5: Micro step types not referencing an attribute or relation type

micro step types referring to an attribute or relation type are denoted as *atomic micro step types* (cf. Fig. 7.6). At run-time, atomic micro steps are reached when a value for the corresponding attribute or relation becomes available. If the value for a particular attribute is missing, a mandatory activity for writing this attribute is assigned to the worklist of a responsible user (cf. Sect. 8.8). Finally, micro step types comprising value step types are called *value-specific micro step types*. As example consider micro step type `proposal` as illustrated in Fig. 7.5 (cf. Sect. 7.1.2 for details).

7.1.2 Value Step Types

Whether or not certain attribute values are mandatory for an object instance may also depend on the values of other object attributes; i.e., while for a particular attribute "arbitrary" values may be set, other ones require specific values. To reflect this, the specification of a micro step type can be extended. More precisely, different values (or value ranges) of an object attribute can be considered in case they influence object behavior. Therefore, any *value-specific micro step type* may comprise a number of *value step types* of which each represents a predicate defined

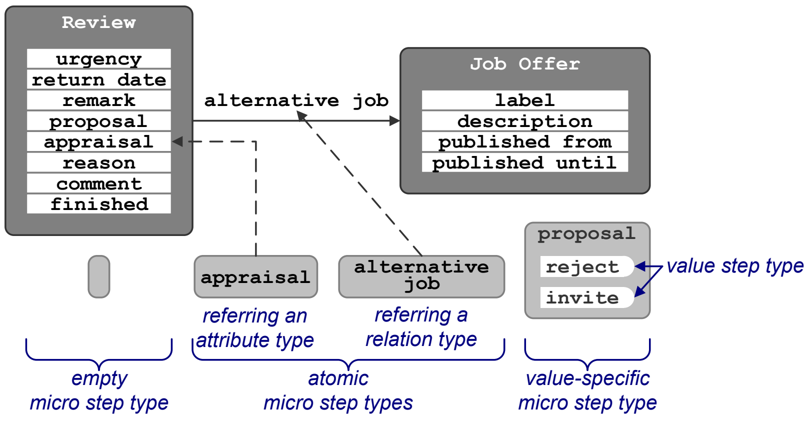


Figure 7.6: Different kinds of micro step types

in respect to the object attribute referred by the micro step type. In addition to the micro step type itself, a particular value step type can be used as source for micro transition types as well. These micro transitions are then activated at run-time if the respective condition of the micro value step evaluates to true.

Example 7.2 (Value-specific micro step types):
Consider micro step type `proposal` in Fig. 7.5. Its corresponding attribute either may have value "reject" or "invite"; i.e., for these values corresponding value step types will be defined. If value step `reject` is reached, either a value for attribute `reason` or a reference to an `alternative job` is then mandatorily required (i.e., either micro step `reason` or micro step `alternative job` may be reached next). Opposed to this, if value step `invite` is reached, a value for attribute `appraisal` is mandatory; i.e., micro step `appraisal` is reachable.

A *value step type* represents a *constraint* requiring a specific value (or value range) to be set. In principle, atomic as well as value-specific micro step types represent a basic *data condition* in relation to an object attribute. Concerning an atomic micro step type, the data condition simply corresponds to *'object attribute != null'* (like for the micro step corresponding to object attribute `urgency`). A value-specific micro step, in turn, realizes a more precise data condition based on specific values (e.g., the data condition of micro step `proposal` corresponds to `proposal` = 'reject' or `proposal` = 'invite'). When defining such constraints, one must ensure that the data type of the specified values is compliant with the one of the corresponding attribute. Which options exist for predicate definition depends on the implementation of the PHILharmonicFlows framework. For example, more advanced implementations may additionally allow for the use of AND- and OR-operators (including comparisons with system variables, system functions, or values of other attributes).

Example 7.3 (Different possibilities for defining value-specific micro step types):
Different possibilities for defining value-specific micro step types, as illustrated in Fig. 7.7, exist:

a) Attribute `number of internships` has type INTEGER. Hence, the constraint of the corresponding value step type may only use predicates referring to INTEGER values (e.g., `number of internships` >= 1).

b) Attribute type `finished` has type BOOLEAN. Hence, the value step type must only be based on predicates with values "true" and "false" (e.g., `finished` = true, `finished` = false).

c) Attribute `return date` has type DATE. In this context, predicates may use system variables like TODAY to compare the attribute value with the current date.

d) Attributes `current salary`, `bonus`, and `wished salary` have type DECIMAL. In order to define a value range for attribute `wished salary`, for example, the OR operator can be used. In addition, the `wished salary` is compared with the `current salary` and the `bonus`.

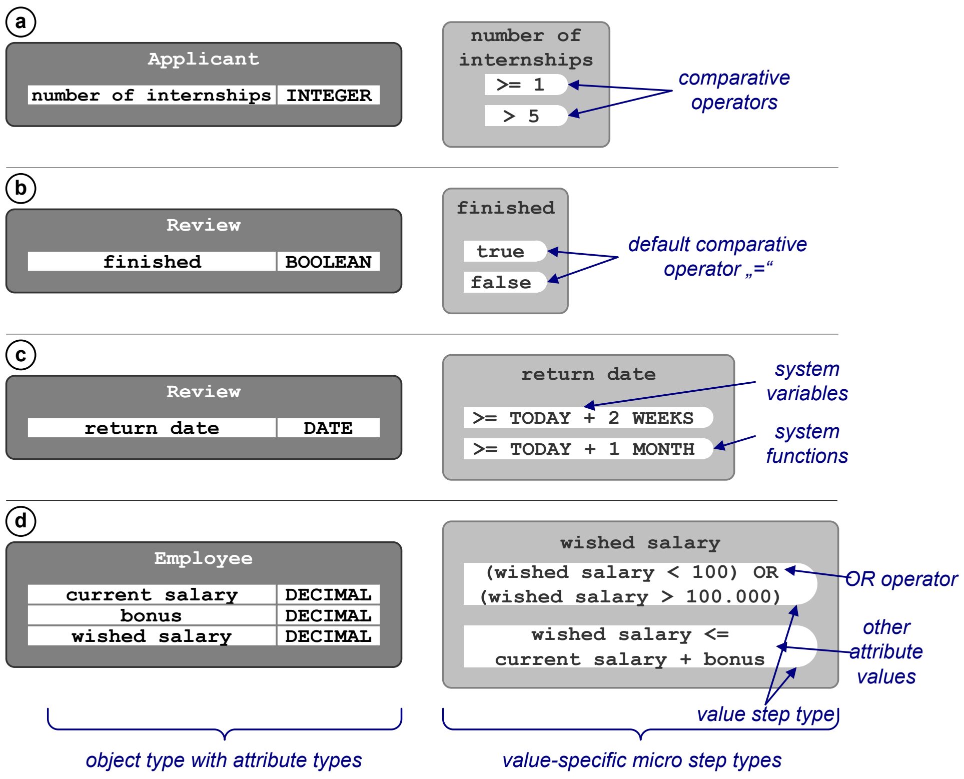


Figure 7.7: Value step types

If a *domain* (cf. Sect. 6.3) is defined for the respective attribute types, the values specified in the context of a predicate of a corresponding value step type must belong to this domain; i.e., only instances of the value type the domain refers to can be used.

Example 7.4 (Value step types with domains):
Consider the predefined values of attribute `proposal` as illustrated in Fig. 7.8. Here, the corresponding value type comprises instances of values "reject" and "invite"; i.e., only these two values may be assigned to attribute `proposal` at run-time and hence be used within predicate definitions.

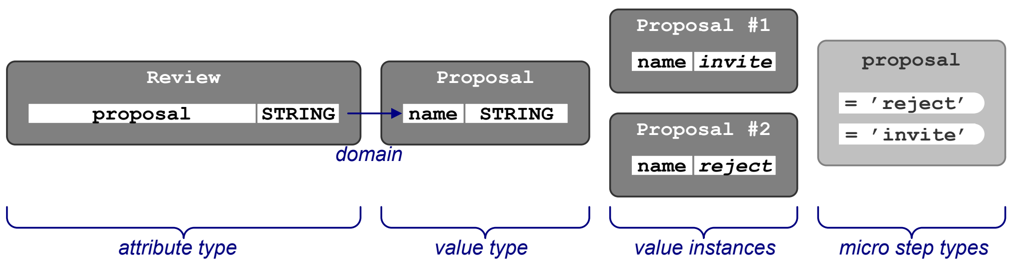


Figure 7.8: Value step types using domains

7.1.3 Micro Transition Types

To define the order in which attribute values and relations shall be set during the processing of a particular object instance, micro step types may be inter-connected using *micro transition types* (cf. Fig. 7.3). Note that this does not restrict the flexibility for processing object instances; i.e., the predefined sequences of micro step types only define a default execution order. Based on additional permissions (cf. Sect. 9.1.1), however, users may set required attribute values before the corresponding micro step is reached. In such a case, the micro steps corresponding to already written attribute values will be immediately completed once they are reachable; i.e., a *data-driven execution* is supported. In addition, it is possible to change attribute values afterwards.
A micro transition type may also use a value step type as source. This way, one can express that alternative paths depend on specific attribute values (e.g., micro step type `proposal` in Fig. 7.5). However, a value step type may only be source of a micro transition type. In turn, the target of a micro transition type is always a micro step type. In order to define alternative execution paths based on user decisions, a micro step type may have more than one outgoing micro transition type (see value step type `reject` in Fig. 7.5). As illustrated in Fig. 7.5, it depends on the user whether he assigns a value to attribute `reason` or `alternative job`. If values for both attributes are provided (e.g., a value for attribute `reason` as well as for relation `alternative job` is available), and needs to ensure that only one of them is fired at run-time. Only then exactly one micro step and hence one state can be reached (cf. Sect. 7.2). Otherwise, several processing states may be activated concurrently resulting in a large number of state combinations and hence a large state space that might be difficult to handle [VRv11]; e.g., it would not be possible to precisely identify the current processing state of an object instance at run-time.
Note that although PHILharmonicFlows ensures that each object instance is always in one processing state, this does not prohibit parallel execution. During the execution of a particular activity, parallel processing of disjoint sets of mandatory as well as optional attributes is possible. More precisely, while a certain user may read or write attributes (and relations) of a particular object instance using a specific form, other users may concurrently edit forms corresponding to the same object instance. In this context, known mechanisms for synchronizing concurrent data access should be applied [KE11]. Moreover, a high degree of concurrency and flexibility exist for processing and coordinating multiple object instances (of same or different type). In particular, a high number of inter-related micro process instances (of same and different type) may be executed concurrently and be synchronized where needed. Despite this asynchronous execution, PHILharmonicFlows allows processing several object instances through executing a specific activity; i.e., PHILharmonicFlows enables vertical flexibility (see Sect. 9.1.1 for details on the parallel processing of object instances of different type and for processing several object instances of the same type in one go in Sect. 9.3).

To ensure that only one alternative path of a micro process is chosen at the same time, the data-driven execution paradigm must be taken into account; i.e., a subsequent micro step will be reached when providing a value for its corresponding attribute (or relation).[1] Consequently, if the attribute values of several subsequent micro steps are available, in principle, all of them could be activated, which must be prevented.

[1]except it is an empty micro step referring to no attribute or relation.

Example 7.5 (Problems with activating subsequent micro steps):
When assigning value "reject" to attribute `proposal`, the corresponding micro value step is reached (cf. Fig. 7.4). Then, all subsequent micro steps may be concurrently reached if values for attribute `reason` as well as relation `alternative job` are available.

To prevent this behavior, micro transition types originating from the same source micro step type or value step type are associated with different *priorities*. The latter may be evaluated at run-time to decide which of the reachable micro steps shall be activated in case of ambiguities; i.e., then only the micro transition with the highest priority is enabled to activate its target micro step.

Example 7.6 (Activating subsequent micro steps using priorities):
Consider Fig. 7.9 when activating value step `reject`, micro steps `reason` and `alternative job` may be reached if for both attributes a value is available. Since the micro transition targeting micro step `alternative job` has higher priority, solely this micro step will be reached (even though a value for attribute `reason` exists as well).

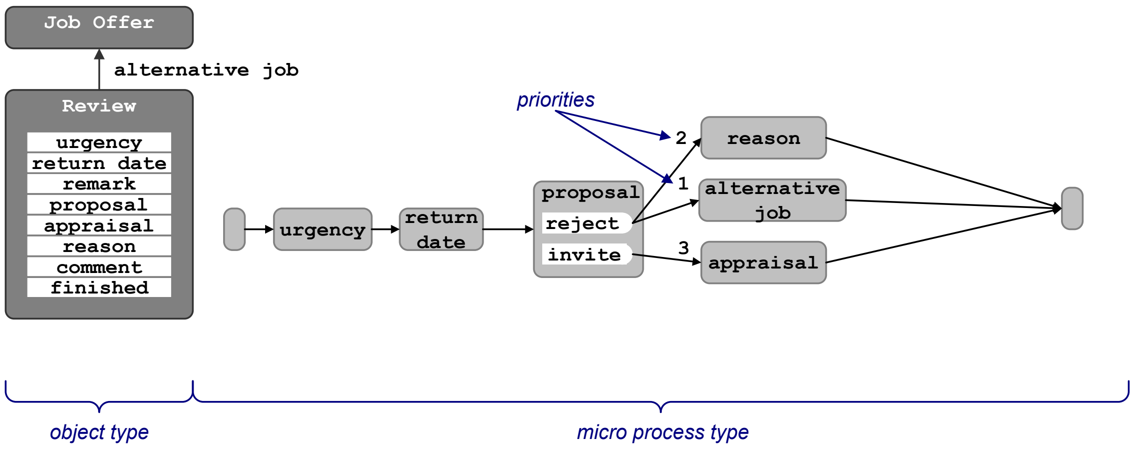


Figure 7.9: Associating micro transition types with priorities

Note that only those outgoing micro transitions (and their priorities) must be evaluated for which the required value of the target micro step is available. Hence, a micro transition with lower priority may be triggered, if for all other ones with higher priority the target micro step is not reachable; i.e., no value for its corresponding attribute or relation exists (cf. Figs. 7.10a and 7.10b).

Example 7.7 (Evaluating priorities):
Consider Fig. 7.10b. Even though the micro transition targeting at micro step `reason` has a lower priority than the one targeting at micro step `appraisal`, `reason` may be reached if a value for its attribute exists, but no attribute value for micro step `appraisal` is available. In turn, if there exist attribute values for both subsequent micro steps, only the micro transition with the higher priority will be activated (cf. Fig. 7.10c).

A micro transition originating from a value step type must be treated in a special way. A value-specific micro step type might comprise a number of value step types that are defined based

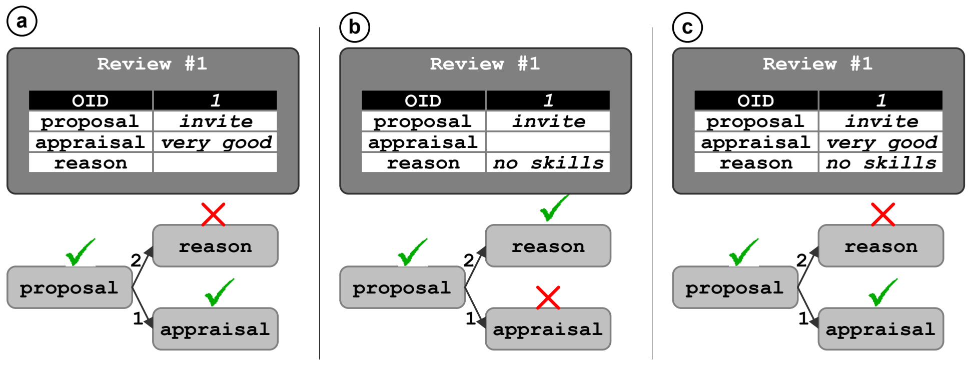


Figure 7.10: Priorities for micro transition types

on predicates. As discussed, in principle, PHILharmonicFlows allows for arbitrary predicates. Hence, the predicates of several value step types might evaluate to true at run-time. Consequently, several micro transitions originating from different value steps may be reached at the same time (cf. micro step `number of internships` in Fig. 7.11). In such a case, only those micro transitions may be reachable for which an attribute value required by the target micro step is available; i.e., only those for which a respective attribute value is available may become activated (cf. Fig. 7.11a). If attribute values relevant for reaching more than one micro step exist (cf. Fig. 7.11b), again only the micro transition with the highest priority is selected.

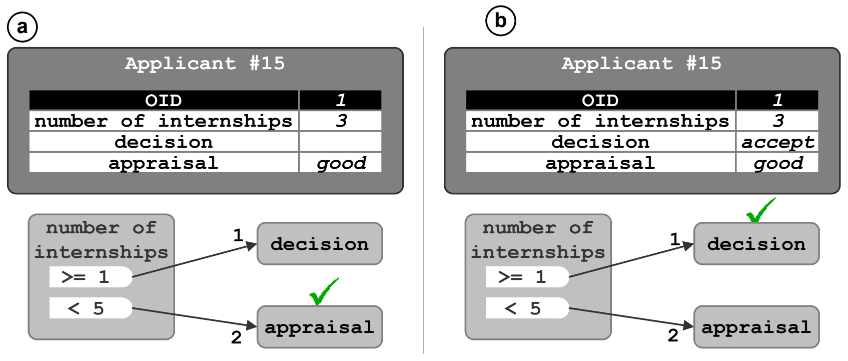


Figure 7.11: Handling value step types with overlapping predicates

Generally, we can not ensure that the value step types defined for a particular micro step type cover all possible attribute values. Especially, certain attribute values might not be considered; i.e., no predicate of any value step will evaluate to true at run-time. For example, assume that value "accept" is assigned to attribute `proposal` (cf. Fig. 7.12a). Since only attribute values "reject" and "invite" are covered by respective value steps, this might lead to a deadlock at run-time. To prevent such errors already at build-time, an outgoing micro transition type should be associated with the micro step type itself (cf. Fig. 7.12b). This way, it becomes possible to cope with arbitrary attribute values for which the predicates of all value steps evaluates to false.

For certain use cases, the definition of such a default transition is not adequate. Instead, certain values are mandatorily required. As illustrated in Fig. 7.13, the micro process instance may only proceed if value "true" is assigned to attribute `finished`. PHILharmonicFlows therefore provides concepts for handling deadlock situations at run-time (cf. Chap. 13.2).

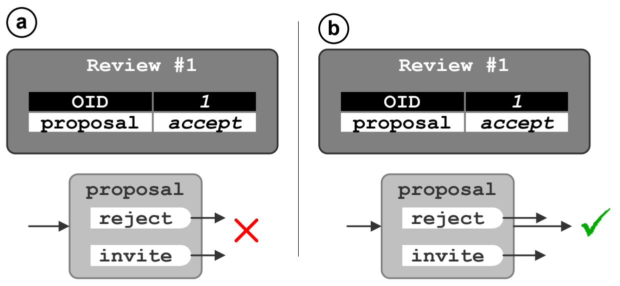


Figure 7.12: Preventing deadlocks at build-time

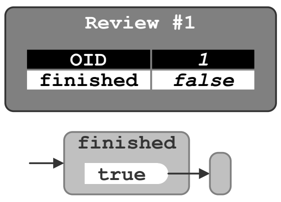


Figure 7.13: Scenario without default micro transition type

7.1.4 Formal Definition of Micro Process Types

In this section, we formally define the notion of a micro process type:

Definition 9 (Micro process types):
Let dm = (name, OTypeSet, RelTypeSet) be a data model. Then:

A **micro process type** is an acyclic graph defined as tuple **micProcType = (oType, MicStepTypeSet, MicTransTypeSet)** with the following properties:

- oType = (name, AttrTypeSet) $\in$ OTypeSet is the object type whose behavior is described by micProcType.
- MicStepTypeSet is a finite set of micro step types. A micro step type **micStepType = (refType, ValueStepTypes)** $\in$ MicStepTypeSet has the following properties:
  - refType $\in$ AttrTypeSet is an attribute type, or
    refType $\in$ RelTypeSet is a relation type with refType.source = oType, or
    refType = NULL is undefined (i.e., the micro step type is empty).
  - ValueStepTypes is a set of boolean predicates with: predicate: ValueStepTypes $\mapsto$ Boolean
- MicTransTypeSet $\subset$ MicStepTypeSet $\times$ MicStepTypeSet $\times$ $\mathbb{N}$ is a finite set of micro transition types with a micro transition type **micTransType = (source, target, priority)** $\in$ MicTransTypeSet having the following properties:
  - source $\in$ MicStepTypeSet $\cup$ ValueStepTypes is the source micro step type or source value step type of micTransType.
  - target $\in$ MicStepTypeSet is the target micro step of micTransType.
  - priority $\in$ $\mathbb{N}$ is the priority of micTransType.

MicProcTypes denotes the set of all definable micro process types.

As a prerequisite for the correct execution of micro process instances during run-time, a number of structural properties must be met. The purpose of these properties is to prevent deadlocks and livelocks as far as possible at run-time. However, when concurrently processing several object instances, certain real-world scenarios may require the occurrence of deadlocks to some degree. To deal with such situations during run-time, PHILharmonicFlows provides respective concepts. We therefore first introduce functions for structurally analyzing micro process types. In particular, for each micro step type *intransCount* (cf. Def. 10) determines the number of its incoming micro transition types. In turn, function *outtransCount* returns the number of micro transition types outgoing from a particular micro step type. Finally, for each micro step type *intrans (outtrans)* determines its incoming (outgoing) micro transition types (cf. Def. 10).

Definition 10 (Functions for structurally analyzing micro process types):
Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet) $\in$ MicProcTypes be an acyclic micro process type. Then:

- **intransCount: MicStepTypeSet** $\mapsto \mathbb{N}_0$ determines the number of incoming micro transition types of a micro step type.
- **outtransCount: MicStepTypeSet** $\cup$ **ValueStepTypes** $\mapsto \mathbb{N}_0$ determines the number of micro transition types outgoing from a micro step type or any of its value step types.
- **intrans: MicStepTypeSet** $\mapsto$ **MicTransTypeSet** determines the set of incoming micro transition types of a micro step type.
- **outtrans: MicStepTypeSet** $\cup$ **ValueStepTypes** $\mapsto$ **MicTransTypeSet** determines micro transition types outgoing from a micro step type or any of its value step types.

Empty micro step types, having no incoming micro transition types, are denoted as *start micro step types*. In turn, empty micro step types having no outgoing micro transition types are called *end micro step types* (cf. Def. 11).

Definition 11 (Start and end micro step types):
Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet) $\in$ MicProcTypes be a micro process type. Then:

- **startMicStepType** of micProcType is the only micro step type micStepType $\in$ MicStepTypeSet
  with intransCount(micStepType) = 0 $\wedge$ micStepType.refType = NULL.
- **endMicStepType** of micProcType is a micro step type micStepType $\in$ MicStepTypeSet
  with outtransCount(micStepType) = 0 $\wedge$ micStepType.refType = NULL.
  Further, **EndMicStepTypes** := {micStepType $\in$ MicStepTypeSet | micStepType is an end micro step type} denotes the set of all defined end micro step types.

Taking solely micro step types and their incoming and outgoing micro transition types into account, a micro process must be *acyclic* (cf. Def. 12a). Backward jumps and the resetting of attribute values are enabled by connecting different state types of a micro process type (cf. Chap. 7.5 for details). According to Def. 12b, each micro process type contains exactly one start micro step type not referring to any attribute or relation type and having no incoming micro transition type. Further, a micro process type must comprise at least one end micro step type not referring to an attribute or relation type and not having any outgoing micro transition type (cf. Def. 12c). All other micro step types, in turn, must have at least one incoming as well as one outgoing micro transition type (cf. Def. 12d and e). To ensure that only one micro step becomes activated at the same time during run-time, micro transition types having the same source micro step type (or one of its value step types) must be associated with priorities (cf. Def. 12f). Note that these properties ensure the reachability of each micro step type (including end micro process types). In particular, each micro step can be reached starting from the start micro step. Finally, from any micro step at least one end micro step can be reached.

Definition 12 (Structural properties of micro process types):
Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet) $\in$ MicProcTypes be a micro process type referring to an object type oType = (name, AttrTypeSet) $\in$ OTypes. Let micStepType = (refType, ValueStepTypes) $\in$ MicStepTypeSet be a micro step type. Then:

a) micProcType is acyclic.

b) $\exists!$ micStepType $\in$ MicStepTypeSet with micStepType = startMicStepType;
i.e., there exists exactly one start micro step type.

c) EndMicStepTypes $\neq \emptyset$;
i.e., there exists at least one end micro step type.

d) $\forall$ micStepType $\in$ MicStepTypeSet - {startMicStepType}:
    intransCount(micStepType) ≥ 1;
i.e., all micro step types except the start micro step type have at least one incoming micro transition type.

e) $\forall$ micStepType $\in$ (MicStepTypeSet $\cup$ ValueStepTypes) - EndMicStepTypes:
    outtransCount(micStepType) ≥ 1;
i.e., all micro step types (except end micro step types) and all value step types have at least one outgoing micro transition type.

f) transType$_i$ $\in$ MicTransTypeSet with
    transType$_i$.source = micStepType $\vee$ transType$_i$.source $\in$ ValueStepTypes, i=1,2 $\wedge$
    transType$_1$ $\neq$ transType$_2$ $\wedge$ transType$_1$.source = transType$_2$.source, $\Rightarrow$
        transType$_1$.priority $\neq$ transType$_2$.priority;
i.e., micro transition types with the same source micro step type (value step type) must not have the same priority.

7.2 State Types

Section 7.1 discussed order constraints for setting attribute values. We use a fine-grained modeling style for which process steps correspond to particular attributes to accomplish this. To coordinate the setting of the attribute values among different users, however, a more coarse-grained modeling style is required. In addition, micro process support should allow for the integration of black-box activities and provide an adequate interface for synchronizing the execution of micro process instances at run-time. For this reason, *state types* are used to realize mandatory activities and coordinate the processing of individual object instances among different users. Generally, a state may be expressed in terms of a particular data condition on selected attributes of the respective object type; i.e., each state postulates specific attribute values to be set.

Example 7.8 (Data condition of states):
Consider object type `review` and its states as depicted in Fig. 7.14. In state `initialized`, values for attributes `urgency` and `return date` are required. In turn, this corresponds to the following data condition: `urgency` != NULL

and `return date` != NULL. Regarding state `pending`, the data condition is as follows: (`proposal` = "reject" and (`reason` != NULL or `alternative job` != NULL)) or (`proposal` = 'invite' and `appraisal` != null).

A state type comprises a subset of micro step types. Note that single micro step types do not represent activities as known from traditional PrMS, but refer to an atomic action (e.g., editing an input field within a form). More precisely, each micro step type represents a mandatory write access to a particular object attribute. In this context, it is not essential which activity is executed to set the required attribute values; i.e., one may use a black-box activity as well as a form-based one for this purpose. Regarding the latter, micro step types belonging to the same state type indicate which input fields must be mandatorily written by an activity. In turn, micro transition types, connecting micro step types of the same state type, define their order and the dependencies between the input fields; i.e., whether or not a particular input field is mandatory may depend on value settings of other input fields. Hence, they reflect the internal logic of a form-based activity. Note that micro process instances are executed during the processing of form-based activities. This way, additional input fields may become mandatory on-the-fly.
Note that a micro process type only defines mandatory input fields and their dependencies. During the execution of form-based activities, it should be further possible to optionally read and write additional attribute values or relations if the user has respective permissions (see Sect. 9.1.1 for details).
Finally, state types can be used to coordinate the execution of several activities among different user roles. For this purpose, state types may be associated with different user roles.

Example 7.9 (User assignment to states):
Fig. 7.14 shows the micro process type describing the behavior of the `review` object type. Before end state `finished` may be reached, each `review` must be created by a `personnel officer` and then be filled out be an `employee`.

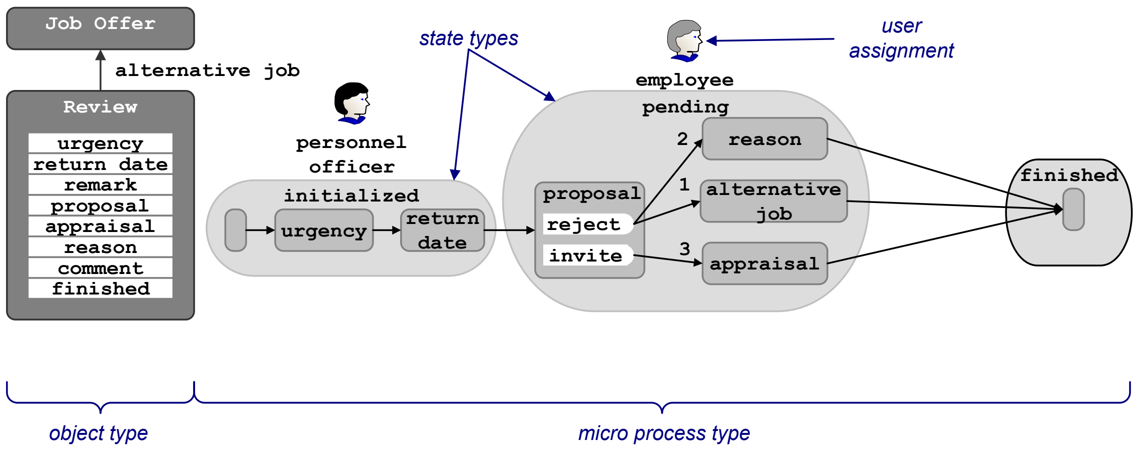


Figure 7.14: State types and related user assignments

According to Def. 13, a state type has a name and a number of micro step types.

Definition 13 (Extending micro process types with state types):
Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet) $\in$ MicProcTypes be a micro process type. Then:

A **state type** of micProcType is a tuple **stateType = (name, sMicStepTypeSet)** where

- name $\in$ Identifiers is an identifier.
- sMicStepTypeSet $\subset$ MicStepTypeSet is a finite set of micro step types.

StateTypeSet$_{\textbf{micProcType}}$ is a finite set of state types defined for micProcType.
To explicitly consider state types in the following, a micro process type is enriched by respective information; i.e., it represents a tuple **micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet)**.

We denote the state type containing the start micro step type as *start state type*, in turn, each *end state type* comprises exactly one end micro step type (cf. Def. 14).

Definition 14 (Start and end state types):
Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet) $\in$ MicProcTypes be an acyclic micro process type. Then:

- **startStateType** $\in$ StateTypeSet is that state type of micProcType containing the start micro step type:
  startMicStepType$_{\text{micProcType}}$ $\in$ stateType.sMicStepTypeSet.
- **endStateType** $\in$ StateTypeSet is a state type of micProcType containing exactly one end micro step type:
  $\exists!$ micStepType $\in$ stateType.sMicStepTypeSet withmicStepType $\in$ EndMicStepType$_{\text{micProcType}}$.
  EndStateTypes denotes the set of all end state types defined for micProcType.

State types may not only correspond to mandatory activities for writing certain attribute values or relations. By adding empty micro step types, it is further possible to define mandatory activities enabling required user commitments even if no attribute value must be mandatorily set. For these states, additional read permissions may be granted to the respective user role (see Sect. 9.1.1 for details).

Example 7.10 (State types comprising empty micro step types):
Consider Fig. 7.15. The depicted micro process type comprises five state types representing mandatory activities and involving two user roles. After the `employee` has filled in the `review` in state `pending`, the `personnel officer` must commit his awareness of the results. Depending on the statement provided by the `employee`, the `review` either reaches state `reject proposed` or `invitation proposed`. Since in both states no attribute values must be set, the states only comprise empty micro steps.

Regarding state types, a number of structural properties must be ensured at build-time. First, each micro step type must belong to exactly one state type (cf. Def. 15a). Hence, the state types of a micro process type must comprise disjoint sets of micro step types. Second, an object instance is always in one processing state, which is precisely defined by a set of elementary actions. Second, each end micro step type must belong to a state type comprising no other micro step types (cf. Def. 15b). If an end state is activated at run-time, no further actions are mandatorily required. Third, there exists exactly one start state type containing the start micro

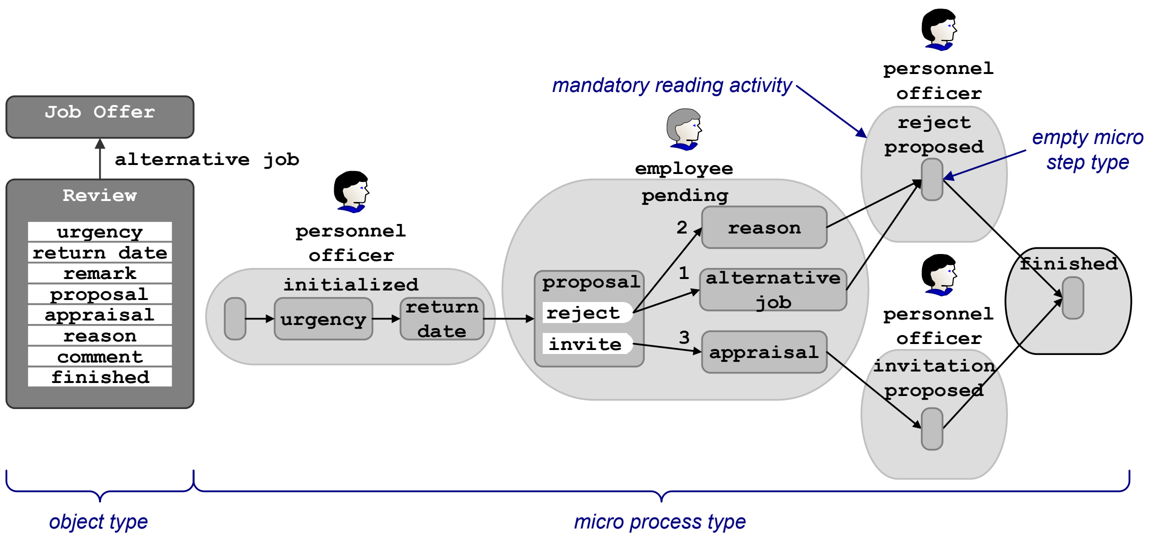


Figure 7.15: Mandatory reading activities

step type (cf. Def. 15c). Opposed to end state types, the start state type may include additional micro step types; i.e., within the start state, mandatory interactions might be required. Fourth, the micro step types belonging to the same state type must be inter-connected (cf. Def. 15d). More precisely, the sub-graph of a micro process type, induced by the micro step types of a particular state type, is compound. Otherwise, it would be possible to activate one and the same processing state several times (cf. Fig. 7.16). Such behavior would be ambiguous for end users; i.e., they would not be able to distinguish such a processing from backward jumps. Finally, regarding form-based activities, these micro step types are typically represented as mandatory input fields (or other form components). Hence, different micro step types of a state type must not refer to the same attribute type or relation type (cf. Def. 15e).

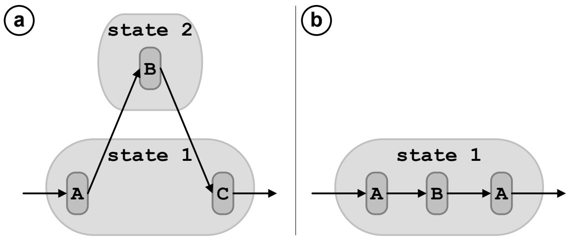


Figure 7.16: Disallowed structures for state type definitions

Definition 15 (Structural properties of state types):
Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet) ∈ MicProcTypes be a micro process type. Let further stateType_i ∈ StateTypeSet, i=1,2 be two state types. Then:

a) ∀ micStepType ∈ MicStepTypeSet: ∃! stateType ∈ $\text{StateTypes}_{\text{micProcType}}$ with
micStepType ∈ stateType.sMicStepTypeSet;
i.e., each micro step type must belong to exactly one state type.

b) ∀ micStepType ∈ $\text{EndMicStepTypes}_{\text{micProcType}}$ with∃! stateType ∈ StateTypeSet:
micStepType ∈ stateType.sMicStepTypeSet ∧ | stateType.sMicStepTypeSet | = 1;
i.e., each end micro step type must belong to a state type comprising no other micro step types.

c) $\exists!$ startStateType $\in$ StateTypeSet which is a start state type;
i.e., it exists exactly one start state type that contains the start micro step type.

d) $\forall$ stateType $\in$ StateTypes$_{micProcType}$: micStepType$_i$ $\in$ stateType.sMicStepTypeSet, i=1,2 with
micStepType$_2$ is a successor of micStepType$_1$, $\Rightarrow$
All micro step types on the path from micStepType$_1$ to micStepType$_2$ belong to stateType.sMicStepTypeSet as well;
i.e., the micro step types belonging to the same state type must be inter-connected.

e) $\forall$ micStepType$_i$ = (refType, ValueStepTypes) $\in$ stateType.sMicStepTypeSet, i=1,2:
micStepType$_1$.refType $\neq$ micStepType$_2$.refType;
i.e., different micro step types of a state type must not refer to the same attribute or relation type.

State types are further used to coordinate the execution of individual micro process instances (cf. Chaps. 10 and 12). In particular, state types constitute the interface between micro process and macro process types (i.e., between object behavior and object interactions).

7.3 External Micro Transition Types

We denote a micro transition type that connects micro step types of two different state types as *external micro transition type* (cf. Def. 16).

Example 7.11 (External micro transition type):
In Fig. 7.17, consider the micro transition type connecting micro step types `return date` and `proposal`.

Definition 16 (External micro transition types):
Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet) $\in$ MicProcTypes be a micro process type and let stateType$_i$ $\in$ StateTypeSet, i=1,2 be two state types. Then:

isexternal: MicTransTypes $\mapsto$ BOOLEAN with:

$$\textbf{isexternal(mtt)} := \begin{cases} \textbf{TRUE}, & \exists \text{stateType}_i \in \text{StateTypeSet}_{\text{micProcType}},\ \text{i=1,2,} \\ & \quad \text{with stateType}_1 \neq \text{stateType}_2 \\ & \quad \wedge\ \text{mtt.source} \in \text{stateType}_1\text{.sMicStepTypeSet} \\ & \quad \wedge\ \text{mtt.target} \in \text{stateType}_2\text{.sMicStepTypeSet} \\ \textbf{FALSE}, & \text{else} \end{cases}$$

At run-time, the firing of an external micro transition triggers a subsequent micro state; i.e., the data-driven execution paradigm applies for the activation of states as well.

Example 7.12 (Implicit activation of states):
A `review` reaches state `pending` as soon as the responsible `personnel officer` has set the values of attributes `urgency` and `return date` (cf. Fig. 7.17).

Opposed to a purely data-driven activation, however, some scenarios might require that a responsible user explicitly commits the completion of an activity he or she worked on.

Example 7.13 (User commitment):
Consider state `pending` in Fig. 7.17. An `employee` may re-execute the activity filling in the `review` form until he explicitly commits to submit the `review` to the `personnel officer`.

To capture *user commitments* in a micro process type, we flag external micro transition types either as *implicit* or *explicit* (cf. Def. 17).

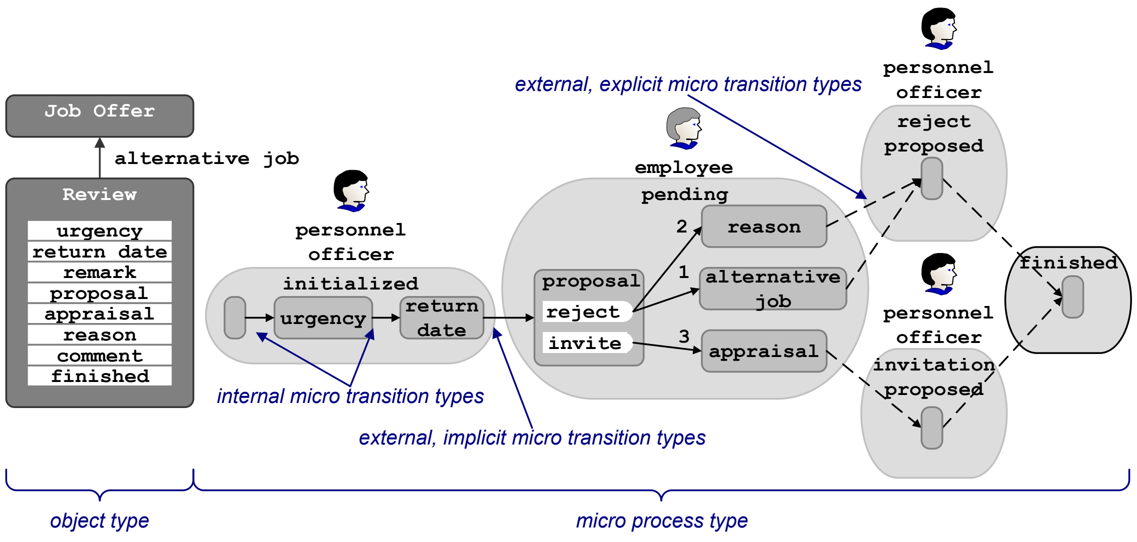


Figure 7.17: Different types of micro transitions

Definition 17 (Explicit micro transition types):
Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet) ∈ MicProcTypes be a micro process type and $\text{micTransType}_{\text{ext}}$:= { t ∈ MicTransType | isexternal(t) = TRUE} be an external micro transition type. Then:

explicit: $\text{MicTransType}_{\text{ext}}$ ↦ BOOLEAN defines for each external micro transition type micTransType, whether it is explicit (i.e., explicit(micTransType) = TRUE) or implicit (i.e., explicit(micTransType) = FALSE).

For correct micro process execution, several structural constraints must be met. Consider Fig. 7.18: several external micro transitions have the same micro step type as source.

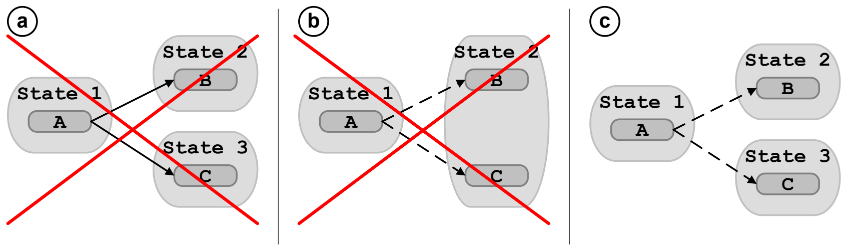


Figure 7.18: Structural properties of external micro transition types

Whether State 2 or State 3 may be activated depends on the values available for attributes B and C. However, a user owns the required write permissions for these attributes only if the subsequent state is already activated. As a consequence, the subsequent state must be activated before the required attribute values and relations can be written. For this purpose, external micro transition types, having the same micro step type as source, must be defined as explicit ones (cf. Def. 18a). As illustrated in Fig. 7.18a, implicit, external micro transition types must not have the same source micro step type. If the source micro step of an external explicit micro transition is reached during run-time, the responsible user must decide which of the subsequent states shall be activated (cf. Fig. 7.18c). This selection can be accomplished by explicitly choosing the desired state from the list of possible states. In this context, we must ensure that the target micro step types of explicit external micro transition types, originating from the same source, belong to different state types (cf. Def. 18b). The situation illustrated in Fig. 7.18b is therefore not allowed.

Definition 18 (Structural properties of external micro transition types):
Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet) $\in$ MicProcTypes be a micro process type. Then:

$\forall$ micTransType$_i$ $\in$ MicTransTypeSet, i=1,2
with micTransType$_1$ $\neq$ micTransType$_2$ $\wedge$ micTransType$_1$.source = micTransType$_2$.source $\wedge$
isexternal(micTransType$_i$) = TRUE, i=1,2:

a) explicit(micTransType$_i$) = TRUE, i=1,2

b) $\exists$ stateType$_i$ = (name$_i$, sMicStepTypeSet$_i$) $\in$ StateTypeSet, i=1,2 with
stateType$_1$ $\neq$ stateType$_2$ $\wedge$ micTransType$_i$.target $\in$ sMicStepTypeSet$_i$, i=1,2

7.4 User Assignment

PHILharmonicFlows allows associating state types and explicit micro transition types with user roles. This way, the actors responsible for executing mandatory activities (i.e., for setting respective attribute values), setting branching decisions, and making commitments (cf. Defs. 19 and 20) can be determined. Note that we consider a user role as entity representing a set of users [FK92, RMR07, RMR09].

Definition 19 (Execution responsibility):
Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet) be a micro process type and let UserRoles be the set of all defined user roles. Then:

An **execution responsibility** is a tuple **execResp = (stateType, role)** where

- stateType $\in$ StateTypeSet is a state type.
- role $\in$ UserRoles is a user role.

Definition 20 (Transition responsibility):
Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet) be a micro process type and let UserRoles be the set of all defined user roles. Then:

A **transition responsibility** is a tuple **transResp = (micTransType, role)** where

- micTransType ∈ MicTransTypeSet is an external, explicit micro transition type requiring a user commitment: isexternal(micTransType) = TRUE ∧ explicit(micTransType) = TRUE.
- role ∈ UserRoles is a user role.

7.5 Backward Jumps

In certain scenarios, it becomes necessary to reset the execution of micro processes by jumping back to a previous state. In PHILharmonicFlows, such backward jumps can be defined using *backward transition types*. As opposed to micro transition types, which connect micro step types, backward transition types always connect two state types with each other.

Example 7.14 (Backward transition types):
As illustrated in Fig. 7.19, if one of the states `reject proposed` or `invitation proposed` is activated, the respective micro process instance may be reset to state `pending`.

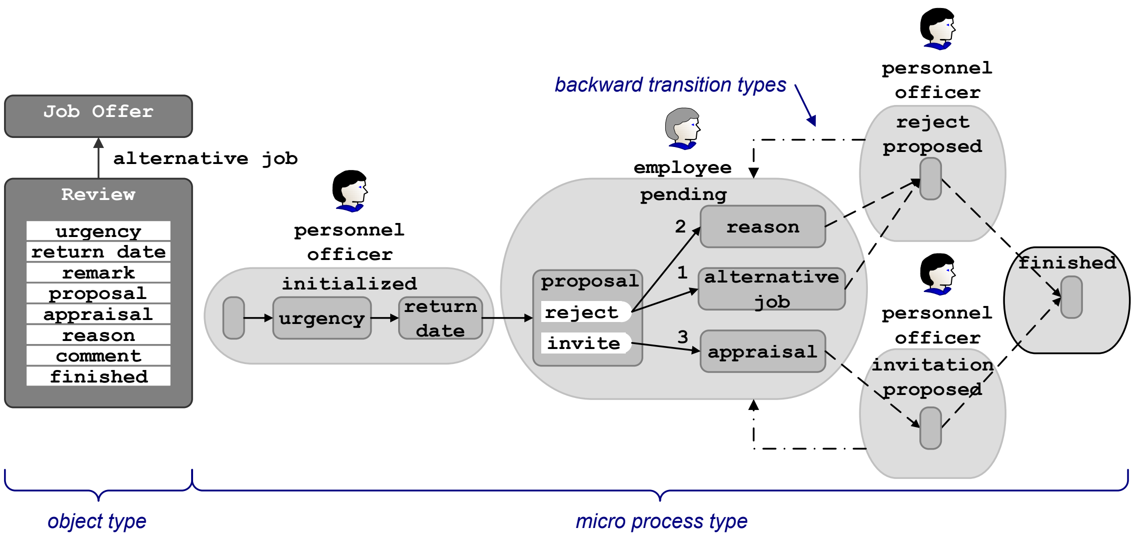


Figure 7.19: Backward transition types

However, for each backward transition type, it must be ensured that its target state type always precedes its source state type (cf. Fig. 7.20a). More precisely, it should not be possible to connect arbitrary state types using backward transitions type; e.g., state types corresponding to different alternative execution paths (cf. Fig. 7.20b).

We define function *beforeStates* to determine all state types lying on a path from one state type to another state type (cf. Def. 21). Using this function we define structural constraints for backward transition types.

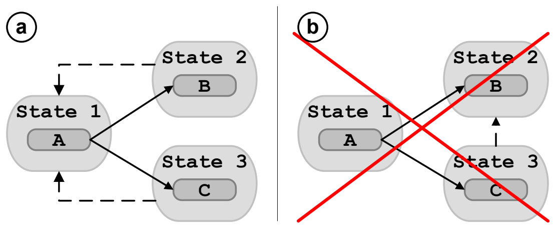


Figure 7.20: Structural properties of backward transition types

Definition 21 (Predecessor states):
Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet) ∈ MicProcTypes be a micro process type. Then:

beforeStates: StateTypeSet $\mapsto$ **$2^{\text{StateTypeSet}}$** with beforeStates(s) determining all state types contained in a path from the start state type to s (including s).

A backward jump is only allowed to state types corresponding to the set *beforeStates* as defined in Def. 22.

Definition 22 (Extending micro process types with backward transition types):
Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet) ∈ MicProcTypes be a micro process type. Then:

A **backward transition type** is a tuple **backTransType = (source, target)** with
source, target ∈ StateTypeSet ∧ source ≠ target ∧ target ∈ beforeStates(source).

$\text{BackTransTypeSet}_{\text{micProcType}}$ denotes a finite set of backward transition types defined on micProcType.
In the following, a micro process type represents a tuple **micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet, BackTransTypeSet)**; i.e., we enrich the definition of a micro process type with backward transition types.

At run-time, before performing a backward jump, it must be ensured that its target state was actually reached earlier. More precisely, if the target state of a backward transition belongs to an alternative path (cf. Fig. 7.21), it must be ensured that this path was actually chosen during run-time. We refer to Chap. 8 for details, which further explains in detail how attribute values are handled in the context of backward jumps (e.g., their resetting and reassignment are discussed).

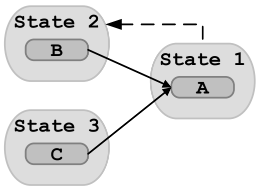


Figure 7.21: Backward transitions and alternative execution paths

Finally, we must define which users may perform a backward jump when the source state of the corresponding backward transition becomes activated. For this purpose, *backward responsibilities* need to be specified (cf. Def. 23).

Definition 23 (Backward responsibility):
Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet, BackTransTypeSet) be a micro process type and let UserRoles be the set of all defined user roles. Then:

A **backward responsibility** is a tuple **backResp = (backTransType, role)** where

- backTransType $\in$ BackTransTypeSet is a backward transition type.
- role $\in$ UserRoles is a user role.

7.6 Reducing Administrative Efforts

Usually, dozens or hundreds of different object types may be involved in a more complex business process. For each of these object types, then, a corresponding micro process type must be defined. In addition, user roles must be assigned to states, external transitions, and backward transitions. Overall, this may lead to high administrative efforts. To remedy these, PHILharmonicFlows, for each defined object type, automatically generates a "minimum" micro process type. Furthermore, default assignments for external micro transitions and backward transitions are used to reduce modeling efforts.

7.6.1 Minimal Micro Process Types

PHILharmonicFlows automatically generates a *minimum micro process type* for each object type. As illustrated in Fig. 7.22, such a minimum micro process type comprises a start and an end micro step type, which are assigned to respective start and end state types. Both the start and end micro step type are empty requiring no attribute value or relation to be set. These two micro step types are connected using an (external) explicit micro transition type. This way, an explicit user commitment is required to activate the end state and therefore to terminate a running micro process instance. Altogether, a minimum micro process type does not require any mandatory attribute values or relations. However, it fulfills all structural properties required. Although it is automatically generated for each object type, it may be refined and modified as desired.

7.6.2 Default User Assignment

To further reduce administrative efforts, each user role owing the execution responsibility for a particular state type automatically obtains the responsibilities for committing outgoing (external) explicit micro transition types as well as for outgoing backward jumps.

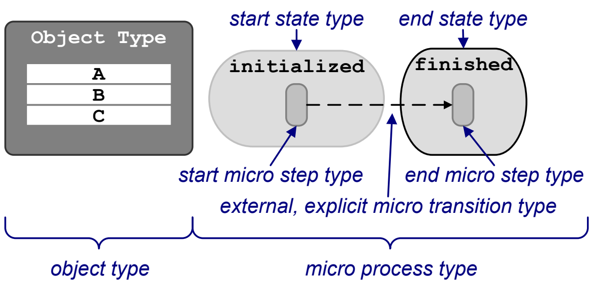


Figure 7.22: Minimum micro process type

Example 7.15 (Default user assignment):
Consider the `review` micro process type from Fig. 7.23. Since user role `employee` is assigned to state `pending` (i.e., user role `employee` owns the execution responsibility for state `pending`), this user role automatically owns the transition responsibility for the explicit micro transition type connecting micro step type `reason` and the empty micro step type belonging to state type `reject proposed`. The same applies to all other explicit micro transition types whose source micro step type belongs to state `pending`; i.e., the ones starting from micro steps `alternative job` or `appraisal`. Regarding state type `reject proposed`, user role `personnel officer` is assigned; i.e., it then owns the corresponding execution responsibility. Hence, the `personnel officer` automatically owns the transition responsibility for the micro transition type connecting the empty micro step type of state type `reject proposed` and the end micro step type of state type `finished`. Moreover, this user additionally owns the backward responsibility for the backward transition type connecting state types `reject proposed` and `pending`. Note that these default assignments can be manually overwritten afterwards.

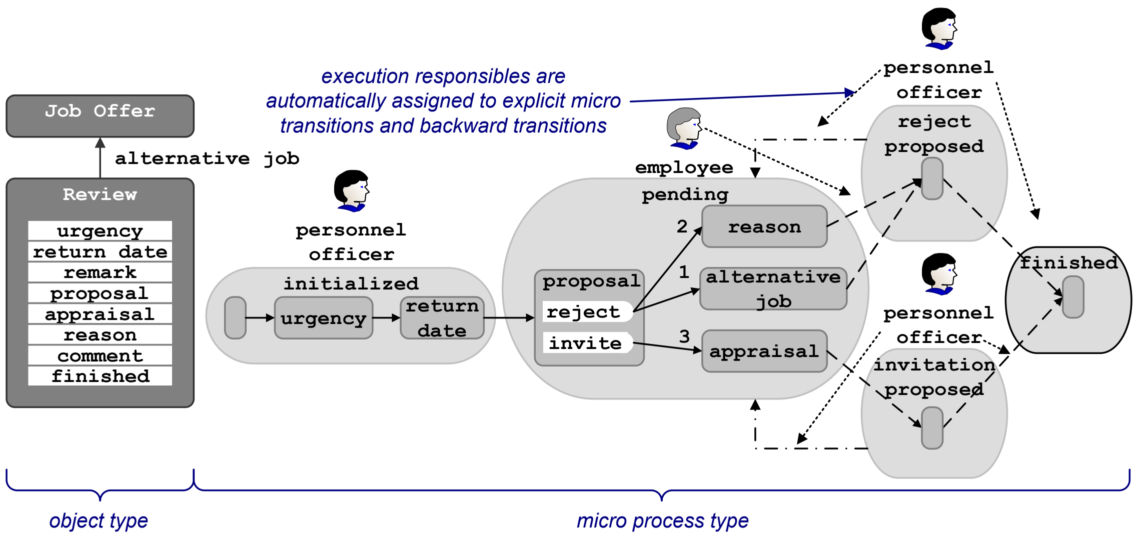


Figure 7.23: Default user assignment

7.7 Summary

To capture and model the behavior of object types, corresponding micro process types must be defined. More precisely, for each object type a micro process type needs to be defined. It comprises a number of abstract state types which are then used to coordinate the execution of corresponding micro process instances among different users.
As opposed to existing state-based approaches, PHILharmonicFlows allows establishing a mapping between states and object attribute values and hence ensuring compliance between them. For this purpose, each state comprises a number of micro steps. In turn, a micro step refers to an attribute or relation type and describes an elementary action for setting them. By connecting micro steps with micro transitions, their default execution order is obtained. To enable user decisions, micro transition types may be declared as explicit; i.e., at run-time, the commitment of a user is required to proceed with the control flow. Finally, based on backward transition types, backward jumps to previous states can be realized.

8 Micro Process Execution

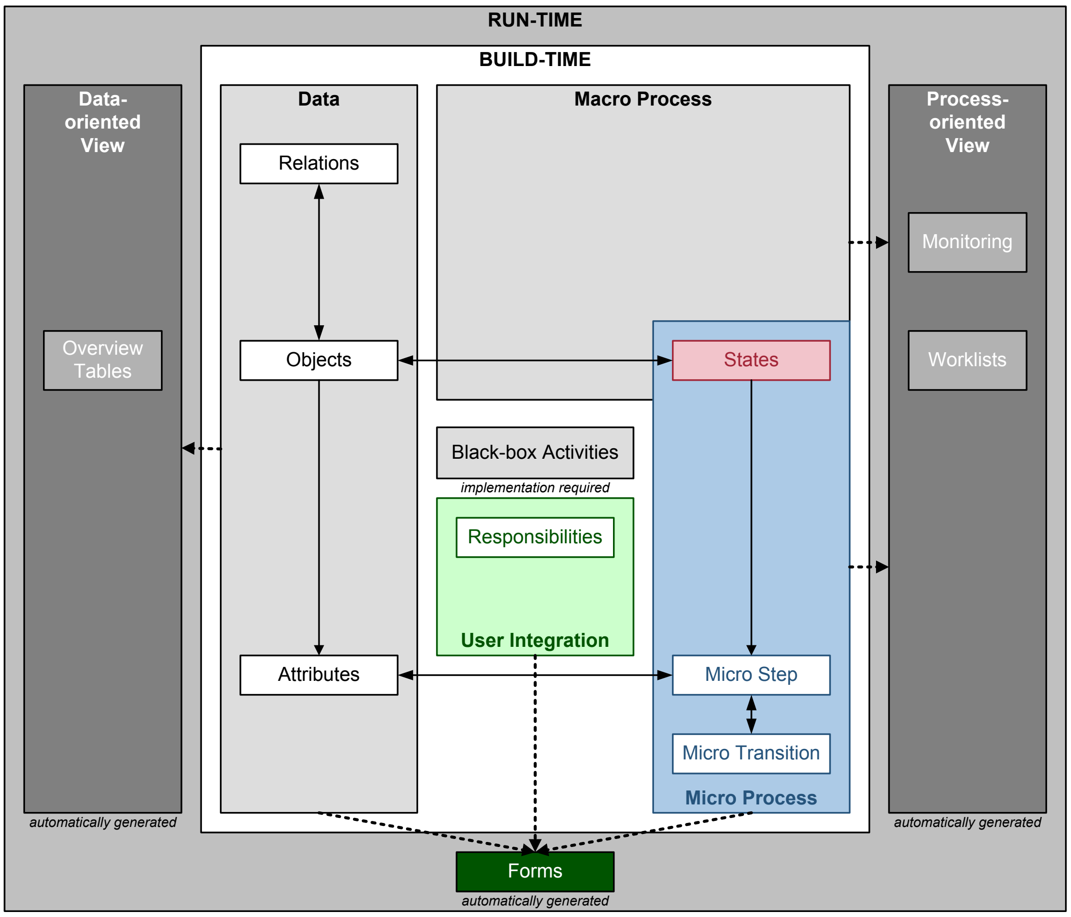


Figure 8.1: Micro process execution in PHILharmonicFlows

In addition to the structural correctness of a micro process type, the correct execution of its corresponding micro process instances is crucial. In particular, this necessitates a well-defined formal semantics, which not only provides the basis for creating and executing micro process instances, but shall enable the automatic generation of end-user run-time components as well, e.g., for running micro process instances, process-oriented views and form-based activities shall be automatically generated. To properly execute a micro process instance, its execution follows a precise operational semantics, which is based on the attribute values of the corresponding object instance. Like in activity-centric approaches [RD98, LR00], we introduce a number of markings for the different components of a micro process instance. The *formal operational semantics* for executing micro process instances is then defined based on these *markings*. More precisely, we define rules that control the interactions among the different components of a micro process instance during its execution. How these interactions are taking place depends on various conditions. In particular, while some rules concern the dependencies between different components, others require user inputs. Finally, to improve the comprehensibility of the rules, PHILharmonicFlows categorizes them into marking rules (highlighted in green colour in the following), execution rules (yellow colour), and reaction rules (blue colour) (cf. Tab. 8.1). Fig. 8.2 illustrates the three different rules and their inter-dependencies.

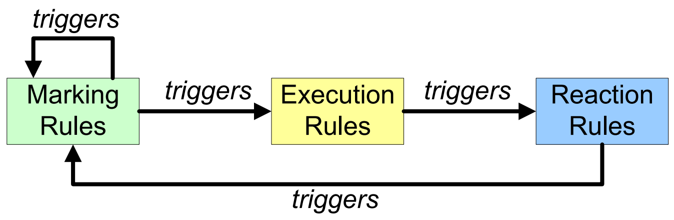


Figure 8.2: Rules and their inter-dependencies

	Abbreviation	Input	Output	Colour
Marking Rules	MR	markings	markings	green
Execution Rules	ER	markings	required user input	yellow
Reaction Rules	RR	user input	markings	blue

Table 8.1: Different micro process rules

- **Marking rules (MR)**. Marking rules change markings of particular components of a micro process instance taking the current markings of other components into account; i.e., the current markings of the individual components of a micro process instance are evaluated to determine follow-up-markings. For example, when a micro step changes its marking from ACTIVATED to UNCONFIRMED, all outgoing implicit micro transitions will be marked as READY.

- **Execution rules (ER)**. At certain points during micro process execution, input from the user is required. Such user inputs comprise the setting of attribute values or the commitment of state changes. Using execution rules, we can exactly specify under what conditions (i.e., markings) respective user inputs are mandatorily required. For example, when a micro step becomes marked as ENABLED, a corresponding attribute value (or relation) must be set. For this purpose, a corresponding form-based mandatory activity is automatically generated and assigned to the worklist of the responsible user.

- **Reaction rules (RR).** Opposed to execution rules, reaction rules determine which markings must be changed when required user input becomes available. For example, if a required attribute value is available, the corresponding micro step changes its marking from ENABLED to ACTIVATED.

Due to explicit consideration of data during process execution, the operational semantics provided by PHILharmonicFlows is much more complex as the one known from activity-centric process support paradigms (e.g., like Petri Nets). In particular, when executing object-aware processes, on one hand, we must support *cross-state process execution* to coordinate different users and synchronize various process instances. On the other, we must capture *state-internal execution logic* (i.e., dependencies between input fields and user guidance). Furthermore, non-determinism and backward jumps need to be considered. For the sake of understandability, we introduce the required rules step-by-step overwriting them where necessary.

8.1 Micro Process Instances

To enable object-aware process support at run-time, the creation of an object instance must be coupled with the one of a corresponding micro process instance. Consequently, each micro process type may be instantiated multiple times; i.e., for each micro process type a number of corresponding micro process instances may exist. According to the definition of a micro process type, a corresponding micro process instance may comprise several micro steps, micro transitions, states, and backward transitions. Further, it is related to exactly one object instance.[1] The execution of micro process instances is then based on different kinds of markings similar to Workflow Nets or Petri Nets [RMRD04] (cf. Def. 24); i.e., the processing state of a micro process instance is defined by the current markings of its states, micro steps, micro transitions, and backward transitions (cf. Fig. 8.3). Furthermore, a micro process instance itself has an associated marking expressing its overall state. Based on these markings, it can be expressed which components (e.g., micro steps) are activated at a certain point in time. Markings are further used to describe which components may be activated later on as well as for which components this is no longer possible (i.e., these components belong to a skipped execution path). Respective information is not only important to identify the actual execution path, but is also essential to identify deadlock situations or reset the execution in the context of backward jumps. For this purpose, at each point during process execution, all components of the micro process instance are associated with a marking; i.e., not only the components currently activated. In the following, we describe the markings required for a particular component to allow for the correct execution of the respective micro process instance.

Example 8.1 (Markings of micro process instances):
Consider the `review` micro process instance illustrated in Fig. 8.3. Currently, this instance is running; i.e., it is marked as RUNNING. State `initialized` has been already finished (marking CONFIRMED). Thus, attribute values corresponding to the particular micro steps of this state are available (marking CONFIRMED). In turn, state `pending` is still activated (i.e., marked as ACTIVATED). In this context, a value for attribute `proposal` is mandatorily required in order to proceed with process execution (marking ENABLED of micro step `proposal`). Opposed to this, micro steps corresponding to attributes (relations) `reason`, `alternative job`, and `appraisal` are currently marked as READY

[1]Note that we omit the qualifier type of the micro process components to improve understandability and to indicate that we are talking about the process instance level. In particular, we consider micro steps, micro transitions, states and loop transitions. Nevertheless, each of these components has a corresponding type.

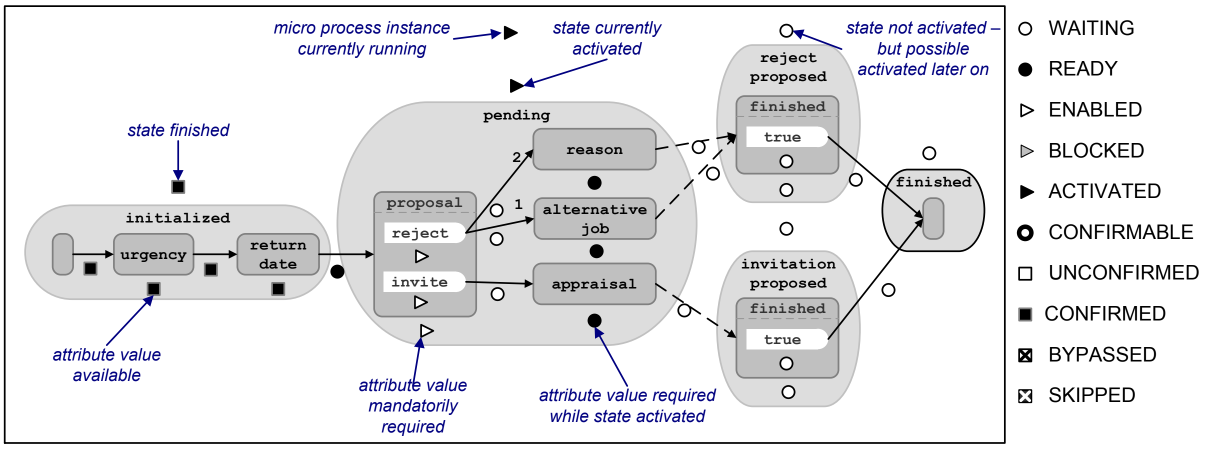


Figure 8.3: Markings for micro process execution

indicating that a respective value is required when state `pending` becomes activated. All subsequent states (e.g., `reject proposed`) are currently marked as WAITING indicating that they may be activated later on.

Definition 24 (Micro process instance):
A **micro process instance** is a tuple **micProcInst = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$)** where

- **micProc** = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet) is a micro process type.
- **oid** is the identifier of the object instance, micProc belongs to.
- **M_{State}: StateSet $\mapsto$ StateMarkings** assigns to each state s of micProc its current marking $M_{State}(s) \in$ **StateMarkings := {WAITING, ACTIVATED, CONFIRMED, SKIPPED}**. The semantics of these markings is described in Tab. 8.2.
- **$M_{MicStep}$: MicStepSet $\mapsto$ MicroStepMarkings** assigns to each micro step micStep (and to each value step valStep) of micProc its current marking $M_{MicStep}$(micStep) $\in$ MicroStepMarkings (and $M_{MicStep}$(valStep) $\in$ MicroStepMarkings) with **MicroStepMarkings := {WAITING, READY, ENABLED, BLOCKED, ACTIVATED, UNCONFIRMED, CONFIRMED, BYPASSED, SKIPPED}**. The semantics of these markings is described in Tab. 8.3.
- **$M_{MicTrans}$: MicTransSet $\mapsto$ MicroTransitionMarkings** assigns to each micro transition micTrans of micProc its current marking $M_{MicTrans}$(micTrans) $\in$ **MicroTransitionMarkings := {WAITING, CONFIRMABLE, READY, ENABLED, ACTIVATED, UNCONFIRMED, CONFIRMED, BYPASSED, SKIPPED}**. The semantics of these markings is described in Tab. 8.4.
- **$M_{BackTrans}$: BackTransSet $\mapsto$ BackwardTransitionMarkings** assigns to each backward transition backTrans of micProc its current marking $M_{BackTrans}$(backTrans) $\in$ **BackwardTransitionMarkings := {WAITING, CONFIRMABLE, READY, BLOCKED, SKIPPED}**. The semantics of these markings is described in Tab. 8.5.

MicProcInstances denotes the set of all micro process instances of any micro process type micProcType $\in$ MicProcTypes. Further, **$M_{MicProc}$: MicProcInstances $\mapsto$ MicroProcessMarkings** assigns to each micro process instance micProcInst its current marking $M_{MicProc}$(micProcInst) $\in$ **MicroProcessMarkings := {RUNNING, FINISHED}**. The semantics of these markings is described in Tab. 8.6.
Finallly, **micprocinstances: MicProcTypes $\mapsto$ $2^{\text{MicProcInstances}}$** assigns to each micro process type micProcType the set of corresponding micro process instances; i.e., micprocinstances(micProcType) $\subseteq$ MicProcInstances running on micProc.

8.1.1 State Markings

Concerning the states of a micro process instance, the following information is relevant:

- Which state is currently activated?
- Which states may be activated later on?
- Which states were already activated during process execution?
- Which states were skipped due to the selection of an alternative path?

For this purpose, each state is either marked as WAITING, ACTIVATED, CONFIRMED or SKIPPED (cf. Def. 24). These markings have the following meanings 8.2:

Marking	**Label**	**Description**
○	WAITING	The state has not been activated yet, but it is still possible that it becomes activated later; i.e., a preceding state is currently activated.
▶	ACTIVATED	The state is currently activated; i.e., attribute values corresponding to the micro steps of this state must be mandatorily set.
■	CONFIRMED	The state was previously activated and a subsequent state is activated now.
☒	SKIPPED	The state has not been activated yet and will also not become activated anymore (unless backward jumps are performed); i.e., the respective state belongs to an alternative execution path not chosen for execution.

Table 8.2: State markings

For each micro process instance, exactly one of its states is activated at a certain point in time. In turn, this indicates that responsible users must assign attribute values or relations, which are referred by the micro steps of this state.

Initially, the start state of a micro process instance is marked as ACTIVATED. In turn, all other states are initially marked as WAITING. If a state becomes marked as ACTIVATED, responsible users must set the attribute values or relations corresponding to the micro steps of this state; i.e., they must execute a mandatory (form-based) activity (cf. Sect. 8.4). If all required attribute values (or relations) are available for the currently activated state, the latter may change its marking from ACTIVATED to CONFIRMED. Following this, another subsequent state becomes activated (cf. Sect. 8.5). If an end state becomes activated, in turn, the micro process instance will terminate.

For coordinating the execution of several micro process instances (i.e., to evaluate the coordination components), it is important to know whether or not a particular state can still be activated; i.e., additional information about the states currently not activated needs to be maintained as well. For this purpose, we apply an *external dead-path elimination* to prevent those states from becoming activated, which belong to skipped paths. As a consequence, respective states are marked as SKIPPED (cf. Sect. 8.5.4).

Fig. 8.4 illustrates the markings, a state may pass as well as their transitions:

1. If a state change takes place, the previous state (currently marked as ACTIVATED) is re-marked as CONFIRMED. In turn, the subsequent state, which is currently marked as WAITING, then changes its marking to ACTIVATED.

2. States that cannot be activated any longer are marked as SKIPPED. To determine these states during run-time, dead paths are eliminated (cf. Sect. 8.5.4).

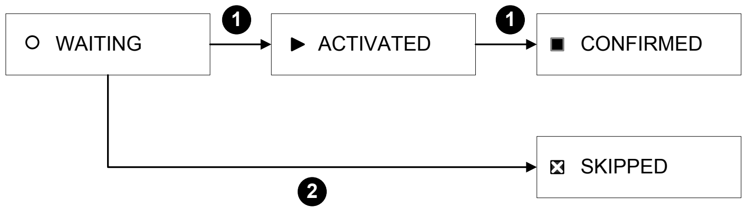


Figure 8.4: State markings and their transitions

8.1.2 Micro Step and Value Step Markings

Different markings are required to indicate the processing state of the micro steps of a micro process instance. In particular, each micro step (or value step) is either marked as WAITING, READY, ENABLED, BLOCKED, ACTIVATED, UNCONFIRMED, CONFIRMED, BYPASSED, or SKIPPED (cf. Def. 24).

As described in Chapt. 7, each state may comprise several micro steps. In turn, a micro step referring to a particular object attribute may additionally comprise a number of value steps. The latter can be reached if the corresponding predicate evaluates to true. First of all, the markings of micro steps (and value steps) depend on the marking of the state they belong to (cf. Fig. 8.5). Especially, all micro steps belonging to a WAITING state (i.e., that still may be activated) are initially marked as WAITING.
A single micro step does do not represent an activity, but corresponds to a particular input field of a user form. More precisely, for each user assigned to the respective state, a form-based activity is generated. For each micro step of the currently activated state, this form then comprises an input field related to the corresponding attribute and relation respectively (cf. Sect. 8.4). When also taking data authorization into account, input fields that may be optionally set can be added to the respective user form (cf. Sect. 9.1.1). To distinguish such optional input fields from mandatory ones, the input fields corresponding to the micro steps of the activated state are marked as READY.
The micro transitions connecting the micro steps of a state represent the internal logic of the corresponding form-based activity. During activity execution, it should be transparent for users which input field must be mandatorily set in order to reach a subsequent micro step. Respective micro steps are marked as ENABLED indicating that a value for the attribute (or relation) they refer to is missing. A micro step will be immediately reached as soon as a value for its corresponding attribute or relation becomes available (or no attribute or relation is available).
Generally, it must be ensured that always one micro step is activated during activity execution. As it is possible that one micro step (or value step) has more than one outgoing micro transition, only the one with the highest priority is reached at run-time and hence it is marked as UNCONFIRMED. All other ones not belonging to the selected execution path are marked as BYPASSED.
In the context of a value-specific micro step which comprises a number of values steps, it might occur that for a particular attribute value none of the corresponding value steps becomes activated; i.e., none of the predicates evaluates to true. If the value-specific micro step itself has no outgoing micro transition, process execution will then be blocked; i.e., the value-specific micro

step will be marked as BLOCKED indicating that another value ir required for the attribute the micro step refers to.
When a subsequent state becomes activated, all micro steps of the preceding state which are currently marked as UNCONFIRMED, change their marking to CONFIRMED. In order to enable users to change already assigned attributes values or relations later on (as long as the state to which they belong is activated), micro steps belonging to a skipped path are first marked as BYPASSED. These micro steps change their marking to SKIPPED when the state is marked as CONFIRMED. This way, alternative execution paths may be changed later on requiring a state-internal reset of micro process instance execution.

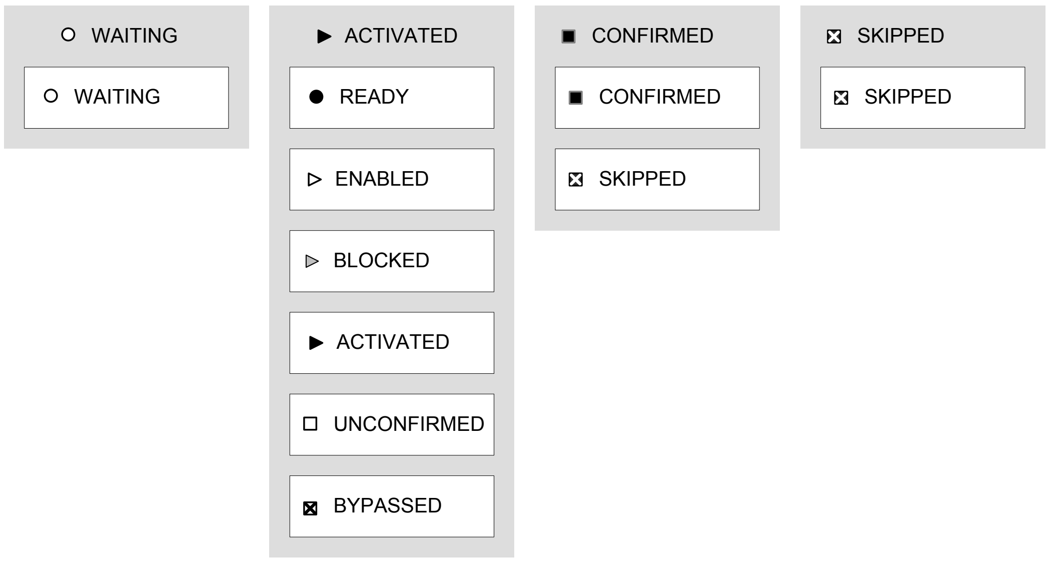


Figure 8.5: Dependencies between state markings and micro step markings

Tab. 8.3 summarizes the possible markings for the micro steps (and value steps) of a micro process instance:

Marking	Label	Description
○	WAITING	The micro step belongs to a state that has not been activated yet; i.e., the state to which the micro step belongs is currently marked as WAITING. Accordingly, the micro step is currently not reachable, but may be reached later when the state it belongs to becomes activated. When marking a micro step as WAITING, all corresponding value steps will be marked as WAITING as well.
●	READY	The micro step belongs to the currently activated state, which is therefore marked as ACTIVATED. Thus, it is possible to activate the micro step. All micro steps and micro transitions belonging to a particular state, however, represent a sub-graph of the micro process instance and hence imply an order for the micro steps of a state. For this reason, usually one micro step (or several ones in case of alternative execution paths) is mandatorily required next. This micro step becomes marked as ENABLED. Opposed to this, if there still exists at least one previous micro step belonging to the sub-graph of the state, the micro step is marked as READY indicating that another attribute value (or relation) is required before. When a micro step becomes marked as READY, its corresponding value steps will be marked as READY as well.

Marking	**Label**	**Description**
▷	ENABLED	The micro step belongs to the currently activated state; i.e., the state the micro step belongs to is marked as ACTIVATED. For an enabled micro step, the corresponding attribute (or relation) must be set in order to proceed with the flow of control. Furthermore, if a micro step is marked as ENABLED, its value steps will be marked as ENABLED as well. Value step predicates can only be evaluated when an attribute value (or relation) becomes available for the respective micro step.
▶	ACTIVATED	The micro step belongs to the currently activated state. To mark a particular micro step as ACTIVATED, we consider the following cases: • An atomic micro step is marked as ACTIVATED when a value for its corresponding attribute (or relation) becomes available. • An empty micro step not referring to any attribute (or relation) immediately becomes marked as ACTIVATED. • A value step becomes marked as ACTIVATED when its corresponding predicate evaluates to true. • Value-specific micro steps are marked as ACTIVATED if at least one of its value steps is marked as ACTIVATED or there exists an outgoing micro transition of the value-specific micro step itself, enabling the micro process instance to proceed even if no value step is activated.
▷	BLOCKED	The (value-specific) micro step belongs to the currently activated state. A value-specific micro step is marked as BLOCKED if none of its value steps is marked as ACTIVATED and no outgoing micro transition from the value-specific micro step itself exists. This indicates that another attribute value (or relation) must be assigned for the corresponding attribute or relation of the micro step to proceed with micro process execution.
☐	UNCONFIRMED	The micro step belongs to the currently activated state. The micro step was reached during micro process execution. More precisely, a value of the corresponding attribute (or relation) is available and one of its incoming micro transitions has been triggered.
■	CONFIRMED	The state to which the micro step belongs has changed its marking from ACTIVATED to CONFIRMED. Then, all micro steps currently marked as UNCONFIRMED change their marking to CONFIRMED; i.e., the micro step is reached and hence a value of the corresponding attribute (or relation) is available. In addition, the path to which the micro steps belong was skipped. In turn, a value step is marked as CONFIRMED if the corresponding predicate is fulfilled and the value step does not belong to a skipped path (i.e., one of its outgoing micro transitions has fired).
☒	BYPASSED	All micro steps of the currently activated state, which have not been reached yet, are marked as BYPASSED. A value step is marked as BYPASSED if its predicate evaluates to false. Since it is possible to change attribute values (or relations) as long as the respective state is activated, these micro steps might still be activated later. For this case, however, a reset and re-execution of affected parts of the micro process instance are required.
☒	SKIPPED	All micro steps not yet reached are marked as SKIPPED. Respective micro steps belong to a state that either is marked as SKIPPED or CONFIRMED.

Table 8.3: Micro step markings

Fig. 8.6 illustrates the markings of a micro step (or value step) and their possible transitions:

1. A micro step changes its marking from WAITING to READY when the state it belongs to becomes marked as ACTIVATED.
2. A micro step changes its marking from READY to ENABLED when it becomes reachable; i.e., the previous micro step is marked as CONFIRMED or UNCONFIRMED.
3. A micro step changes its marking from ENABLED to ACTIVATED when a value for its corresponding attribute (or relation) becomes available (or the micro step is an empty one). Concerning a value-specific micro step, at least one of its value steps must then be marked as ACTIVATED or there is a micro transition outgoing from the micro step itself.
4. A value-specific micro step changes its marking from ENABLED to BLOCKED when a value for its corresponding attribute (or relation) becomes available, but none of its value steps is marked as ACTIVATED (i.e., the predicates of all value steps evaluate to false) and there is no micro transition outgoing from the micro step itself.
5. A value-specific micro step changes its marking from BLOCKED to ENABLED when the value of the corresponding attribute (or relation) is deleted.
6. A micro step changes its marking from ACTIVATED to UNCONFIRMED if one of the following two cases holds:
   a) The respective micro step is the only one currently marked as ACTIVATED.
   b) There are other micro steps currently marked as ACTIVATED. In this case, only the one with the highest priority (of its incoming micro transition) is marked as UNCONFIRMED. (All other ones are marked as BYPASSED.)
7. A micro step changes its marking from ACTIVATED to BYPASSED if the priority of its incoming micro transition is lower than the priority of another activated micro step.
8. If a subsequent state becomes activated, all micro steps belonging to the previous state and currently marked as UNCONFIRMED are re-marked as CONFIRMED.
9. If a subsequent state becomes activated, all micro steps belonging to the previous state and currently marked as BYPASSED are re-marked as SKIPPED.
10. A micro step changes its marking from WAITING to SKIPPED due to an external dead-path elimination.
11. A micro step changes its marking from ENABLED to BYPASSED when a subsequent micro step becomes marked as UNCONFIRMED.
12. A micro step changes its marking from BLOCKED to BYPASSED when an alternative subsequent micro step becomes marked as UNCONFIRMED.
13. When applying a state-internal dead-path elimination, the micro steps belonging to an alternative execution path are marked as BYPASSED.

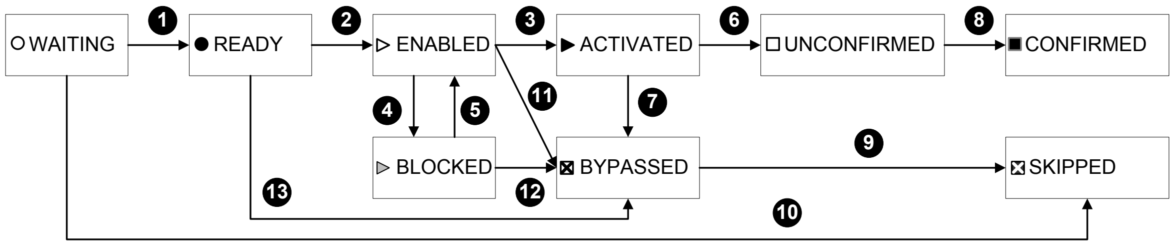


Figure 8.6: Micro step markings and their transitions

8.1.3 Micro Transitions Markings

Micro transitions connect micro steps with each other. When a micro process instance is created, all micro transitions are initially marked as WAITING. In this context, *internal micro transitions*, connecting micro steps of the same state, represent the internal process logic of a *form-based activity*; i.e., they describe the default order in which required attribute values (or relations) shall be set. In turn, *external micro transitions* activate subsequent states; i.e., they coordinate the processing of an object instance among different users. Further, external micro transitions may be flagged as explicit. Such an *explicit micro transition* requires a user commitment before a subsequent state may be activated. To express this behavior, we introduce marking CONFIRMABLE. Based on it, a respective mandatory activity is automatically assigned to the worklist of the responsible user who must perform the commitment. If a state change is committed, the micro transition will be marked as READY.
Opposed to this, *implicit micro transitions* are immediately marked as READY (once they are reachable) indicating that the subsequent micro step is reachable. In addition, for *external implicit micro transitions*, a state change (i.e., the activation of a subsequent state) is then initiated. If attribute values (or relations) for subsequent micro steps are available (i.e., the micro steps are marked as ACTIVATED), micro transitions are marked as ENABLED. Generally, it is possible that more than one subsequent micro step is reachable; i.e., all these micro steps are marked as ACTIVATED. For all micro transitions currently marked as ENABLED, their respective *priorities* must be evaluated. Only the micro transition with the highest priority is marked as ACTIVATED enabling the subsequent micro step to change its marking from ACTIVATED to UNCONFIRMED. Like for micro steps, we differentiate between UNCONFIRMED and CONFIRMED as well as between BYPASSED and SKIPPED, enabling the correct re-assignment of attribute values and relations during the activation of the respective state.

In summary, each micro transition has one of the following markings: WAITING, CONFIRMABLE, READY, ENABLED, ACTIVATED, UNCONFIRMED, CONFIRMED, BYPASSED, or SKIPPED (cf. Tab. 8.4).

Marking	Label	Description
○	WAITING	When creating a micro process instance, its micro transitions are initially marked as WAITING. A micro transition marked as WAITING is currently not reachable. In particular, its source micro step has not been reached yet; i.e., it has not been marked as UNCONFIRMED yet.
◉	CONFIRMABLE	A micro transition is marked as CONFIRMABLE when it becomes reachable, but a user commitment is required to proceed with control flow (i.e., to mark the micro transition as READY).

●	READY	A micro transition is marked as READY when it becomes reachable; i.e., its source micro step is reached and either marked as UNCONFIRMED (internal micro transition) or CONFIRMED (external micro transition). If the micro transition is explicit, a user commitment has been made.
▷	ENABLED	The target micro step is marked as ACTIVATED; i.e., the required attribute value (or relation) is available. Since more than one micro step may be marked as ACTIVATED at a certain point during process execution, more than one micro transition may be marked as ENABLED. For respective micro transitions, their priorities must then be evaluated.
▶	ACTIVATED	The micro transition may fire; i.e., either only one micro transition has been marked as ENABLED or it has the highest priority of all micro transitions currently marked as ENABLED.
□	UNCONFIRMED	The micro transition has fired; i.e., the target micro step is reached and marked as UNCONFIRMED.
■	CONFIRMED	The micro transition has fired and a subsequent state has already been activated.
☒	BYPASSED	The micro transition has not fired yet. Either the subsequent micro step was not reachable (since the required attribute value or relation is missing) or its priority was too low.
☒	SKIPPED	The micro transition has not fired and a subsequent state has already been activated.

Table 8.4: Micro transition markings

Fig. 8.7 illustrates the different markings of a micro transition and their possible transitions:

1. An implicit micro transition changes its marking from WAITING to READY when reaching its source micro step; i.e., its source micro step is marked as UNCONFIRMED.
2. An explicit micro transition changes its marking from WAITING to CONFIRMABLE when reaching its source micro step; i.e., the source micro step is marked as UNCONFIRMED.
3. An explicit micro transition changes its marking from CONFIRMABLE to READY, when the required user commitment is made.
4. A micro transition changes its marking from READY to ENABLED as soon as its target micro step becomes reachable; i.e., the required attribute value or relation to which the target micro step refers becomes available and the target micro step is marked as ACTIVATED.
5. A micro transition changes its marking from ENABLED to ACTIVATED either if exactly one micro transition is marked as ENABLED or the micro transition has the highest priority of all micro transitions currently marked as ENABLED.
6. An internal micro transition changes its marking from ACTIVATED to UNCONFIRMED when reaching its target micro step; i.e., its target micro step becomes marked as UNCONFIRMED.
7. An external micro transition changes its marking from ACTIVATED to UNCONFIRMED when reaching its target micro step; i.e., its target micro step is marked as UNCONFIRMED.
8. To mark states belonging to a skipped execution path of the current state, a state-internal dead-path elimination is applied. Based on it, internal micro transitions, belonging to skipped paths, change their marking from WAITING to BYPASSED.

9. An internal micro transition changes its marking from ENABLED to BYPASSED if the micro transition has not the highest priority of all micro transition currently marked as ENABLED.
10. An external micro transition changes its marking from WAITING to SKIPPED if another micro transition is either used to activate a subsequent state or a state-external dead-path elimination has been applied.
11. An external micro transition changes its marking from ACTIVATED to CONFIRMED when reaching its target micro step; i.e., its target micro step is marked as UNCONFIRMED.
12. If a subsequent state is activated, all micro transitions belonging to the previous state and currently marked as UNCONFIRMED change their marking to CONFIRMED.
13. If a subsequent state is activated, all micro transitions belonging to the previous state and currently marked as BYPASSED change their marking to SKIPPED.

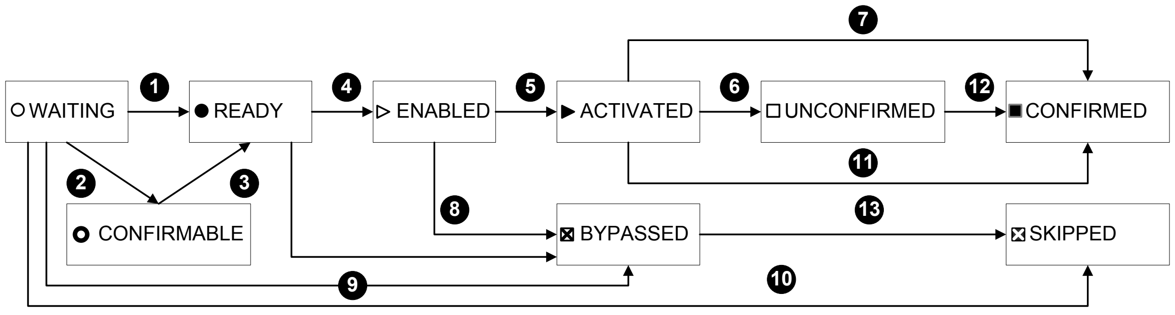


Figure 8.7: Markings of a micro transition and their transitions

8.1.4 Backward Transition Markings

Backward transitions between the states of a micro process instance describe backward jumps. Initially, they are marked as WAITING. A backward jump is enabled if the source state of the backward transition is currently marked as ACTIVATED. It then is marked as CONFIRMABLE; i.e., any backward transition is treated as explicit transition and hence the respective backward jump requires a commitment of an authorized user. As soon as such a commitment is made, the backward transition is marked as READY. Following this, the state change may be performed; i.e., the micro process instance is reset to a previous state. Note that a backward transition will be only enabled (i.e., marked as CONFIRMABLE), if its target state was reached before; i.e., its target state must not belong to an alternative path not chosen during micro process execution. In the latter case, the backward transition is marked as BLOCKED and the backward jump must not be performed. Since a particular backward jump may be performed several times during the execution of a micro process instance, a backward transition is re-marked as WAITING once it has been fired. Finally, an uncommitted backward transition is marked as SKIPPED, when a subsequent state becomes activated during micro process execution.

Any backward transition has one of the following markings: WAITING, CONFIRMABLE, READY, BLOCKED, CONFIRMED, or SKIPPED (cf. Tab. 8.5).

Marking	Label	Description
○	WAITING	A backward transition is initially marked as WAITING indicating that its source state has not been activated yet; i.e., its source state is not marked as ACTIVATED and the backward jump must not be performed.
◉	CONFIRMABLE	To perform the backward jump, a user commitment is required. Afterwards, the backward jump can be performed; i.e., the source state of the corresponding backward transition is then activated (i.e., marked as ACTIVATED).
●	READY	The source state of the backward transition is activated (i.e., marked as ACTIVATED) and the required user commitment has been made. This marking then triggers the backward jump; i.e., the resetting of the micro process instance to the target state of the backward transition.
▷	BLOCKED	The backward jump must not be performed since the target state of the backward transition belongs to a skipped path; i.e., the target state is marked as SKIPPED.
⊠	SKIPPED	The backward jump was not performed; i.e., the micro process execution proceeds without this backward jump.

Table 8.5: Backward transition markings

Fig. 8.8 illustrates the markings of a backward transition and their possible transitions:

1. A backward transition changes its marking from WAITING to CONFIRMABLE when the source state of the backward transition becomes activated.
2. A backward transition changes its marking from CONFIRMABLE to READY when the required user commitment becomes available.
3. A backward transition changes its marking from READY to WAITING after performing the backward jump.
4. A backward transition changes its marking from CONFIRMABLE / BLOCKED to SKIPPED when the micro process execution proceeds without having performed any backward jump.
5. A backward transition changes its marking from WAITING to BLOCKED when the source state of the backward transition becomes activated (i.e., marked as ACTIVATED). However, the target state of the backward transition belongs to a non-selected execution path; i.e., it was skipped during micro process execution.

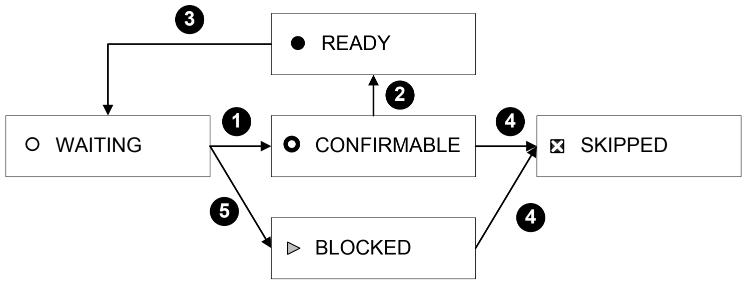


Figure 8.8: Markings of a backward transition and their transitions

Marking	Label	Description
▶	RUNNING	The micro process instance was started and no end state has been activated yet (i.e., the currently activated state is not an end state).
■	FINISHED	An end state of the micro process instance has been activated (i.e., it is marked as ACTIVATED).

Table 8.6: Micro process instance markings

8.1.5 Micro Process Instance Markings

The processing state of a micro process instance can be defined based on the markings of its states, micro steps, micro transitions, and backward transitions. In addition, each micro process instance has its own marking indicating whether an instance is INITIALIZED, RUNNING, or FINISHED (cf. Tab. 8.6).

Fig. 8.9 illustrates the markings of a micro process instance and their possible transitions:

1. A micro process instance changes its marking from RUNNING to FINISHED when an end state of the micro process instance becomes activated.
2. When using backward jumps, a micro process instance may be re-activated; i.e., it is possible to jump from an end state to a previous state.

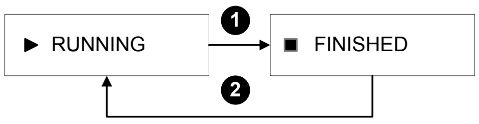
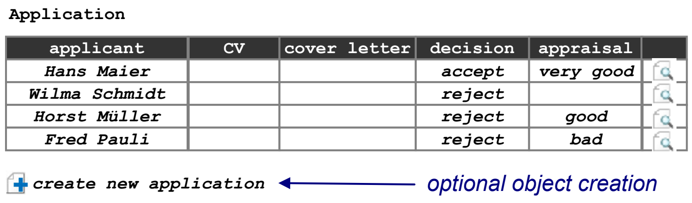


Figure 8.9: Markings of a micro process instance and their transitions

8.2 Creating Object Instances

Using *create permissions* (cf. Def. 25), we can define which user role may create object instances of a particular object type at run-time. Activities for creating new object instances can be invoked based on the overview table provided for the corresponding object type (cf. Fig. 8.10). More precisely, depending on their create permissions, users may optionally create object instances of particular object types. These activities are denoted as *optional object creation* and are defined by *Execution Rule ER1*.


Application

applicant	CV	cover letter	decision	appraisal	
Hans Maier			*accept*	*very good*	
Wilma Schmidt			*reject*		
Horst Müller			*reject*	*good*	
Fred Pauli			*reject*	*bad*	

create new application

Figure 8.10: Optional creation of object instances

Definition 25 (Create permissions):
Let dm = (name, OTypeSet, RelTypeSet) be a data model and UserRoles be the set of all defined user roles.
A **create permission createPerm = (oType, role)** is a tuple where

- oType $\in$ OTypeSet is an object type.
- role $\in$ UserRoles is a user role.

CreatePermissions corresponds to the set of all definable create permissions.

Execution Rule (ER1: Optional object creation):
$\forall$ createPerm = (oType, r) $\in$ CreatePermissions:
An object instance of type oType may be optionally created by users owning role r.

8.3 Initializing Micro Process Instances

When creating an object instance, a corresponding micro process instance is automatically created and initialized using *Reaction Rule RR1*. For each component of the micro process instance then an *initial marking* is set. First, the micro process instance itself is marked as RUNNING (cf. Reaction Rule RR1a). Furthermore, all *states* of the micro process instance except its *start state* are marked as WAITING (cf. Reaction Rule RR1b); remember that this indicates that they may still be activated during micro process execution. By contrast, the *start state* is marked as ACTIVATED (cf. Reaction Rule RR1c).
If a (start) state becomes marked as ACTIVATED, attribute values and relations referred by its micro steps must be set. Hence, after creating a micro process instance all *micro steps* (and *value steps*) corresponding to the start state, except the start micro step, are marked as READY (cf. Reaction Rules RR1d + g). Since the *start micro step* refers to the OID of the related object instance, it is immediately marked as UNCONFIRMED (cf. Reaction Rule RR1e); i.e., the OID is available as soon as the object instance is created. In turn, all other micro steps and value steps not belonging to the start state are marked as WAITING (cf. Reaction Rules RR1f + h). Finally, all micro transitions and backward transitions are initially marked as WAITING (cf. Reaction Rules RR1i + j).

Reaction Rule (RR1: Initializing a micro process instance):
Let dm = (name, OTypeSet, RelTypeSet) $\in$ DM be a data model and ds = (dm, OSet, RelSet) be a corresponding data structure. Further, let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Finally, let startState = (name, sMicStepSet) $\in$ StateSet be the start state of micProcInstance and startMicStep $\in$ MicStepSet be its corresponding start micro step.

When creating an object instance o (i.e., OSet = OSet $\cup$ $\{o\}$ with o = (oid, oType, attrval) $\wedge$ o.oid = micProcInstance.oid) then:

a) $M_{MicProc}$(micProcInstance) := RUNNING;
i.e., the micro process instance is initially marked as RUNNING.

b) $\forall$ state $\in$ StateSet - {startState}: M_{State}(state) := WAITING;
i.e., all states except the start state are initially marked as WAITING.

c) M_{State}(startState) := ACTIVATED;
i.e., the start state is initially marked as ACTIVATED.

d) $\forall$ micStep $\in$ startState.sMicStepSet - {startMicStep}: $M_{MicStep}$(micStep) := READY;
i.e., all micro steps belonging to the start state are initially marked as READY.

e) $M_{MicStep}$(startMicStep) := UNCONFIRMED;
i.e., the start micro step is initially marked as UNCONFIRMED.

f) $\forall$ micStep $\in$ MicStepSet - {startState.sMicStepSet}: $M_{MicStep}$(micStep) := WAITING;
i.e., all other micro steps not belonging to the start state are marked as WAITING.

g) $\forall$ micStep = (ref, ValueSteps) $\in$ MicStepSet with micStep $\notin$ startState:
$\forall$ valueStep $\in$ ValueSteps: $M_{MicStep}$(valueStep) := WAITING;
i.e., all value steps of any micro step not belonging to the start state are initially marked as WAITING.

h) $\forall$ micStep = (ref, ValueSteps) $\in$ MicStepSet with micStep $\in$ startState:
$\forall$ valueStep $\in$ ValueSteps: $M_{MicStep}$(valueStep) := READY;
i.e., all value steps of any micro step belonging to the start state are initially marked as READY.

i) $\forall$ micTrans $\in$ MicTransSet: $M_{MicTrans}$(micTrans) := WAITING;
i.e., all micro transitions are initially marked as WAITING.

j) $\forall$ backTrans $\in$ BackTransSet: $M_{BackTrans}$(backTrans) := WAITING;
i.e., all backward transitions are initially marked as WAITING.

Example 8.2 (Applying Reaction Rule RR1):
Consider the `review` micro process instance from Fig. 8.11, which is currently marked as RUNNING. Start state `initialized` is marked as ACTIVATED. In turn, all other states (i.e., `pending`, `reject proposed`, `invitation proposed`, and `finished`) are initially marked as WAITING. Micro steps `urgency` and `return date`, which both belong to the start state, are marked as READY. By contrast, the start micro step, is marked as UNCONFIRMED. All other micro steps, value steps, and micro transitions are initially marked as WAITING.

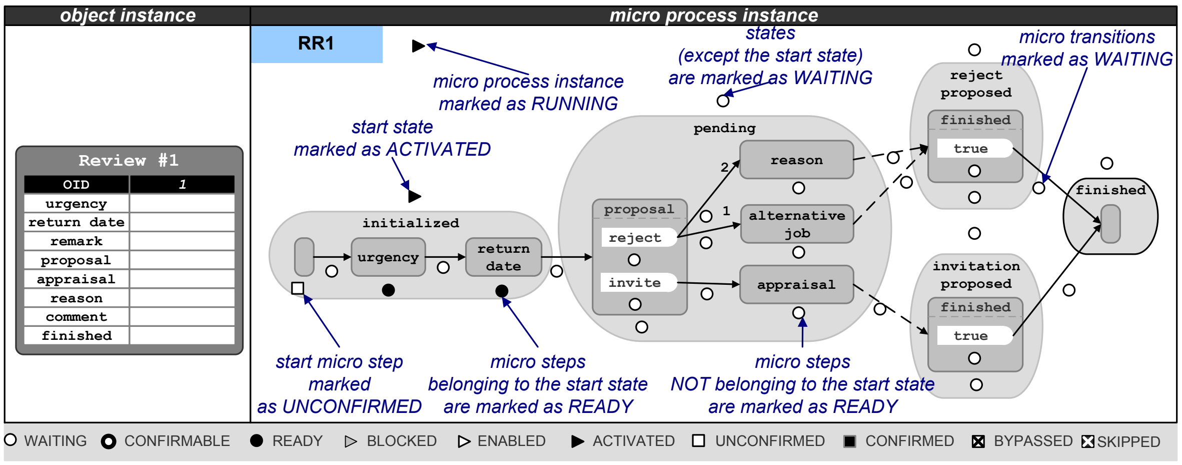


Figure 8.11: Applying Reaction Rule RR1

The initial markings described provide the basis for defining the operational semantics of micro processes. In particular, marking UNCONFIRMED of the start micro step constitutes the trigger of a set of rules that drive micro process execution. As illustrated in Fig. 8.12, after creating an object instance, Reaction Rule RR1 is triggered. According to this rule, the micro process instance the object instance refers to is automatically initialized.

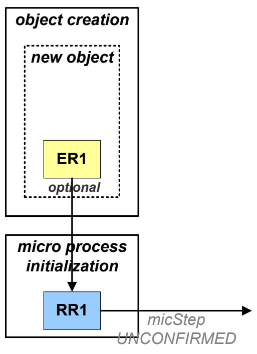


Figure 8.12: Rules for initializing a micro process instance

8.4 State-internal Execution

We first focus on the state-internal execution of a micro process instance. Generally, for each state, the required attribute values or relations must be defined by executing respective atomic (and value-specific) micro steps. When a state becomes enabled during micro process execution (i.e., during the processing of an object instance), usually, a form-based activity for entering the required attribute values is generated (cf. Fig. 8.13).

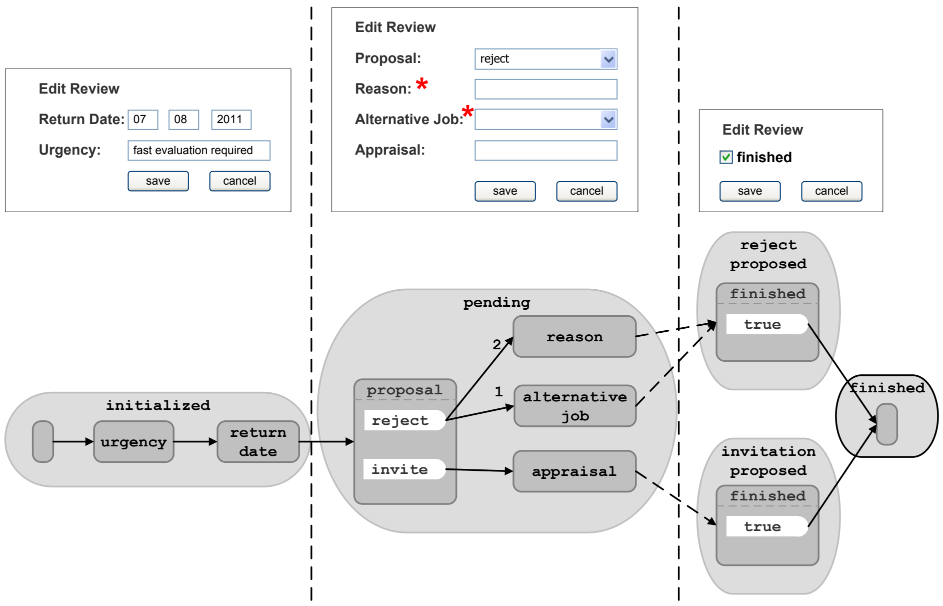


Figure 8.13: State-specific generation of form-based activities

The input fields of a form-based activity must be filled in by the responsible user when the corresponding state becomes activated (cf. Sect. 7.4). Thereby, the user is guided in setting the required attribute values (or relations); e.g., by highlighting the input fields to be set next. In order to capture the internal logic of a form for setting object attributes, the corresponding micro steps are linked with internal micro transitions.

8.4.1 Deterministic Execution

Using (state-)internal micro transitions, the internal logic of a form-based activity can be defined; i.e., the default order in which the mandatory input fields of the corresponding form shall be edited. In turn, this allows guiding users in filling respective forms. Despite any predefined sequence of micro steps, users should be allowed to choose the work practice they prefer; i.e., the order in which values are assigned to object attributes does not have to coincide with the one defined for the corresponding micro steps of the form-based activity. In particular, at runtime a micro step is completed as soon as a value is assigned to its object attribute (or relation). If the attribute value (or relation) the next micro step refers to is missing, the respective input field will be marked as mandatory (e.g., using a red asterisk).

We first discuss how deterministic micro process instances are executed. When a micro process instance becomes initialized, its start micro step, which belongs to the currently activated start state, is marked as UNCONFIRMED (cf. Reaction Rule RR1). Note that the start micro step is empty and hence does not refer to any object attribute (or relation); i.e., no data input is required to reach this start micro step. If there are other micro steps belonging to the start state, values for the attributes (or relations) these micro steps refer to are required. Consider micro steps `urgency` and `return date` in Fig. 8.14, for which respective attribute values must be set by the responsible user. When a micro step becomes marked as UNCONFIRMED, all (implicit) micro transitions outgoing from this micro step immediately change their marking to READY. Accordingly, the subsequent micro step is reachable and micro process execution may proceed. Note that internal micro transitions do not require any user commitment. To realize this behavior, we introduce *Marking Rule MR1* which is triggered when a micro step changes its marking from ACTIVATED to UNCONFIRMED.

Marking Rule (MR1: Marking implicit micro transitions as READY):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Then:

```
∀ micStep ∈ MicStepSet with $M_{MicStep}$(micStep) = UNCONFIRMED:
    ∀ micTrans ∈ outtrans(micStep) with explicit(micTrans) = FALSE:
        $M_{MicTrans}$(micTrans) := READY;
```

i.e., when marking a micro step as UNCONFIRMED, all implicit micro transitions outgoing from this micro step are marked as READY.

Example 8.3 (Applying Marking Rule MR1):
Consider Fig. 8.14. After initializing the `review` micro process instance, the start micro step is marked as UNCONFIRMED. According to Marking Rule MR1, this triggers the marking of the micro transition between the start micro step and micro step `urgency` as READY.

Assume that a micro transition is marked as READY. Then its target micro step can be reached when the attribute value (or relation) it refers to becomes available. When applying Marking Rule MR1, therefore, *Marking Rule MR2* is triggered afterwards. This rule allows the target micro step of the micro transition to change its marking from READY to ENABLED. Note that this indicates that a value for the attribute or relation the respective micro step refers to is then mandatorily required.

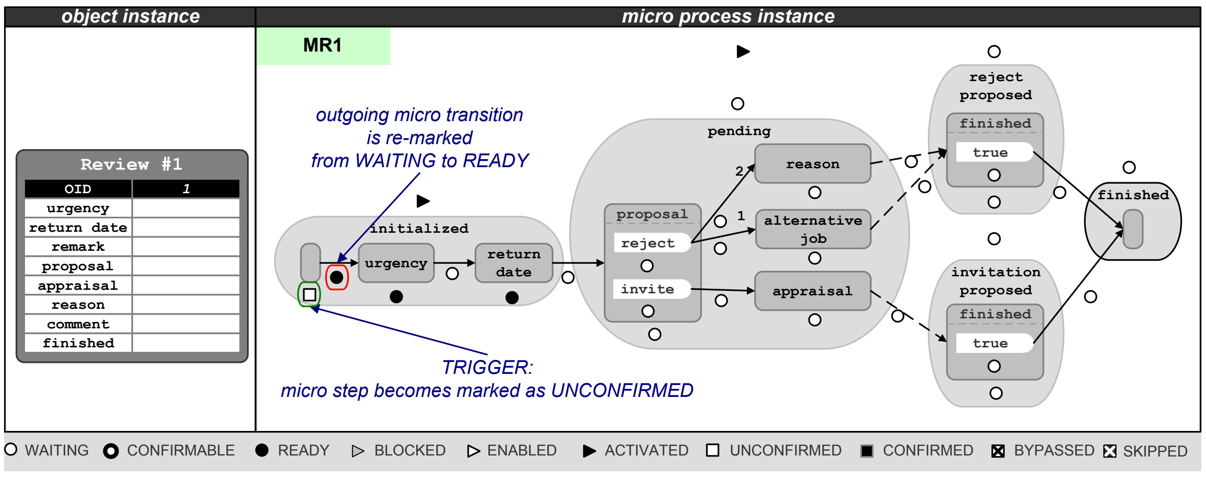


Figure 8.14: Applying Marking Rule MR1

Marking Rule (MR2: Marking micro steps as ENABLED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

$\forall$ micTrans $\in$ MicTransSet with $M_{MicTrans}$(micTrans) = READY: $M_{MicStep}$(target) := ENABLED;

i.e., when the marking of a micro transition changes to READY, its target micro step will be marked as ENABLED.

Example 8.4 (Applying Marking Rule MR2):
Consider Fig. 8.15 and the depicted `review` micro process instance. Assume that the micro transition between the start micro step and micro step `urgency` becomes marked as READY. In turn, this triggers a change of the marking of micro step `urgency` from READY to ENABLED according to Marking Rule MR2.

Marking a micro step as ENABLED indicates that a value for the attribute (or relation) this micro step refers to is required. Accordingly, in the corresponding user form the respective input field is highlighted (cf. Fig. 8.16). If an attribute value (or relation) is missing for a micro step, a user input is mandatory to proceed with micro process execution. This is expressed by *Execution Rule ER2*.

Execution Rule (ER2: Data input required):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Further, let Attr denote the set of all attributes of any attribute type and RelSet be a finite set of relations. Then:

$\forall$ micStep $\in$ MicStepSet with micStep.ref $\neq$ NULL $\wedge$ $M_{MicStep}$(micStep) = ENABLED:

- ref=(attrType, attrValue) $\in$ Attr, $\Rightarrow$ attrValue must be set
- ref=(relType, soid, toid) $\in$ RelSet $\wedge$ soid = oid, $\Rightarrow$ toid must be set

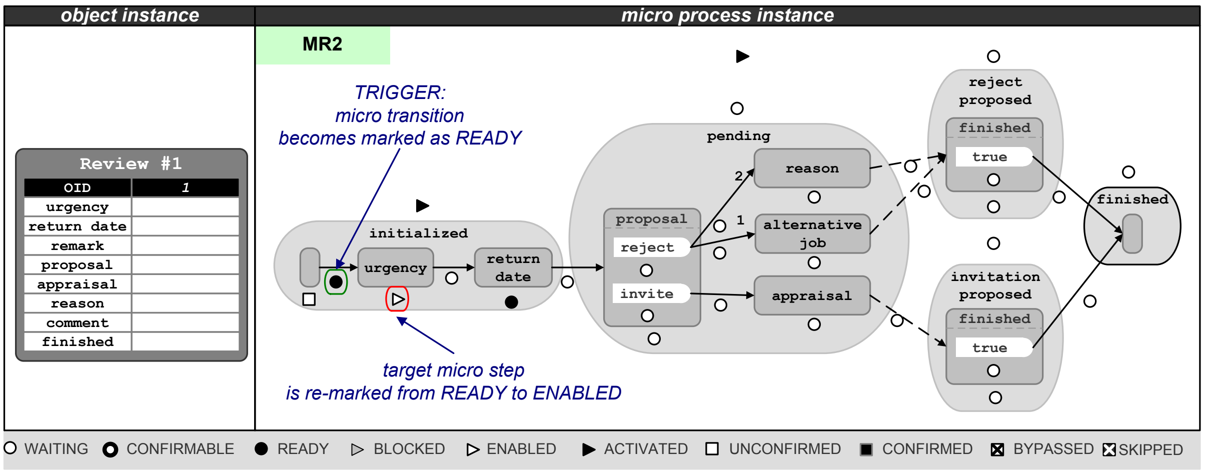


Figure 8.15: Applying Marking Rule MR2

If an atomic (or value-specific) micro step changes its marking to ENABLED, a value for the corresponding attribute (or relation) must be set.

Example 8.5 (Applying Execution Rule ER2):
Consider Fig. 8.16. Since micro step `urgency` is currently marked as ENABLED, a value for attribute `urgency` is mandatorily required. Hence, the corresponding input field is highlighted using a red asterisk.

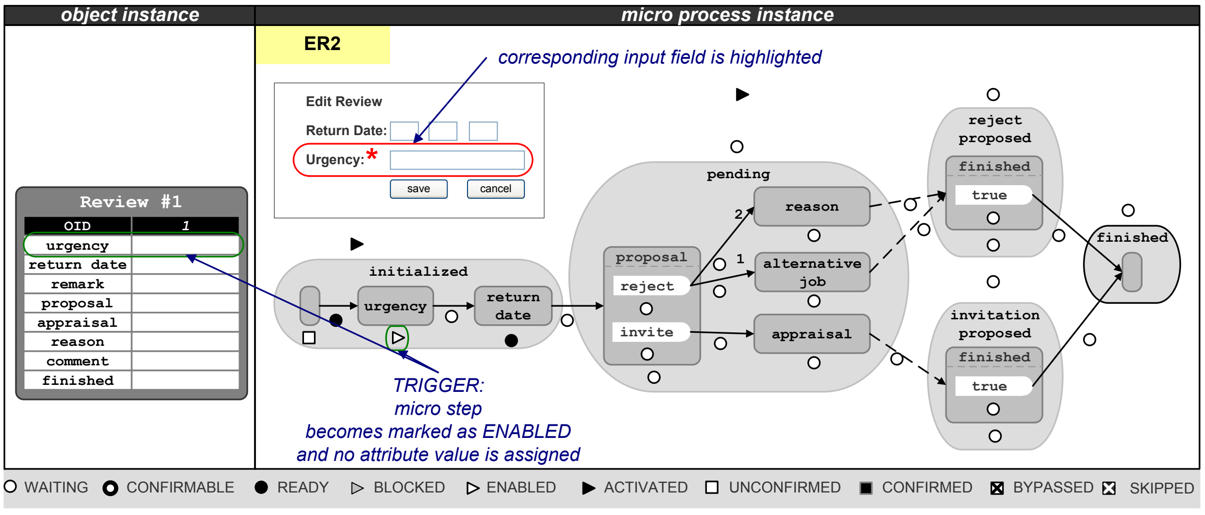


Figure 8.16: Applying Execution Rule ER2

When a required attribute value (or relation) is set, the respective micro step will be marked as ACTIVATED according to *Reaction Rule RR2*. Note that if the required attribute value (or relation) is already available, Execution Rule ER2 will be skipped and Reaction Rule RR2 directly be executed.

Reaction Rule (RR2: Marking micro steps as ACTIVATED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

$\forall$ micStep=(ref, ValueSteps) $\in$ MicStepSet with $M_{MicStep}$(micStep) = ENABLED $\wedge$ ref $\neq$ NULL $\wedge$ ValueSteps = $\emptyset$:
    ref=(attrT, attrV) $\in$ Attr $\wedge$ attrV $\neq$ NULL $\vee$ ref=(relT, s, t) $\in$ RelSet $\wedge$ t $\neq$ NULL, $\Rightarrow$
        $M_{MicStep}$:= ACTIVATED;

i.e., all atomic micro steps will be marked as ACTIVATED if a value for the corresponding attribute or relation is available.

Example 8.6 (Applying Reaction Rule RR2):
Consider Fig. 8.17. Value "high" is assigned to attribute `urgency`. According to Reaction Rule RR2, then micro step `urgency` is marked as ACTIVATED.

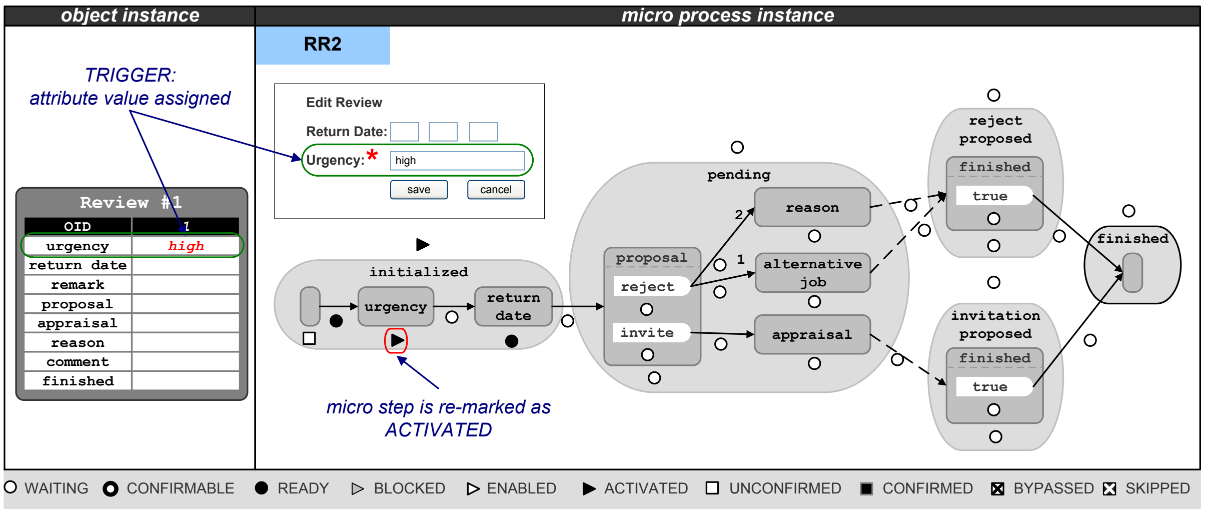


Figure 8.17: Applying Reaction Rule RR2

If a micro step is marked as ACTIVATED, one may proceed with micro process execution. According to *Marking Rule MR3*, therefore, all incoming micro transitions, which are currently marked as READY, change their marking to ENABLED.

Marking Rule (MR3: Marking micro transitions as ENABLED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

$\forall$ micStep $\in$ MicStepSet with $M_{MicStep}$(micStep) = ACTIVATED:
    $\forall$ micTrans $\in$ intrans(micStep) with $M_{MicTrans}$(micTrans) = READY:
        $M_{MicTrans}$(micTrans) := ENABLED;

i.e., if a micro step becomes marked as ACTIVATED, all incoming micro transitions currently marked as READY change their marking to ENABLED.

Example 8.7 (Applying Marking Rule MR3):
In our running example, the micro transition connecting the start micro step with micro step `urgency` becomes marked as ENABLED (cf. Fig. 8.18).

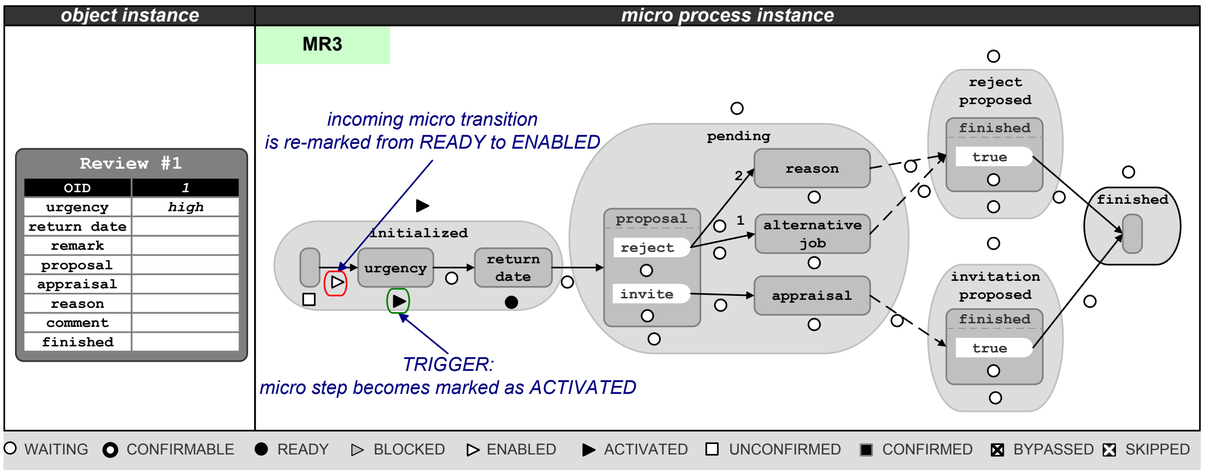


Figure 8.18: Applying Marking Rule MR3

If only one micro transition becomes marked as ENABLED at a certain point in time, *Marking Rule MR4* will be triggered immediately. In particular, deterministic micro transitions, currently marked as ENABLED, immediately change their marking to ACTIVATED.[2].

Marking Rule (MR4: Marking micro transitions as ACTIVATED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

$\exists!$ micTrans $\in$ MicTransSet with $M_{MicTrans}$(micTrans) = ENABLED, $\Rightarrow$ $M_{MicTrans}$(micTrans) := ACTIVATED.

i.e., if there exists only one micro transition currently marked as ENABLED, it will immediately marked as ACTIVATED.

Example 8.8 (Applying System Rule MR4):
Consider Fig. 8.19: The micro transition connecting the start micro step with micro step `urgency` will immediately marked as ACTIVATED when the micro transition becomes ENABLED.

[2] Regarding a non-deterministic micro process instance, however, multiple succeeding micro steps may exist. In the given context, when the attribute values (or relations) required for these steps become available, several of them may be simultaneously marked as ACTIVATED. To prevent parallel micro step execution at this level, only one micro step (and hence one state) may be reached. To ensure this, micro transitions originating from the same micro steps are associated with different priorities. In order to evaluate priorities during micro process execution, the respective micro transitions (targeting at micro steps currently marked as ACTIVATED) will be marked as ENABLED (cf. Marking Rule MR3). Only the micro transition with the highest priority among all enabled micro transitions will be selected. For this purpose, marking ENABLED is used for micro transitions to evaluate which subsequent micro step (currently marked as ACTIVATED) may be actually reached; i.e., which alternative path shall be selected (see Sect. 8.4.3 for details).

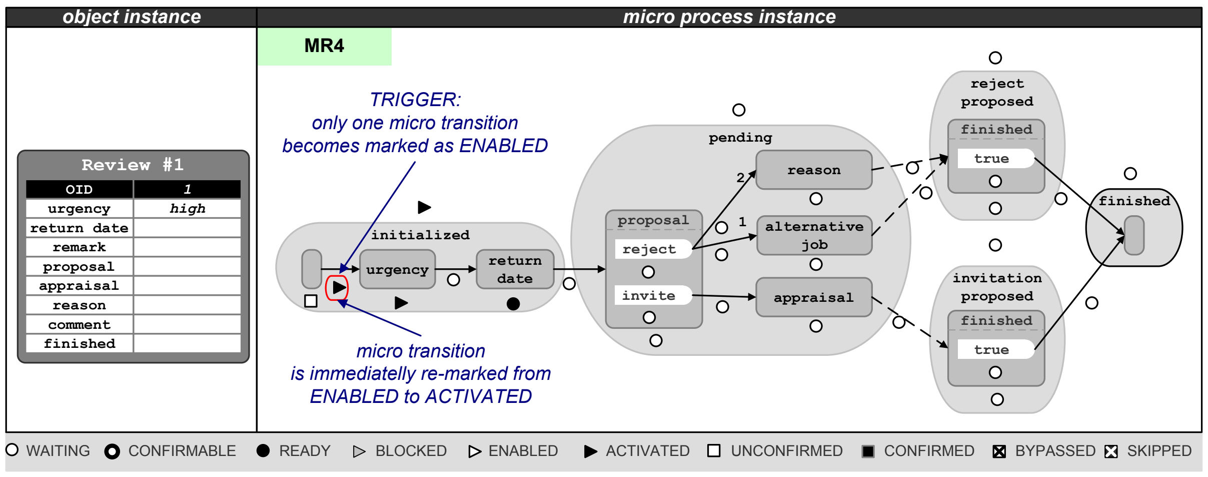


Figure 8.19: Applying Marking Rule MR4

If a micro transition is marked as ACTIVATED, its target micro step can be reached and is therefore marked as UNCONFIRMED. This is defined by *Marking Rule MR5*.

Marking Rule (MR5: Marking micro steps as UNCONFIRMED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Then:

∀ micTrans=(source, target, priority) ∈ MicTransSet with $M_{MicTrans}$(micTrans) = ACTIVATED:
    $M_{MicStep}$(target) := UNCONFIRMED;

i.e., if a micro transition becomes marked as ACTIVATED, its target micro step will be re-marked as UNCONFIRMED.

Example 8.9 (Applying Marking Rule MR5):
Regarding our `review` micro process instance (cf. Fig. 8.20), micro step `urgency` can now be reached and marked as UNCONFIRMED.

When a micro step becomes marked as UNCONFIRMED, its incoming micro transition is also marked as UNCONFIRMED to indicate the path selected during micro process execution. To ensure this, the application of Marking Rule MR5 triggers *Marking Rule MR6*.

Marking Rule (MR6: Marking micro transitions as UNCONFIRMED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Then:

∀ micStep ∈ MicStepSet with $M_{MicStep}$(micStep) = UNCONFIRMED:
    ∀ micTrans ∈ intrans(micStep) with $M_{MicTrans}$(micTrans) = ACTIVATED ∧ isexternal(micTrans) = FALSE:
        $M_{MicTrans}$(micTrans) := UNCONFIRMED;

i.e., if a micro step becomes marked as UNCONFIRMED, its incoming (internal) micro transition, which is currently marked as ACTIVATED, will be re-marked as UNCONFIRMED.

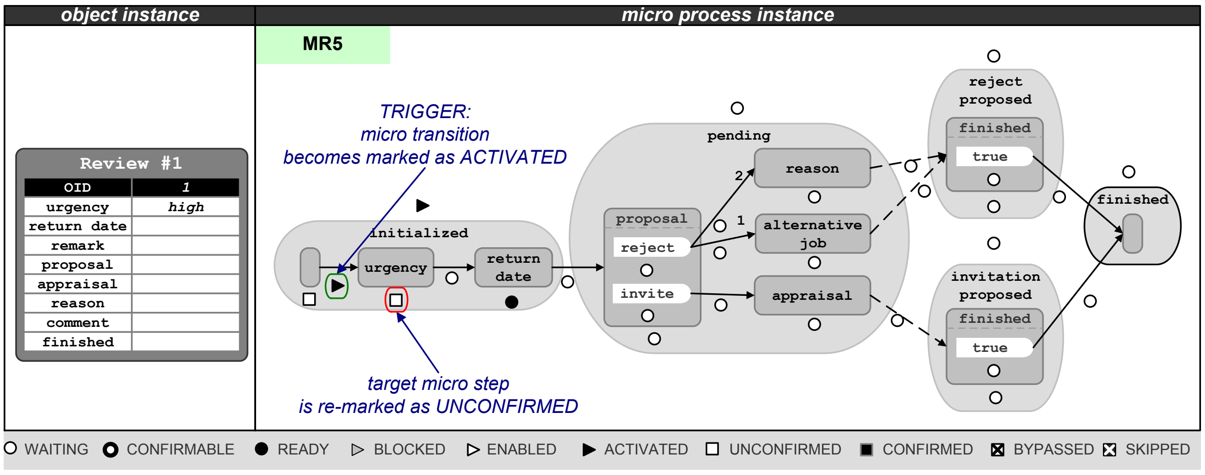


Figure 8.20: Applying Marking Rule MR5

Example 8.10 (Applying Marking Rule MR6):
Consider Fig. 8.21. The micro transition connecting the start micro step with micro step `urgency` is re-marked as UNCONFIRMED.

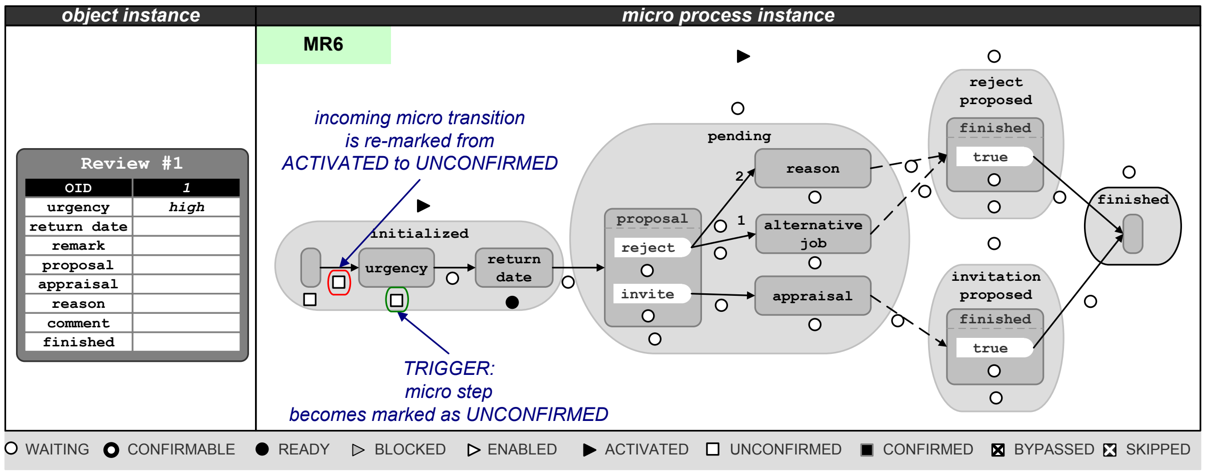


Figure 8.21: Applying Marking Rule MR6

The execution of Marking Rule MR5 (i.e., marking micro steps as UNCONFIRMED) triggers Marking Rule MR1 (cf. Fig. 8.22). According to Marking Rule MR1, all micro transitions, whose source micro step has just been marked as UNCONFIRMED, change their marking from WAITING to READY.

The rules introduced for the deterministic execution of micro process instances are summarized in Fig. 8.22: After marking a micro step as UNCONFIRMED, its outgoing (internal) micro transitions change their marking from WAITING to READY (i.e., Marking Rule MR1). In turn, this triggers Marking Rule MR2 according to which the target micro steps of these micro transitions

(with marking READY) change their marking from READY to ENABLED. If the attribute or relation to which such a micro step refers has been already set, Reaction Rule RR2 is immediately triggered. Opposed to this, if the required data input (i.e., attribute value or relation) is still missing, Execution Rule ER2 is triggered. According to ER2, a mandatory form-based activity is automatically assigned to the worklist of the responsible user. Using Reaction Rule RR2, in turn, the micro step will be marked as ACTIVATED when the required data input becomes available. This triggers Marking Rule MR3 according to which all incoming micro transitions change their marking from READY to ENABLED.
Regarding deterministic micro process execution, exactly one micro transition is marked as ENABLED at any point during process execution. For this special case, the respective transition will be immediately marked as ACTIVATED according to Marking Rule MR4. When a micro transition becomes marked as ACTIVATED, in turn, its target micro step is re-marked from ACTIVATED to UNCONFIRMED (i.e., Marking Rule MR5). On one hand this indicates that all incoming micro transitions (marked as ACTIVATED) may change their marking to UNCONFIRMED. On the other, all outgoing (internal) micro transitions are then marked as READY (cf. Marking Rule MR1). This then triggers the introduced rules iteratively.

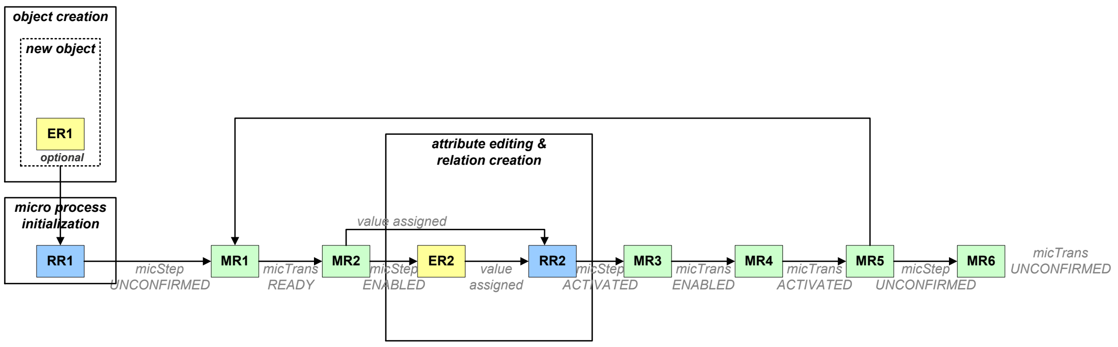


Figure 8.22: Rules for deterministic micro process execution

8.4.2 Handling Value-specific Micro Steps

So far, we have described the rules enabling a state-internal execution of deterministic micro processes. In this context, we have focused on atomic micro steps. This section extends the rules driving micro process execution by additionally considering *value-specific micro steps*.

If a state changes its marking from WAITING to ACTIVATED, exactly one of its incoming external micro transitions is marked as READY. According to Marking Rule MR2, when a micro transition becomes marked as READY, all target micro steps will be marked as ENABLED. If the target micro step is a value-specific one, however, all value steps belonging to it must be marked as ENABLED as well. For this purpose, we extend Marking Rule MR2 (cf. Marking Rule MR2').

Marking Rule (MR2': Marking value steps as ENABLED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

a) see MR2 in Sect. 8.4.1

b) $\forall$ micStep=(ref, ValueSteps) $\in$ MicStepSet with $M_{MicStep}$(micStep) = ENABLED:
   $\forall$ valueStep $\in$ ValueSteps: $M_{MicStep}$(valueStep) := ENABLED;
   i.e., when a value-specific micro step becomes marked as ENABLED, its corresponding value steps will be marked as ENABLED as well.

Example 8.11 (Applying Marking Rule MR2'):
Regarding our running example (cf. Fig 8.23), state `pending` contains the value-specific micro step `proposal`. Since the micro transition between micro steps `return date` and `proposal` is marked as READY, its target micro step (i.e., micro step `proposal`) is re-marked to ENABLED according to Marking Rule MR2; i.e., micro step `proposal` is marked as ENABLED. In addition, using Marking Rule MR2', all value steps belonging to micro step `proposal` (i.e., value step `reject` as well as micro step `invite`) are marked as ENABLED.

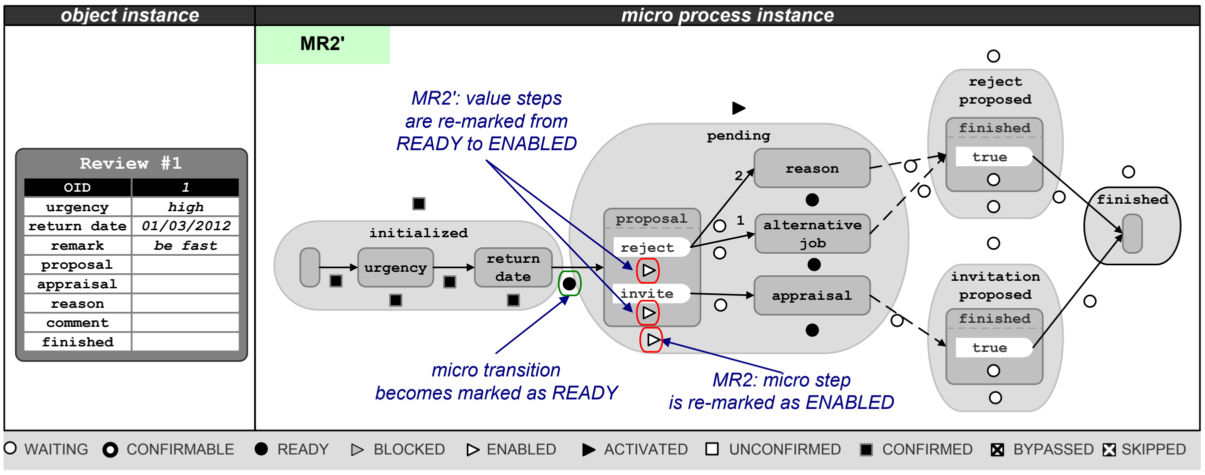


Figure 8.23: Applying Marking Rule MR2'

Marking a micro step as ENABLED indicates that a value for the attribute (or relation) the micro step refers to is required; i.e., Execution Rule ER2 must be applied. For this purpose, in the respective form the corresponding input field is highlighted (e.g., the input field corresponding to micro step `proposal` in Fig. 8.24).

When a value for an attribute (or relation), a value-specific micro step refers to, becomes available, whether the micro step can be marked as ACTIVATED depends on its value steps. For this purpose, one first must evaluate which of the value steps may be activated. More precisely, a value step will be marked as ACTIVATED if its predicate evaluates to true. For this purpose, we extend Reaction Rule RR2 (cf. Reaction Rule RR2').

Reaction Rule (RR2': Marking value steps as ACTIVATED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

a) see RR2 in Sect. 8.4.1

b) $\forall$ micStep=(ref, ValueSteps) $\in$ MicStepSet with $M_{MicStep}$(micStep) = ENABLED $\wedge$ micStep.ref $\neq$ NULL:
   $\forall$ valueStep $\in$ ValueSteps with constraint(valueStep) = TRUE:
      $M_{MicStep}$(valueStep) := ACTIVATED;
   i.e., all value steps evaluating to True are marked as ACTIVATED.

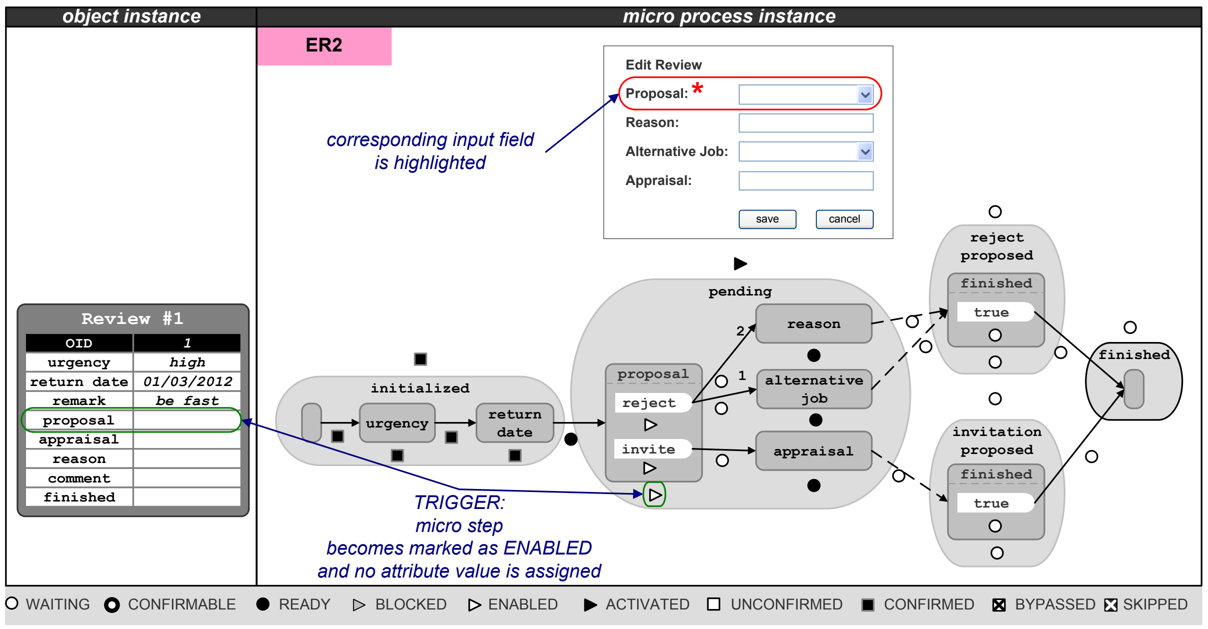


Figure 8.24: Applying Execution Rule ER2 for value-specific micro steps

Example 8.12 (Applying Marking Rule RR2'):
Consider Fig. 8.25. When a value for attribute `proposal` becomes available, whether micro step `proposal` can be marked as ACTIVATED depends on its value steps. Since value "reject" is assigned to attribute `proposal`, value step `reject` is marked as ACTIVATED.

Opposed to atomic micro steps, which will be immediately marked as ACTIVATED when the required attribute value (or relation) becomes available, a value-specific micro step may only be marked as ACTIVATED if at least one of its value steps is marked as ACTIVATED (cf. Reaction Rule RR2").

Reaction Rule (RR2": Marking value-specific micro steps as ACTIVATED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

a) see RR2 in Sect. 8.4.1

b) see RR2' in Sect. 8.4.2

c) $\forall$ micStep=(ref, VSteps) $\in$ MicStepSet with $M_{MicStep}$(micStep) = ENABLED $\wedge$ ref $\neq$ NULL $\wedge$ VSteps $\neq \emptyset$:
$\exists$ valueStep $\in$ VSteps with $M_{MicStep}$(valueStep) = ACTIVATED, $\Rightarrow$
$M_{MicStep}$(micStep) := ACTIVATED;
i.e., a value-specific micro step changes its marking from ENABLED to ACTIVATED if at least one of its value steps is marked as ACTIVATED.

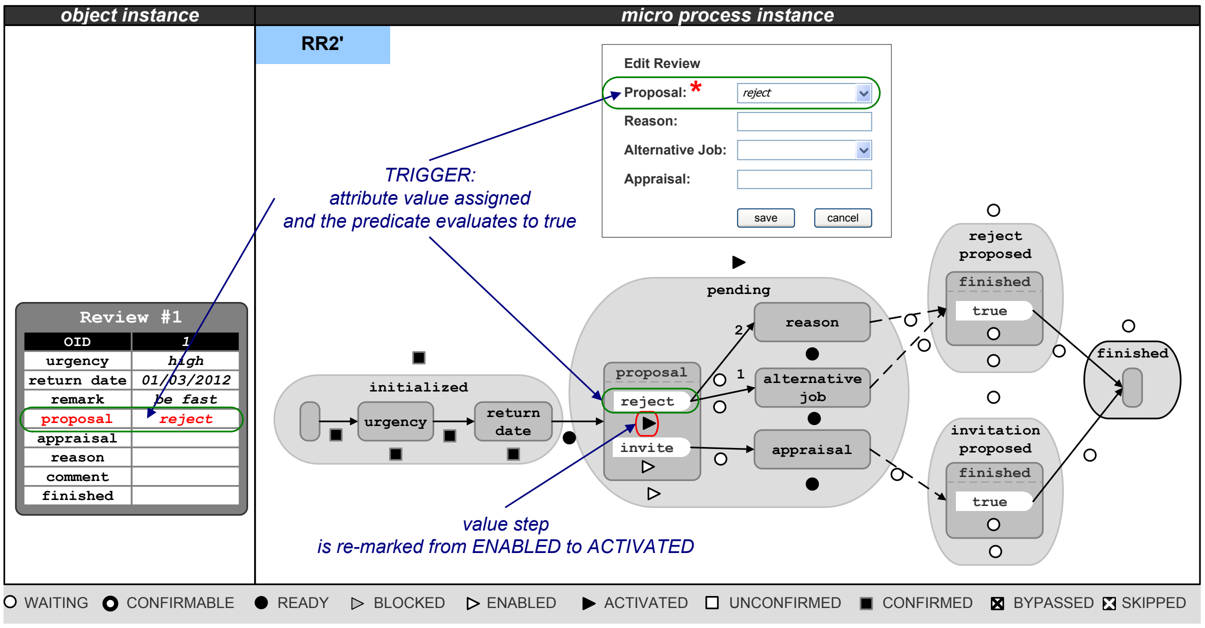


Figure 8.25: Applying Marking Rule RR2'

Example 8.13 (Applying Marking Rule RR2"):
Consider Fig. 8.26. Since value step `reject` is currently marked as ACTIVATED, micro step `proposal` changes its marking from ENABLED to ACTIVATED.

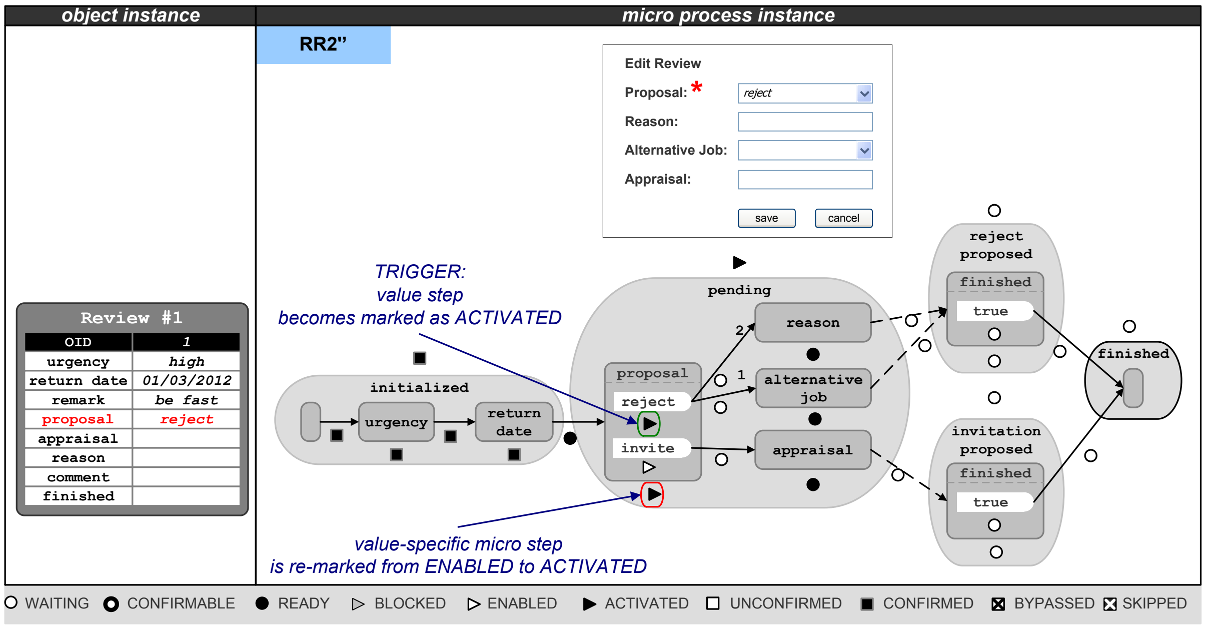


Figure 8.26: Applying Marking Rule RR2"

If no value step of any enabled value-specific micro step is currently ACTIVATED, but a respective attribute value is available, two scenarios are considered:

First, if the (value-specific) micro step itself has an outgoing micro transition, it can be handled like an atomic micro step; i.e., it will be marked as ACTIVATED when the required attribute value (or relation) becomes available (whether any of the predicates related to the value steps are met). This indicates that micro process execution may proceed regardless of the specific attribute value assigned (cf. Reaction Rule RR2"'d). Second, if no such micro transition exists, and none of the value steps can be activated although an attribute value (or relation) is available, the respective value-specific micro step will be marked as BLOCKED indicating that micro process execution must not proceed. For these micro steps another attribute value (or relation) must be assigned (cf. Reaction Rule RR2"'e).

Reaction Rule (RR2"': Marking value-specific micro steps as BLOCKED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

a) see RR2 in Sect. 8.4.1

b)-c) see RR2' in Sect. 8.4.2

d) $\forall$ micStep=(ref, VSteps) $\in$ MicStepSet with $M_{MicStep}$(micStep) = ENABLED $\wedge$ ref $\neq$ NULL $\wedge$ VSteps $\neq \emptyset$:
    if outtransCount(micStep) ≥ 1:
    (ref=(attrT, attrV) $\in$ Attr $\wedge$ attrV $\neq$ NULL) $\vee$ (ref=(relT, s, t) $\in$ RelSet $\wedge$ t $\neq$ NULL), $\Rightarrow$
        $M_{MicStep}$:= ACTIVATED;
   i.e., a value-specific micro step changes its marking from ENABLED to ACTIVATED if a value for its corresponding attribute becomes available and the micro step itself has at least one outgoing micro transition (in addition to the outgoing micro transitions of value steps belonging to this micro step).

e) $\forall$ micStep=(ref, VSteps) $\in$ MicStepSet with
    $M_{MicStep}$(micStep) = ENABLED $\wedge$ micStep.ref $\neq$ NULL $\wedge$ VSteps $\neq \emptyset$ $\wedge$
    ((ref=(attrT, attrV) $\in$ Attr $\wedge$ attrV $\neq$ NULL) $\vee$ (ref=(relType, s, t) $\in$ RelSet $\wedge$ t $\neq$ NULL)):
        if outtransCount(micStep) = 0 $\wedge \nexists$ vStep $\in$ VStepswith $M_{MicStep}$(vStep) = ACTIVATED, $\Rightarrow$
            $M_{MicStep}$(micStep) := BLOCKED;
   i.e., a value-specific micro step changes its marking from ENABLED to BLOCKED if it has no outgoing micro transition and no value step currently marked as ACTIVATED.

To indicate that a value-specific micro step is blocked, the corresponding input field is highlighted (e.g., using a red exclamation mark).

Example 8.14 (Applying Marking Rule RR2"'):
Consider Fig. 8.27. For micro step `proposal` only value steps `reject` and `invite` exists. Both are associated with outgoing micro transitions. Consequently, if a value other than "reject" or "invite" is assigned to attribute `proposal`, the execution of the `review` micro process instance will be blocked; i.e., micro step `proposal` will be marked as BLOCKED. To make this situation transparent to users the corresponding input field referring to attribute `proposal` is then marked with a red exclamation mark.

In situations in which micro process execution is blocked due an incorrect attribute value, responsible users must reset the respective input field; i.e., delete the incorrect attribute value. Following this, according to *Reaction Rule RR3*, the respective micro step changes is marking from BLOCKED to ENABLED indicating that a corresponding attribute value is now missing. Note that value-specific micro steps always refer to an attribute, but not to a relation. If a new attribute value is assigned, Reaction Rule RR2 will be triggered again.

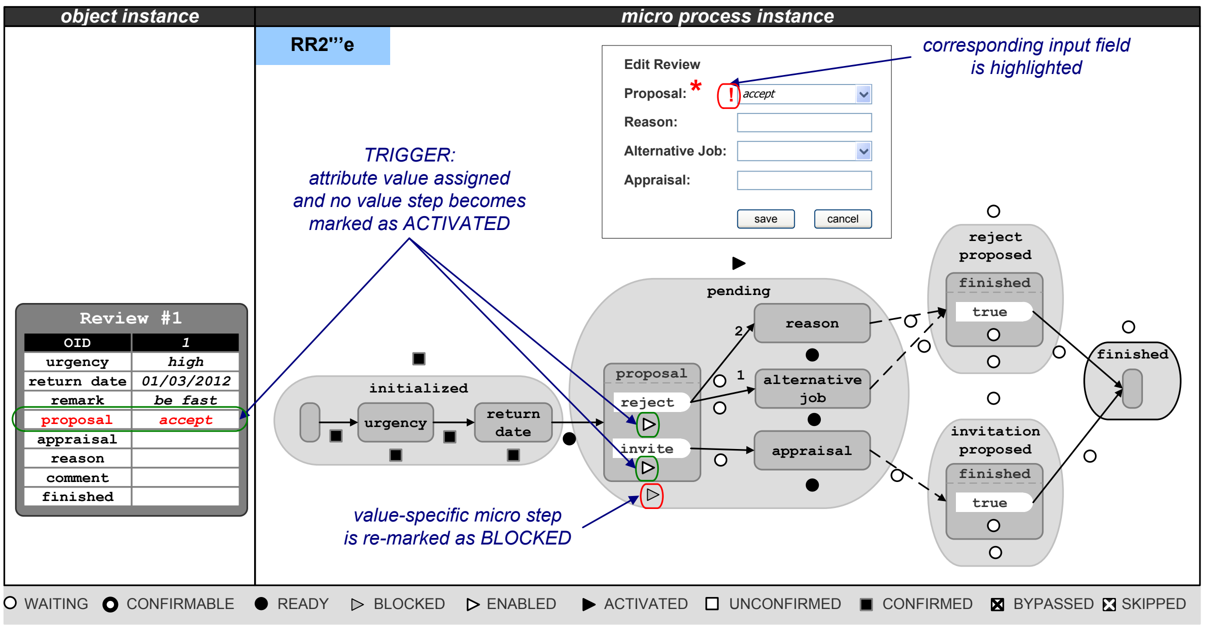


Figure 8.27: Applying Marking Rule RR2'''

Reaction Rule (RR3: Re-marking value-specific micro steps as ENABLED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

$\forall$ micStep=(ref, ValueSteps) $\in$ MicStepSet with $M_{MicStep}$(micStep) = BLOCKED:
 ref=(attrType, attrValue) $\in$ Attr $\wedge$ attrValue = NULL, $\Rightarrow$ $M_{MicStep}$(micStep) := ENABLED;
i.e., if the attribute value of a micro step, which is currently marked as BLOCKED, is deleted, the micro step will be re-marked as ENABLED.

Example 8.15 (Applying Marking Rule RR3):
Consider Fig. 8.28. If value "accept" is deleted from the input field corresponding to attribute `proposal`, the micro step `proposal` will be re-marked as ENABLED.

If another value is provided, which is covered by a value step (i.e., the respective predicate evaluates to True), micro process execution may proceed. For this purpose, the value step becomes marked as ACTIVATED (cf. Reaction Rule RR2'. In turn, this triggers Reaction Rule RR2" according to which the micro step itself changes its marking from ENABLED to ACTIVATED.

Example 8.16 (Applying Reaction Rule RR2' and RR2"):
Consider Fig. 8.29. If value "reject" is assigned, value step `reject` will be marked as ACTIVATED (cf. Reaction Rule RR2' which triggers Reaction Rule RR2"). In turn, when applying Reaction Rule RR2", the micro step itself (referring to attribute `proposal`) changes its marking from ENABLED to ACTIVATED.

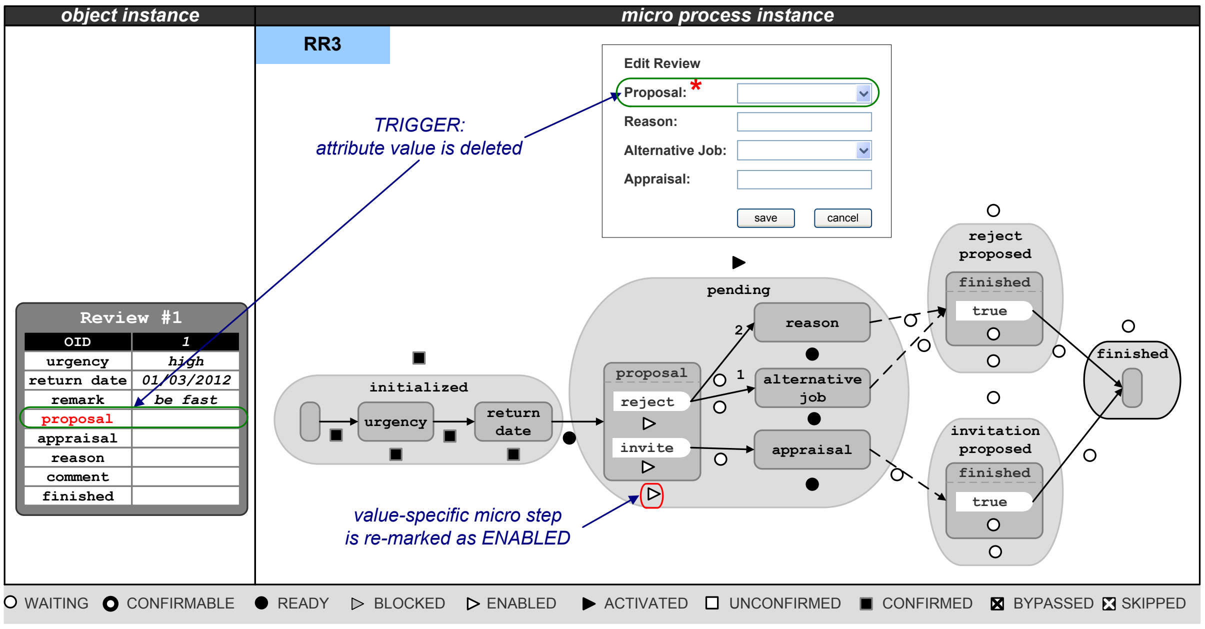


Figure 8.28: Applying Marking Rule RR3

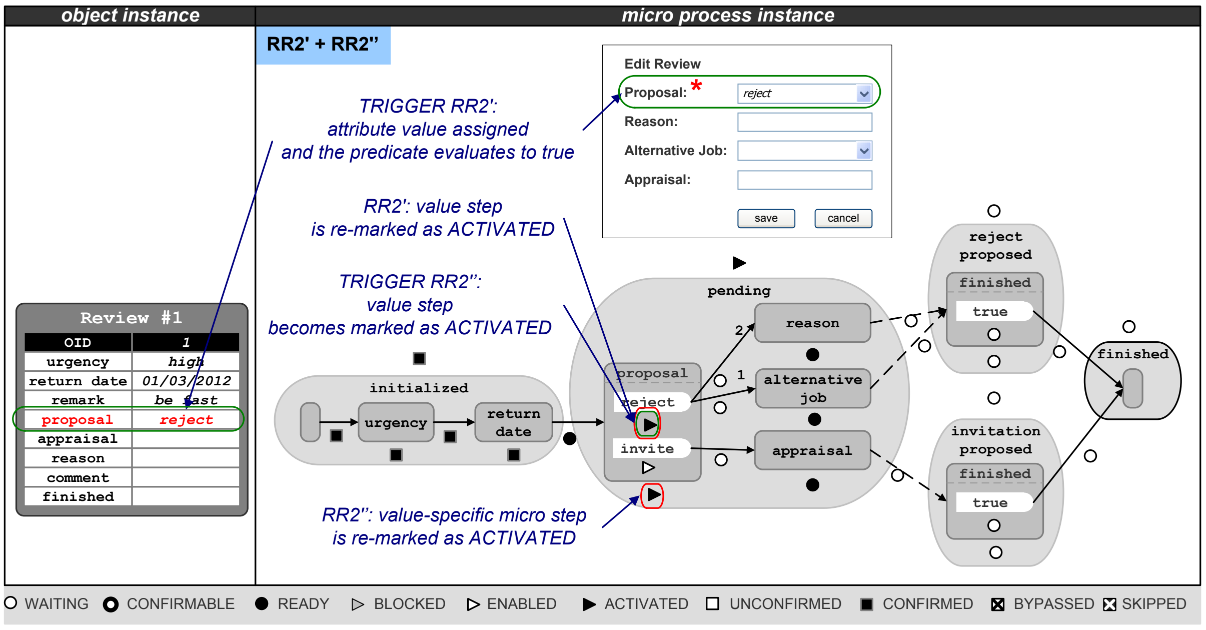


Figure 8.29: Applying Reaction Rule RR2' and RR2"

When a micro step becomes ACTIVATED, its incoming micro transition, which is currently marked as READY, is re-marked as ENABLED (cf. Marking Rule MR3). If this micro transition is the only one targeting at the respective micro step, it immediately changes its marking from ENABLED to ACTIVATED (cf. Marking Rule MR4). In turn, this triggers Marking Rule MR5, which allows the target micro step to change its marking from ACTIVATED to UNCONFIRMED (cf. Fig. 8.30).

When the micro step is a value-specific one, corresponding value steps must be re-marked as well. For this purpose, we extend Marking Rule MR5 (cf. Marking Rule MR5'). In particular, all value steps, currently marked as ACTIVATED, change their marking to UNCONFIRMED. Opposed to this, all value steps, still marked as ENABLED, are now re-marked as BYPASSED; i.e., their outgoing micro transitions must not fire anymore and hence it is not possible to activate the execution paths originating from these bypassed value steps.

Marking Rule (MR5': Marking value steps as UNCONFIRMED or BYPASSED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

a) see MR5 in Sect. 8.4.1

b) $\forall$ micStep=(ref, VSteps) $\in$ MicStepSet with $M_{MicStep}$(micStep) = UNCONFIRMED $\wedge$ VSteps $\neq \emptyset$:

$$\forall vS \in VSteps: M_{MicStep}(vS) := \begin{cases} \text{UNCONFIRMED}, & \text{if } M_{MicStep}(vS) = \text{ACTIVATED} \\ \text{BYPASSED}, & \text{if } M_{MicStep}(vS) = \text{ENABLED} \end{cases}$$

i.e., when a value-specific micro step becomes re-marked ACTIVATED to UNCONFIRMED, all value steps currently marked as ACTIVATED are re-marked as UNCONFIRMED. Furthermore, all value steps currently marked as ENABLED change their marking to BYPASSED.

Example 8.17 (Applying Marking Rule MR5'):
Consider Fig. 8.30. Value step `reject` changes its marking from ACTIVATED to UNCONFIRMED, whereas value step `invite` is re-marked as BYPASSED.

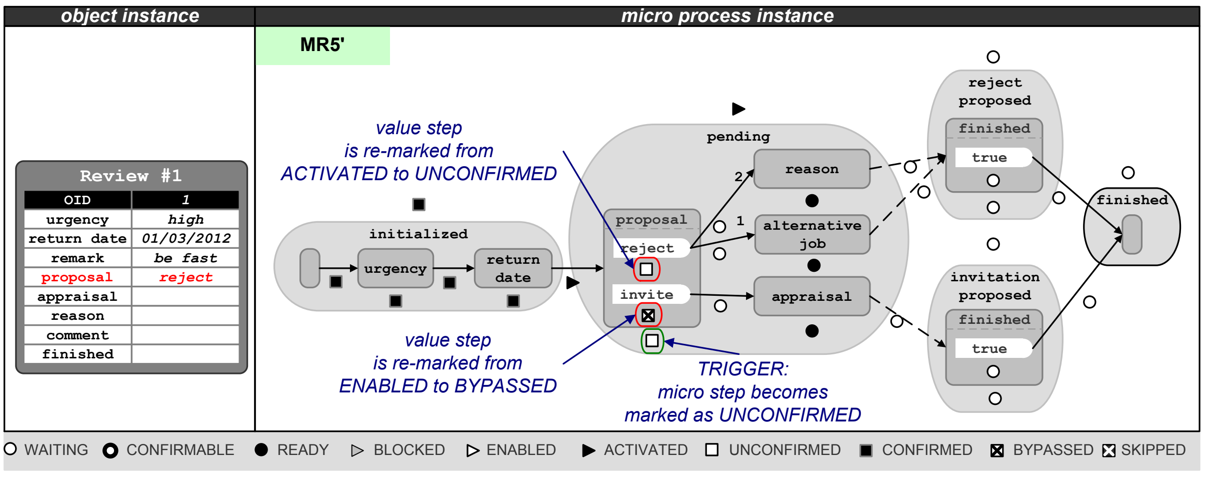


Figure 8.30: Applying Marking Rule MR5'

In turn, when a micro step is marked as UNCONFIRMED, its outgoing micro transitions are re-marked from WAITING to READY (cf. Marking Rule MR1). Similarly, when a value step is re-marked from ACTIVATED to UNCONFIRMED, its outgoing micro transitions are re-marked as READY (cf. Marking Rule MR1').

Marking Rule (MR1': Marking micro transitions originating from value steps as READY):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

a) see MR1 in Sect. 8.4.1

b) $\forall$ valueStep $\in$ ValueSteps with $M_{MicStep}$(valueStep) = UNCONFIRMED:
    $\forall$ micTrans $\in$ outtrans(valueStep) with explicit(micTrans) = FALSE:
        $M_{MicTrans}$(micTrans) := READY;
i.e., if a value step is marked as UNCONFIRMED, all implicit micro transitions outgoing from this micro step will be marked as READY.

Example 8.18 (Applying Marking Rule MR1'):
Consider Fig. 8.31. Value step `reject` is marked as UNCONFIRMED. This triggers Marking Rule MR1'. In particular, the micro transition between value step `reject` and micro step `reason` as well as the micro transition between value step `reject` and micro step `alternative job` are marked as READY; i.e., more than one succeeding micro step may be reached (cf. Sect. 8.4.3 for details). Opposed to this, the execution path comprising micro step `appraisal` is skipped due to a state-internal dead-path eliminating (cf. Sect. 8.4.4 for details).

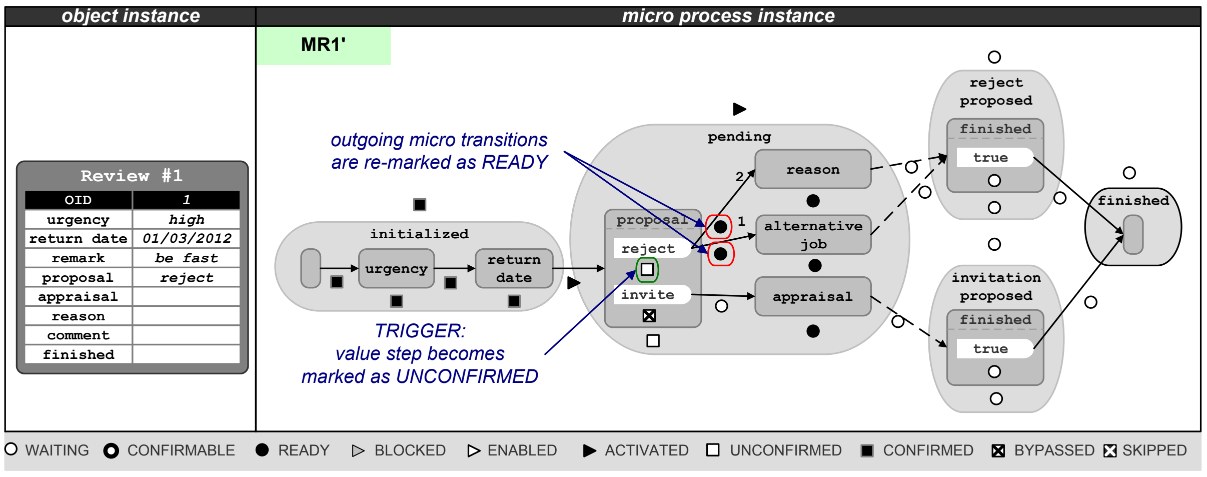


Figure 8.31: Applying Marking Rule MR1'

Altogether, Fig. 8.32 illustrates the various rules for handling value-specific micro steps and their interdependencies. The depicted *flow of rules* differs from the handling of atomic micro steps: First, regarding Marking Rule MR1, micro transitions starting from value steps currently marked as UNCONFIRMED must change their marking to READY.[3] Second, when a micro step becomes marked as ENABLED (cf. Marking Rule MR2), corresponding value steps are also marked as ENABLED. Moreover, in the context of Reaction Rule RR2, value steps must be considered; i.e., a value step is marked as ACTIVATED if its predicate evaluates to true. Further, Reaction Rule RR2 is extended to adequately deal with value-specific micro steps. The latter may be marked as ACTIVATED, if at least one of its value steps is marked as ACTIVATED or the value-specific micro step itself is source of a micro transition (and a value for the attribute the micro step refers to is available). Opposed to this, if a value for the attribute, the value-specific

[3]Note that the same applies to micro transitions originating from a micro step itself.

micro step refers to, is set and it is not possible to activate the micro step (i.e., none of its value steps evaluates to true), the micro step changes its marking to BLOCKED. In turn, this marking indicates that an attribute value different from the current one must be set. In this context, we introduce Reaction Rule RR3 to re-mark respective micro steps as ENABLED. Finally, when applying Marking Rule MR5 to mark micro steps as UNCONFIRMED, corresponding value steps currently ACTIVATED must be marked as UNCONFIRMED as well. Value steps, in turn, which are still marked as ENABLED, change their marking to BYPASSED.

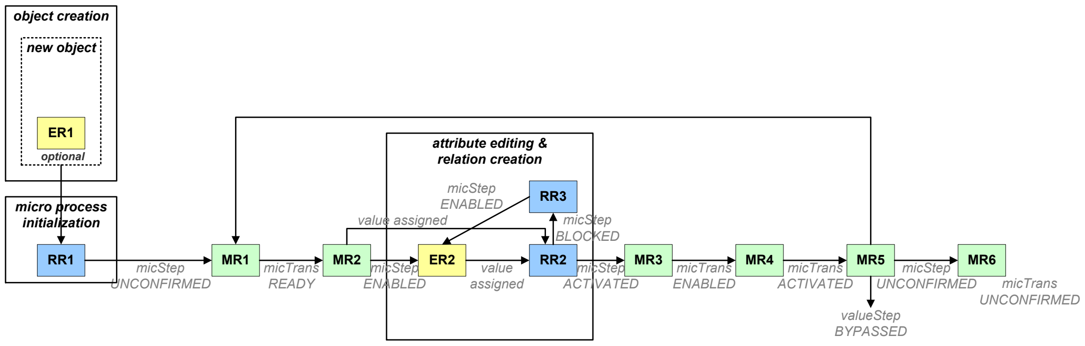


Figure 8.32: Rules considering value-specific micro steps

8.4.3 Non-deterministic Execution

We now discuss how non-deterministic micro process instances are executed. Thereby, we differentiate between the following scenarios: In general, non-deterministic execution paths exist in the context of value-specific micro steps. For example consider Fig. 8.33a, where it depends on the attribute value of the source micro step, which of the alternative paths will be selected (i.e., which value step will be activated at run-time). If only one value step is activated at run-time, all rules introduced in the context of deterministic execution and handling value-specific micro steps can be applied. In general, however, we also have to consider scenarios in which a micro step (or value step) has more than one outgoing micro transition (cf. Fig. 8.33b); i.e., users may choose one attribute (or relation) mandatorily required. Here, it depends on the target micro step, which particular micro transition will be chosen. More precisely, an outgoing micro transition may be chosen, if a value for the attribute (or relation) the target micro step refers to is available (or this target micro step is empty). In this context, we must ensure that only one subsequent micro step may be reached (even if for multiple target micro steps respective values are available). To be able to cope with such scenarios, respective micro transitions are associated with different priorities at build-time. Based on them, only the micro transition having the highest priority (i.e., the lowest number) is selected at run-time. Note that value steps as well as the micro step itself may have outgoing micro transitions (cf. Fig. 8.33c). Here, micro transitions originating from both a value and a micro step may be simultaneously chosen. This may lead to the parallel activation of different states. In addition, more than one value step may be activated at a certain point during process execution. As example consider Fig. 8.33d.

Using Marking Rule MR1, outgoing (internal) micro transitions are marked as READY when their source micro step or source value step becomes marked as UNCONFIRMED. In turn, this triggers Marking Rule MR2. Accordingly, all target micro steps of the micro transitions currently

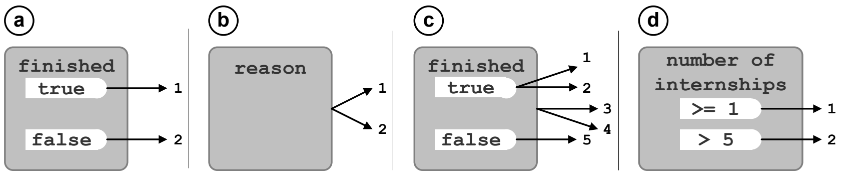


Figure 8.33: Non-deterministic execution paths

marked as READY are re-marked as ENABLED. This indicates that a value for the attribute (or relation) the micro step refers to is now mandatorily required.

Example 8.19 (Applying Marking Rule MR2 for several micro steps):
Consider Fig. 8.34. If value "reject" is assigned to attribute `proposal` either a value for attribute `reason` (i.e., to reach micro step `reason`) or relation `alternative job` (i.e., to reach micro step `alternative job`) is mandatorily required. Hence, corresponding input fields are highlighted with a red star.

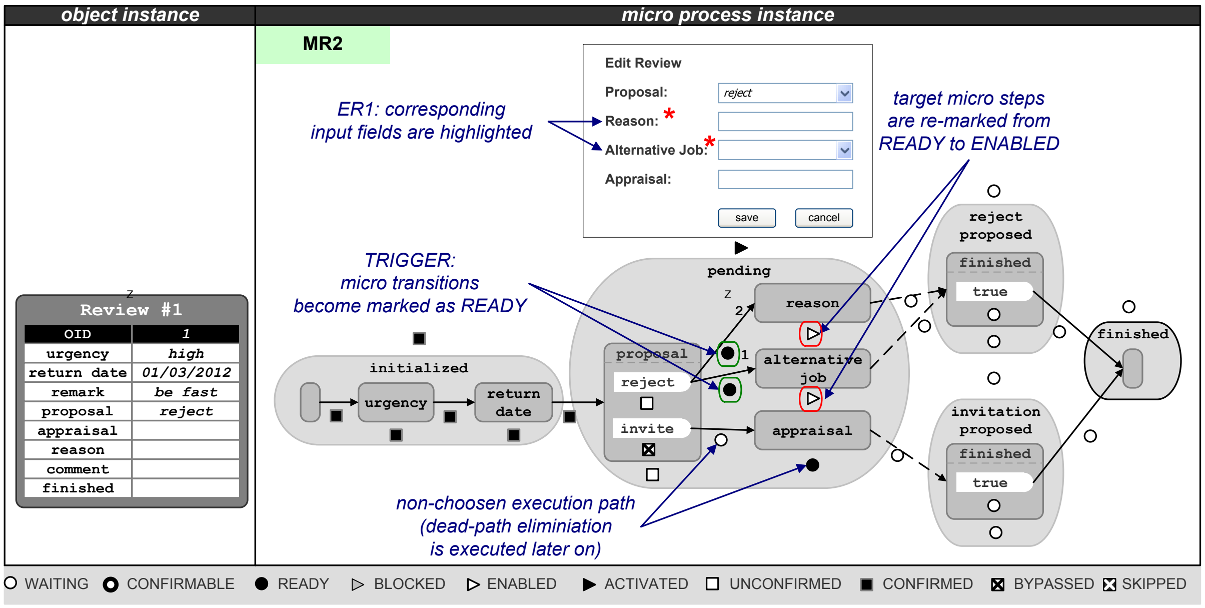


Figure 8.34: Applying Marking Rule MR2 for several micro steps

If an attribute value (or relation) is only available for one subsequent micro step (i.e., this micro step is marked as ACTIVATED according to Reaction Rule RR2), only one alternative execution path is chosen. More precisely, when applying Marking Rule MR3, the incoming micro transition of a micro step which is currently marked as ACTIVATED, is re-marked from READY to ENABLED. If only one micro transition becomes ENABLED, Marking Rule MR4 immediately re-marks this micro transition as ACTIVATED. Furthermore, all other execution paths, which are still marked as READY, are re-marked as BYPASSED indicating that corresponding micro transitions belong to alternative execution paths currently skipped (since there is no value for the attribute or relation their target micro step refers to). We extend System Rule MR4 (cf. Marking Rule MR4').

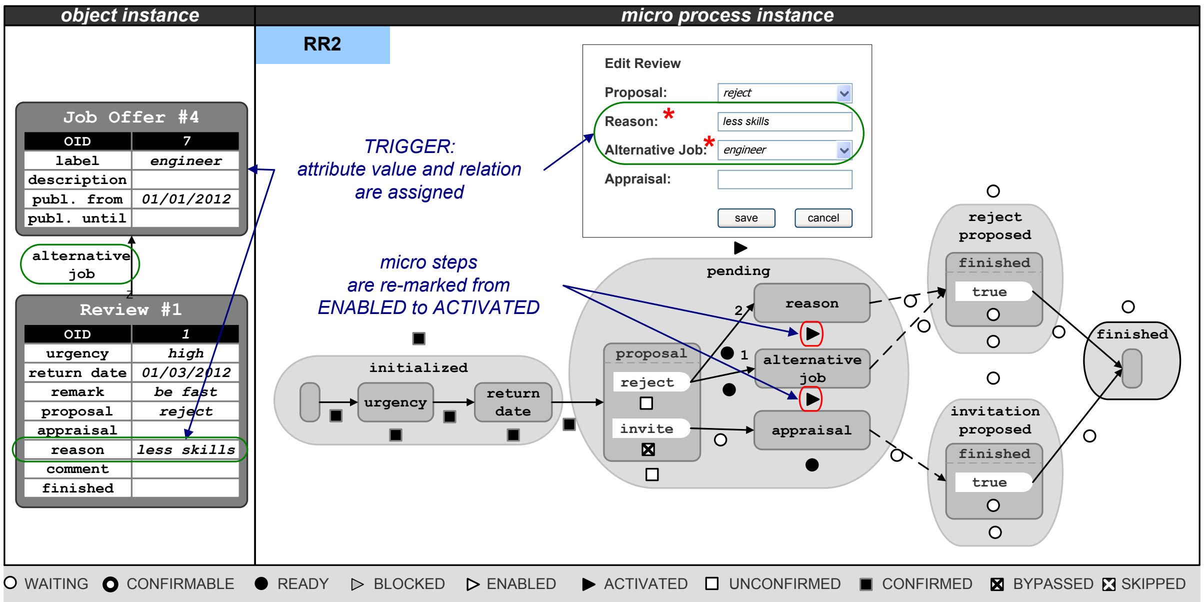
Marking Rule (MR4’: Marking micro transitions as BYPASSED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

a) see MR4 in Sect. 8.4.1

b) $\exists$ micTrans’ $\in$ MicTransSet with $M_{MicTrans}$(micTrans’) = ACTIVATED:
  $\forall$ micTrans $\in$ MicTransSet with $M_{MicTrans}$(micTrans) = READY:
    $M_{MicTrans}$(micTrans) := BYPASSED;
i.e., if a micro transition is marked as ACTIVATED all other micro transitions currently marked as READY are re-marked as BYPASSED.

Furthermore, we must consider the case for which values of the attributes (or relations) of more than one subsequent micro step are available.

Example 8.20 (Applying Reaction Rule RR2 for several micro steps):
Consider Fig. 8.35. Values for both attribute `reason` and for relation `alternative job` are available. Consequently, when applying Reaction Rule RR2, the corresponding micro steps are marked as ACTIVATED.



Job Offer #4

OID	7
label	engineer
description	
publ. from	01/01/2012
publ. until	

Review #1

OID	1
urgency	high
return date	01/03/2012
remark	be fast
proposal	reject
appraisal	
reason	less skills
comment	
finished	

Edit Review

Proposal:	reject
Reason: *	less skills
Alternative Job: *	engineer
Appraisal:	

save cancel

Figure 8.35: Applying Reaction Rule RR2 to several micro steps

If more than one micro step is marked as ACTIVATED at a certain point during micro process execution, Marking Rule MR3 is applied several times for each of these micro steps.

Example 8.21 (Applying Marking Rule MR3 for several micro steps):
Consider Fig. 8.36. When applying Marking Rule MR3, the micro transition between value step `reject` and micro step `reason` is marked as ENABLED. The same applies to the micro transition linking value step `reject` and micro step `alternative job`.

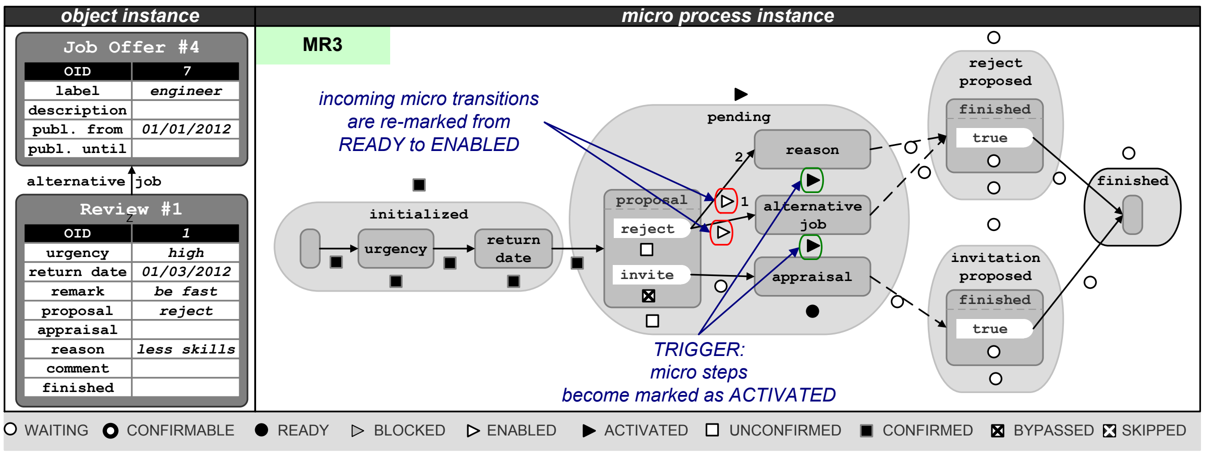


Figure 8.36: Applying Marking Rule MR3 for several micro steps

In the given context, we must consider the priorities of enabled micro transitions to determine which of the latter shall be activated. Consider Marking Rule MR4”, which will be triggered if more than one micro transition becomes marked as ENABLED. According to this rule, from the ENABLED micro transitions only the one having the highest priority[4] is re-marked as ACTIVATED. In turn, all other ones are re-marked from ENABLED to BYPASSED (cf. Marking Rule MR4’). In addition, these micro transitions, which are currently marked as READY, are re-marked as BYPASSED. Note that the latter cannot be reached since the value of the attribute (or relation) the target micro step refers to is not available.

Marking Rule (MR4”: Priority Evaluation):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

a) see MR4 in Sect. 8.4.1

b) see MR4’ in Sect. 8.4.3

c) Let $\text{MicTrans}_{\text{ENABLED}} := \{\text{micTrans} \in \text{MicTransSet} \mid M_{\text{MicTrans}}(\text{micTrans}) = \text{ENABLED}\}$.
Let further micTrans* $\in \text{MicTrans}_{\text{ENABLED}}$ be the enabled micro transition having the highest priority;
i.e., micTrans*.priority $\leq$ micTrans.priority $\forall$ micTrans $\in \text{MicTrans}_{\text{ENABLED}}$. Then:

$$M_{\text{MicTrans}}(\text{micTrans}) := \begin{cases} \text{ACTIVATED}, & \text{micTrans} = \text{micTrans*} \\ \text{BYPASSED}, & \text{micTrans} \in \text{MicTrans}_{\text{ENABLED}} - \{\text{micTrans*}\} \end{cases}$$

i.e., the micro transition with the highest priority (i.e., lowest number) is marked as ACTIVATED, where all other ones are marked as BYPASSED.

Example 8.22 (Applying Marking Rule MR4” for priority evaluation):
Consider Fig. 8.37. Only the micro transition between value step `reject` and micro step `alternative job` is marked as ACTIVATED. By contrast, due to its lower priority, the micro transition between value step `reject` and micro step `reason`, is marked as BYPASSED.

[4]Note that the lowest value represents the highest priority.

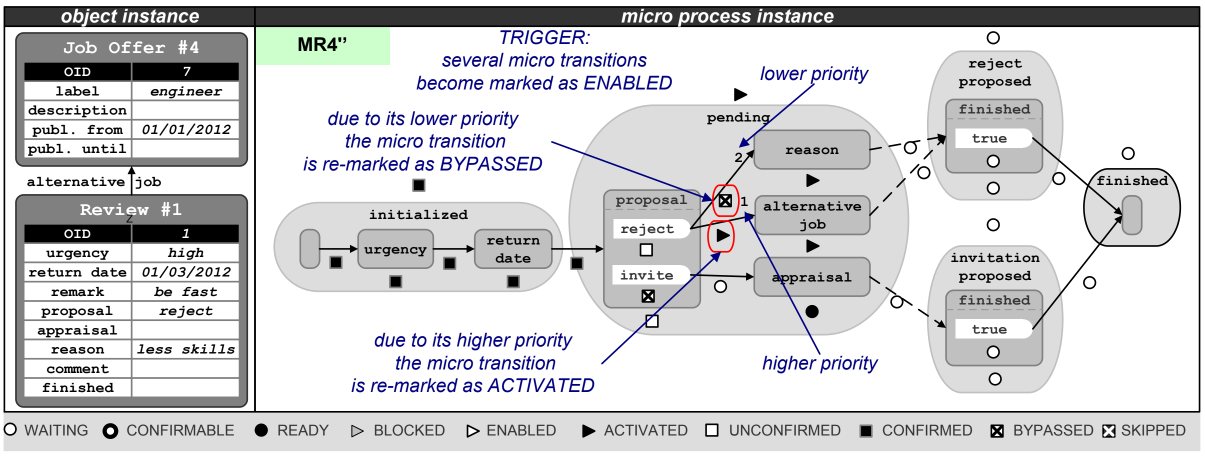


Figure 8.37: Applying Marking Rule MR4” for priority evaluation

After applying Marking Rule MR4, Marking Rule MR5 is triggered and the target micro step of the micro transition marked as ACTIVATED is re-marked as UNCONFIRMED. Following this, according to Marking Rule MR6, the incoming micro transition of a micro step currently marked as UNCONFIRMED is re-marked from ACTIVATED to UNCONFIRMED as well.

Altogether, to execute non-deterministic micro process instances, we extended Marking Rule MR4 in two ways (cf. Fig. 8.38). First, all micro transitions currently marked as READY are re-marked as BYPASSED in case another micro transition is ACTIVATED. Note that these micro transitions cannot fire because the attribute value (or relation) required to activate their target micro step is missing. Second, it is possible that more than one micro transition may fire and hence is marked as ENABLED. For this case, only the micro transition with the highest priority (i.e., having the lowest value) is ACTIVATED. All other ones are marked as BYPASSED.

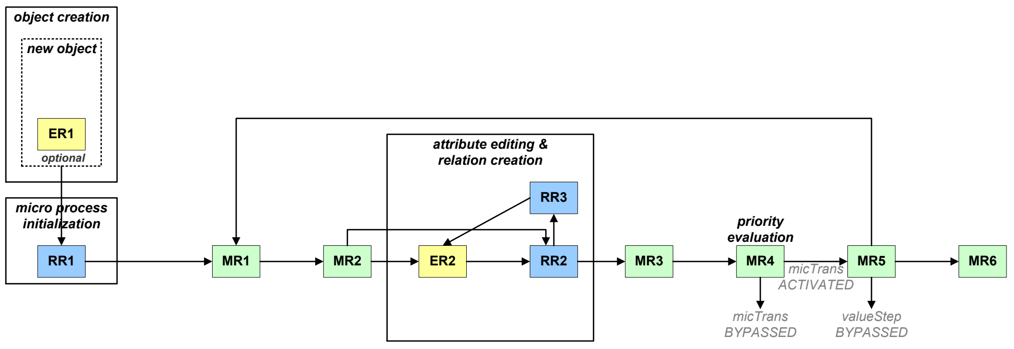


Figure 8.38: Rules for non-deterministic micro process execution

8.4.4 Internal Dead-path Elimination

Especially, when coordinating the execution of multiple micro process instances (cf. Chap. 10), we must be able to identify those states that can no longer be reached (since an alternative execution path was selected). To identify them, different kinds of dead-path eliminations are supported. In particular, we mark all micro steps, micro transitions, and states not selected during micro process execution accordingly. However, we must consider that values of attributes (or relations) may be changed as long as the state the corresponding micro steps belong to is still ACTIVATED.[5] For this purpose, we differentiate between *external* and *internal dead-path elimination*. Regarding the latter, we mark all micro steps and micro transitions belonging to the currently ACTIVATED state as BYPASSED if another execution path within the same state is chosen. Consider Marking Rules MR4 and MR5. Applying the latter, all value steps for which the corresponding predicate evaluates to false are marked as BYPASSED, when the micro step the value step belongs to is marked as UNCONFIRMED. As example consider value step `invite` in Fig. 8.39.

In turn, when applying Marking Rule MR4, all outgoing micro transitions that cannot be activated are marked as BYPASSED (e.g., the micro transition between value step `reject` and micro step `reason` in Fig. 8.40).

Both value steps and micro transitions currently marked as BYPASSED (cf. Marking Rule MR4 and MR5) constitute the starting point of an internal dead-path elimination. For the latter, two additional marking rules are needed. First, *Marking Rule MR7* remarks those micro transitions as BYPASSED whose source value or source micro step is currently marked as BYPASSED. However, only internal micro transitions are considered; i.e., Marking Rule MR7 only addresses micro transitions whose source and target micro steps belong to the same state (i.e., the state currently marked as ACTIVATED).

Marking Rule (MR7: Marking micro transitions as BYPASSED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Then:

a) ∀ micStep ∈ MicStepSet with $M_{MicStep}$(micStep) = BYPASSED:
   ∀ micTrans ∈ outtrans(micStep) with isexternal(micTrans) = FALSE ∧ $M_{MicTrans}$(micTrans) = WAITING:
      $M_{MicTrans}$(micTrans) := BYPASSED;
   i.e., if a micro step is marked as BYPASSED, all internal micro transitions outgoing from it are marked as BYPASSED as well.

b) ∀ micStep=(ref, ValueSteps) ∈ MicStepSet with
   $M_{MicStep}$(micStep) = BYPASSED ∨ $M_{MicStep}$(micStep) = UNCONFIRMED:
      ∀ valueStep ∈ ValueSteps with $M_{MicStep}$(valueStep) = BYPASSED:
         ∀ micTrans ∈ outtrans(valueStep) with isexternal(micTrans) = FALSE
         ∧ $M_{MicTrans}$(micTrans) = WAITING:
            $M_{MicTrans}$(micTrans) := BYPASSED;
   i.e., if a value step is marked as BYPASSED, all internal micro transitions outgoing from this value step are marked as BYPASSED as well.

Second, according to *Marking Rule MR8*, if all incoming micro transitions of any micro step, belonging to the currently ACTIVATED state, are marked as BYPASSED, this micro step will be marked as BYPASSED as well. Note that Marking Rules MR7 and MR8 trigger each other iteratively. Finally, an internal dead-path elimination will terminate if the micro transitions originating from a BYPASSED micro step are external ones.

[5]Note that this flexibility is required in practice and such value changes are therefore tolerated.

Marking Rule (MR8: Marking micro steps as BYPASSED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

$\forall$ micStep $\in$ MicStepSet with $M_{MicStep}$(micStep) $\in$ {READY, ENABLED, BLOCKED}:
$\forall$ micTrans $\in$ intrans(micStep) with $M_{MicTrans}$(micTrans) = BYPASSED:
$M_{MicStep}$(micStep) := BYPASSED;
i.e., if all incoming micro transitions of a micro step are marked as BYPASSED, it will be marked as BYPASSED as well.

Example 8.23 (Internal dead-path elimination):
Consider Fig. 8.39. Value step `invite`, which is marked as BYPASSED, triggers Marking Rule MR7. As a consequence, the micro transition between value step `invite` and micro step `appraisal` is re-marked from WAITING to BYPASSED. Consider now micro steps `reason` and `appraisal`, whose incoming micro transitions are both marked as BYPASSED. In turn, this triggers Marking Rule MR8, according to which both micro steps are marked as BYPASSED as well (cf. Fig. 8.40) Finally, since all micro transitions originating from micro steps `reason` and `appraisal` are external ones, Marking Rule MR7 is not re-triggered; i.e., the internal dead-path elimination terminates.

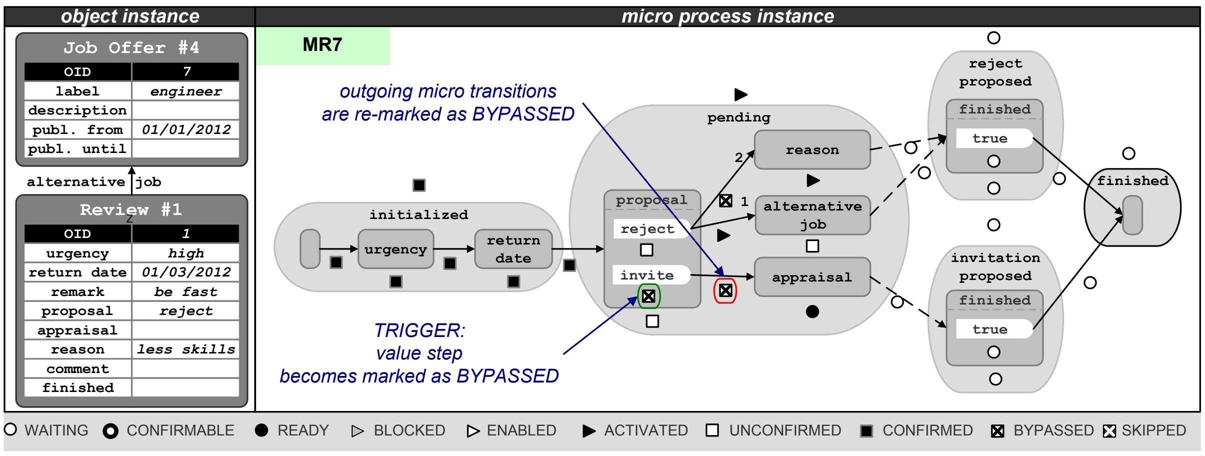


Figure 8.39: Applying Marking Rule MR7

Fig. 8.41 summarizes the rules relevant for an internal dead-path elimination. Furthermore, Fig. 8.41 shows which markings trigger the respective rules. Marking Rule MR8 is triggered when all incoming micro transitions of a micro step are marked as BYPASSED. In turn, micro transitions are marked as BYPASSED when executing Marking Rule MR4. Moreover, when applying Marking Rule MR5, value steps may be marked as BYPASSED. In turn, the latter triggers the execution of Marking Rule MR7, which then marks all micro transitions originating from these value steps as BYPASSED. Finally, Marking Rules MR7 and MR8 are iteratively applied.

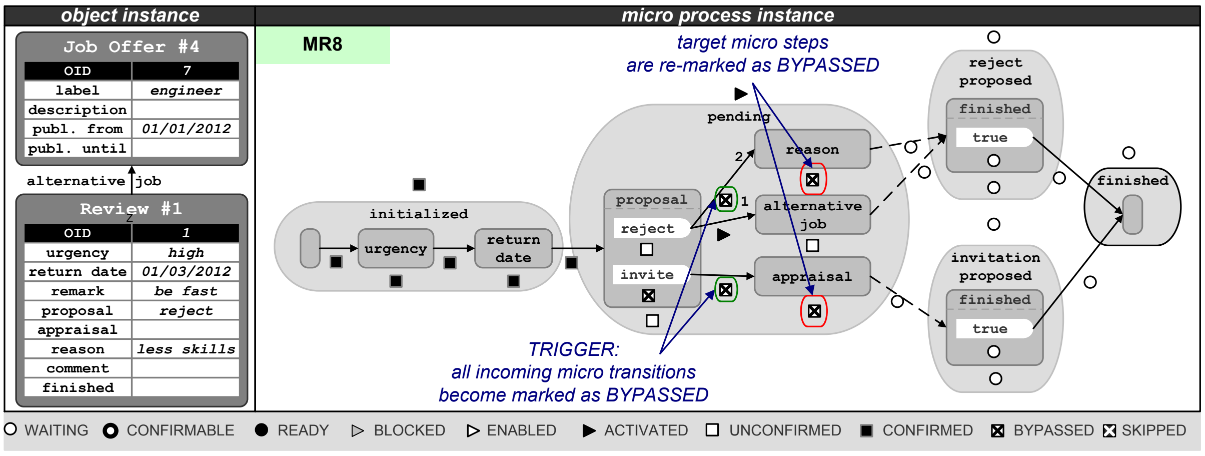


Figure 8.40: Applying Marking Rule MR8

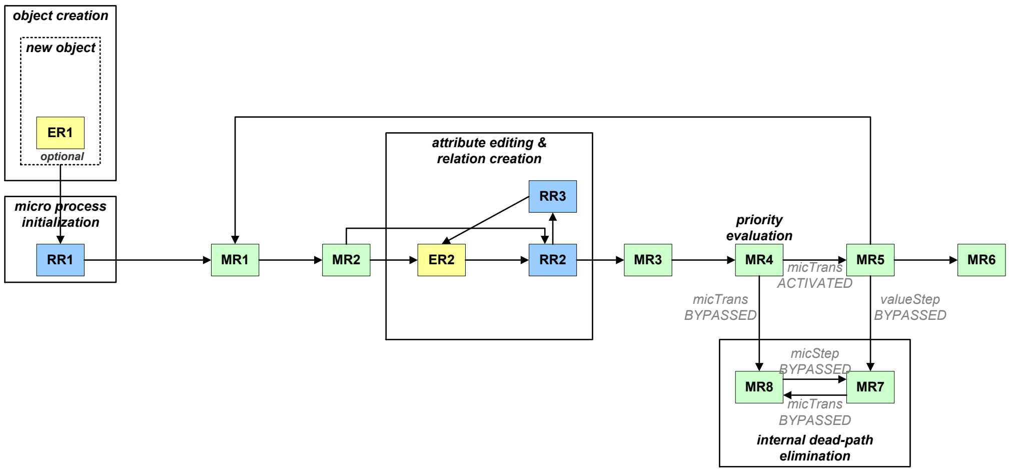


Figure 8.41: Rules for an internal dead-path elimination

8.4.5 Handling Empty Micro Steps

So far, we have considered atomic and value-specific micro steps. This section extends these considerations by additionally considering *empty micro steps* (e.g., end micro steps).

Example 8.24 (Applying Marking Rule MR2 for empty micro steps):
Consider Fig. 8.42. State `finished` comprises an empty micro step, which is marked as ENABLED when one of its incoming micro transitions becomes marked as READY.

Marking ENABLED indicates that a value for the attribute (or relation) the respective micro step refers to is mandatorily required to proceed with micro process execution. However, empty micro steps do not refer to any attribute or relation, and are hence immediately marked as ACTIVATED. For this purpose, we extend Reaction Rule RR2 to RR2'.

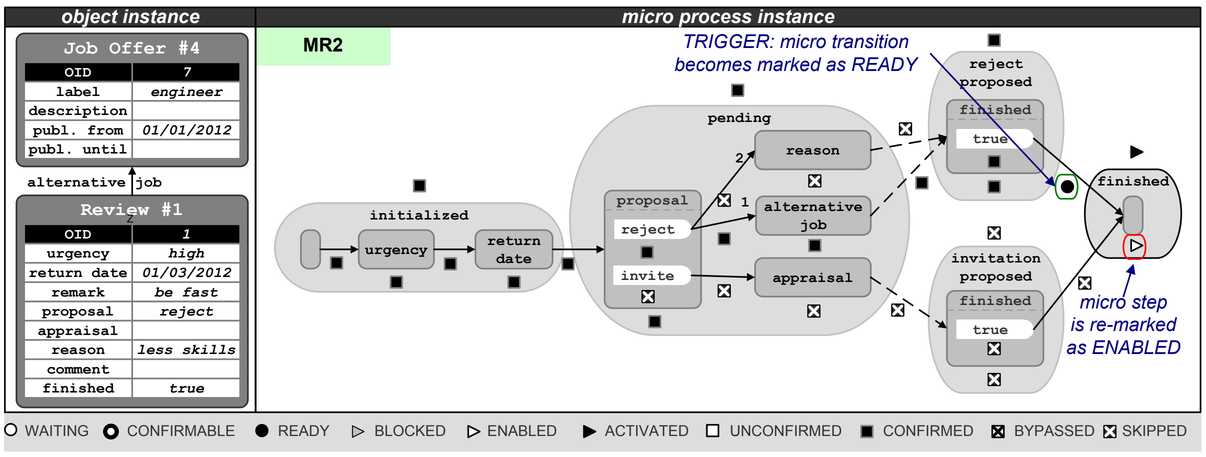


Figure 8.42: Applying Marking Rule MR2 to empty micro steps

Reaction Rule (RR2'''': Marking empty micro steps as ACTIVATED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Then:

a) see RR2 in Sect. 8.4.1

b)-e) see RR2' in Sect. 8.4.2

f) ∀ micStep=(ref, ValueSteps) ∈ MicStepSet with ref = NULL ∧ $M_{MicStep}$(micStep) = ENABLED:
$M_{MicStep}$(micStep) := ACTIVATED;
i.e., all empty micro steps that become marked as ENABLED are immediately re-marked as ACTIVATED.

Example 8.25 (Applying Reaction Rule RR2'''' for empty micro steps):
Consider Fig. 8.43. According to Reaction Rule RR2'''', the empty micro step of state `finished` will be immediately marked as ACTIVATED.

Execution Rule ER2 is not applied in the context of empty micro steps. Instead, Reaction Rule RR2 is immediately triggered (cf. Fig. 8.44). If an empty micro step is marked as ACTIVATED, Marking Rule MR3 will be triggered followed by the already introduced sequence of rules.

8.4.6 Re-assigning Attribute Values and Relations

When a state becomes enabled during the processing of an object instance, a corresponding form-based activity for entering the required attribute values (or relations) is automatically generated (cf. Fig. 8.45). In particular, for each micro step of the currently activated state, a corresponding input field is generated, enabling the assignment of the corresponding attribute value (or relation). These input fields must then be filled out by the responsible user who is assigned to the respective state.

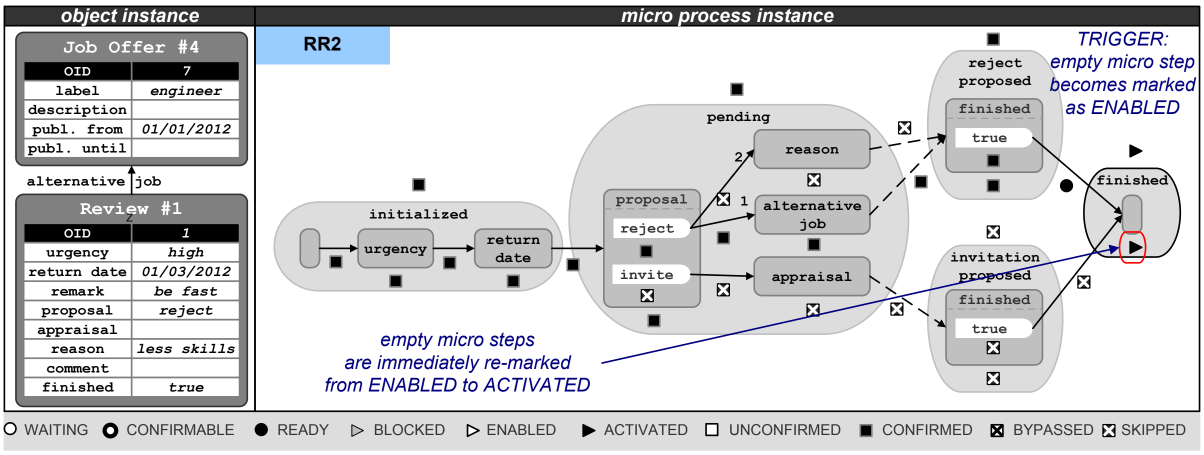


Figure 8.43: Applying Reaction Rule RR2"" to empty micro steps

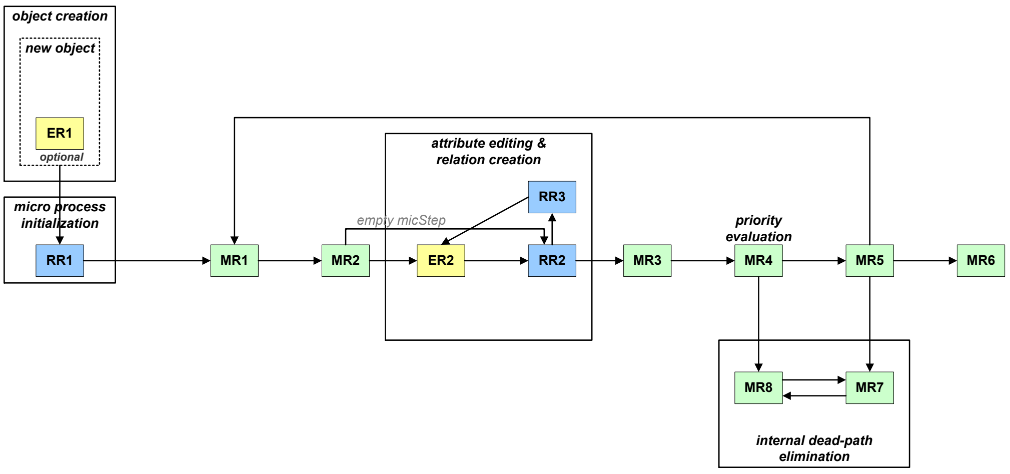


Figure 8.44: Rules for empty micro steps

Example 8.26 (Executing state pending):
Consider Fig. 8.45. Value step `reject` and micro step `alternative job` are both marked as UNCONFIRMED; i.e., the required attribute values and relations are available – attribute `proposal` has value "reject" and relation `alternative job` refers to the "engineer" object instance. When applying an internal dead-path elimination, all other micro steps (i.e., `reason` and `appraisal`), value step `invite`, and micro transitions (i.e., the micro transition between `reject` and `reason` and the one between `invite` and `appraisal`) are marked as BYPASSED.

As long as a state change has not been triggered, the responsible user may still change attribute settings. For example, he might assign value "invite" instead of value "reject" to attribute `proposal`. To provide this flexibility, the ACTIVATED state needs to be internally reset. More precisely, the micro process instance is reset to the first micro step of the currently activated state. To detect when value changes are applied during micro process execution, each (atomic or value-specific) micro step comprises a *trace* saving the value of the referenced attribute or

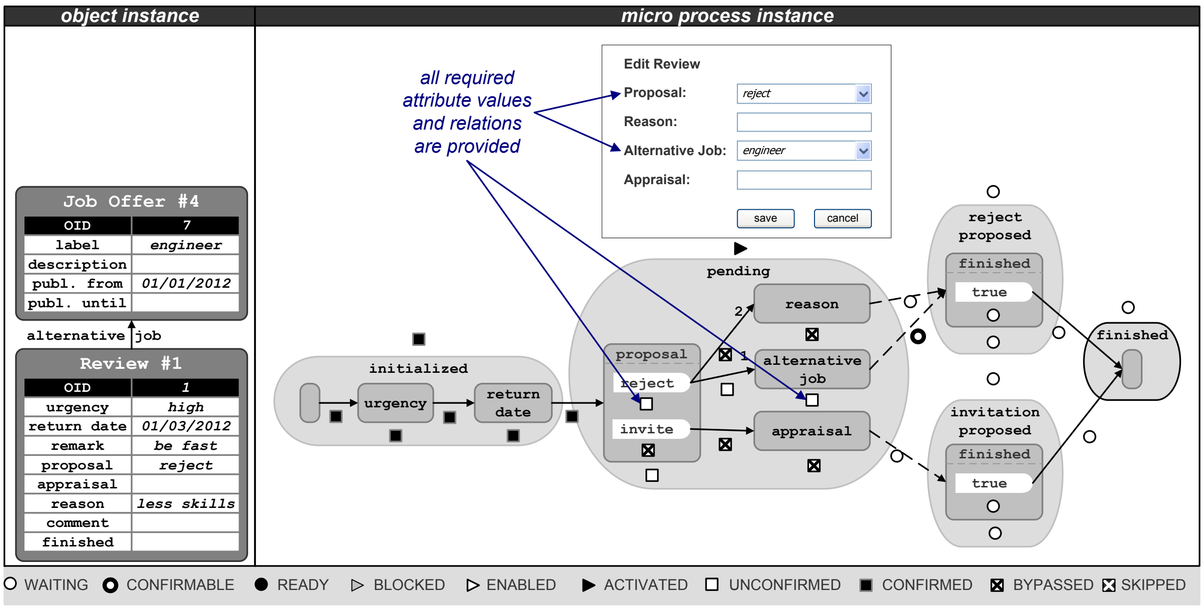


Figure 8.45: Micro process execution in state pending

relation when the respective micro step is marked as UNCONFIRMED (cf. Def. 26).

Definition 26 (Trace):
Let ds=(dm, OSet, RelSet) be a data structure with OSet be the set of all object instances, Attr the set of all attributes, and AttrValues the set of all possible attribute values.
Further, let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Finally, let micStep = (ref, ValueSteps) $\in$ MicStepSet with ref $\neq$ NULL be a value-specific or atomic micro step. Then:

trace: MicStepSet $\mapsto$ **AttrValues** $\cup$ **OSet** is a function that determines for each micro step the value (or object reference) assigned to the corresponding attribute (or relation) when the micro step is re-marked from ACTIVATED to UNCONFIRMED. In detail:

- trace(micStep) = attrValue, if ref = (attrType, attrValue) $\in$ Attr $\wedge$ $M_{MicStep}$(micStep) = UNCONFIRMED
  i.e., in this case function trace determines for each micro process step the value assigned to the corresponding attribute, when the micro step becomes re-marked from ACTIVATED to UNCONFIRMED; i.e., trace(micStep) = attrValue.
- trace(micStep) = toid, if ref = (relType, soid, toid) $\in$ RelSet $\wedge$ $M_{MicStep}$(micStep) = UNCONFIRED
  i.e., in this case trace is a function that determines for each micro process step the reference assigned to the corresponding relation, when the micro step becomes re-marked from ACTIVATED to UNCONFIRMED; i.e., trace(micStep) = toid.

According to *Reaction Rule RR4*, an internal reset is triggered if the currently assigned value of the attribute or relation is not the same as saved by the corresponding trace. All micro and value steps of the currently ACTIVATED state are marked as READY, regardless whether they are currently marked as UNCONFIRMED or BYPASSED (cf. Reaction Rule RR4a+b). In addition, all internal micro transitions, for which the source and target micro (or value) steps belong to the currently activated state, are marked as WAITING (cf. Reaction Rule RR4c). In turn, the external micro transition, which was used to activate the current state, changes its marking from

CONFIRMED to READY (cf. Reaction Rule RR4d). If the currently activated state corresponds to the start state of the micro process instance, the corresponding start micro step is marked as UNCONFIRMED and its outgoing micro transitions are marked as READY (cf. Reaction Rule RR4e). Finally, all external micro transitions, used to enable a subsequent state, are marked as WAITING (cf. Reaction Rule RR4f).

Reaction Rule (RR4: Internal reset):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Further, let state=(name, sMicStepSet) ∈ StateSet be a state with M_{State}(state) = ACTIVATED. Then:

∃ micStep = (ref, ValueSteps) ∈ sMicStepSet with $M_{MicStep}$(micStep) = UNCONFIRMED ∧
(ref ∈ Attr ∧ attrValue ≠ trace(micStep) ∨ ref ∈ RelSet ∧ ref.toid ≠ trace(micStep), ⇒

a) ∀ micStep ∈ sMicStepSet: $M_{MicStep}$(micStep) := READY;
i.e., all micro steps of the currently activated state are marked as READY.

b) ∀ micStep = (ref, ValueSteps) ∈ sMicStepSet: ∀ valueStep ∈ ValueSteps: $M_{MicStep}$(valueStep) := READY;
i.e., all value steps of the currently activated state are marked as READY.

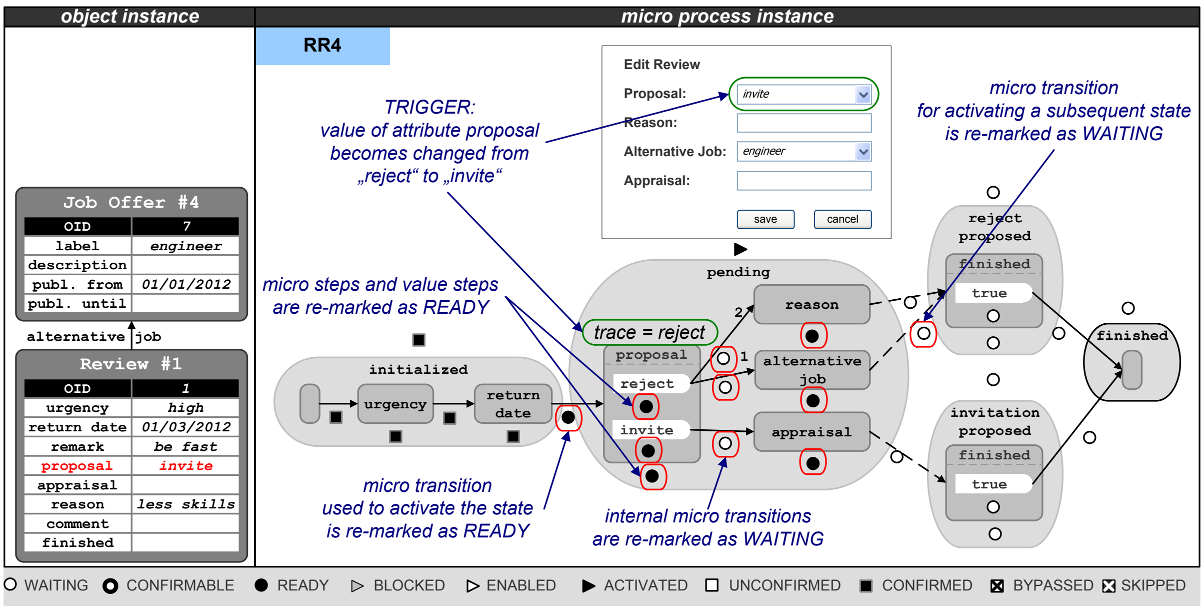
c) ∀ micTrans=(source, target, priority) ∈ MicTransSet with {source, target} ⊆ sMicStepSet:
$M_{MicTrans}$(micTrans) := WAITING;
i.e., all internal micro transitions of the currently activated state are marked as WAITING.

d) state != $startState_{micProcInstance}$: micTrans=(source, target, priority) ∈ MicTransSet with
source ∉ sMicStepSet ∧ target ∈ sMicStepSet ∧ $M_{MicTrans}$(micTrans) = CONFIRMED:
$M_{MicTrans}$(micTrans) := READY;
i.e., if the currently activated state does not correspond to the start state of the micro process instance, the marking of the external micro transition whose target micro step belongs to the currently activated state is reset from CONFIRMED to READY.

e) state = $startState_{micProcInstance}$:
$M_{micStep}$($startMicStep_{micProcInstance}$) := UNCONFIRMED ∧
∀ micTrans ∈ MicTransSet with micTrans.sourceStep = startMicStep:
$M_{MicTrans}$(micTrans) := READY;
i.e., if the currently activated state corresponds to the start state of the micro process instance, the corresponding start micro step is re-marked as UNCONFIRMED and the micro transitions originating from it are marked as READY.

f) micTrans=(source,target,prio) ∈ MicTransSet with source ∈ sMicStepSet ∧ target ∉ sMicStepSet
∧ $M_{MicTrans}$(micTrans) = CONFIRMABLE:
$M_{MicTrans}$(micTrans) := WAITING;
i.e., all external micro transitions whose source micro step (or value step) belongs to the currently activated state is marked as WAITING.

Example 8.27 (Applying Reaction Rule RR4):
Consider Fig. 8.46. The trace of micro step `proposal` is "reject" since this value was used during micro process execution. However, the responsible user changed this value afterwards; i.e., value "invite" is now assigned to attribute `proposal`. When performing the required internal reset, micro steps `proposal`, `reason`, `alternative job`, and `appraisal` as well as value steps `reject` and `invite` are re-marked as READY. In addition, the internal micro transitions connecting value steps `reject` and `invite` with the respective micro steps `reason`, `alternative job`, or `appraisal` are marked as WAITING. In turn, the external micro transition connecting micro steps `return date` and `proposal` was used to activate state `pending`. Hence, this micro transition is re-marked as READY. Finally, the external micro transition between micro steps `alternative job` and `finished` (belonging to state `reject proposed`) is re-marked as WAITING. Before setting attribute value `proposal`, this micro transition was marked as CONFIRMABLE (cf. Fig. 8.45).


object instance
micro process instance
RR4
TRIGGER: value of attribute proposal becomes changed from „reject“ to „invite“
micro transition for activating a subsequent state is re-marked as WAITING
micro steps and value steps are re-marked as READY
micro transition used to activate the state is re-marked as READY
internal micro transitions are re-marked as WAITING
trace = reject
initialized
urgency
return date
pending
proposal
reject
invite
reason
alternative job
appraisal
reject proposed
invitation proposed
finished
true
alternative
job
WAITING
CONFIRMABLE
READY
BLOCKED
ENABLED
ACTIVATED
UNCONFIRMED
CONFIRMED
BYPASSED
SKIPPED

Edit Review

Proposal:	invite
Reason:	
Alternative Job:	engineer
Appraisal:	

save cancel

Job Offer #4

OID	7
label	engineer
description	
publ. from	01/01/2012
publ. until	

Review #1

OID	1
urgency	high
return date	01/03/2012
remark	be fast
proposal	invite
appraisal	
reason	less skills
comment	
finished	

Figure 8.46: Applying Reaction Rule RR4

After an internal reset, the re-initialized region of the micro process instance may be re-executed based on the already presented marking rules (cf. Fig. 8.47). Opposed to the first execution of this region, however, the different attribute values and relations are now considered leading to a dynamic adaption of the corresponding user form.

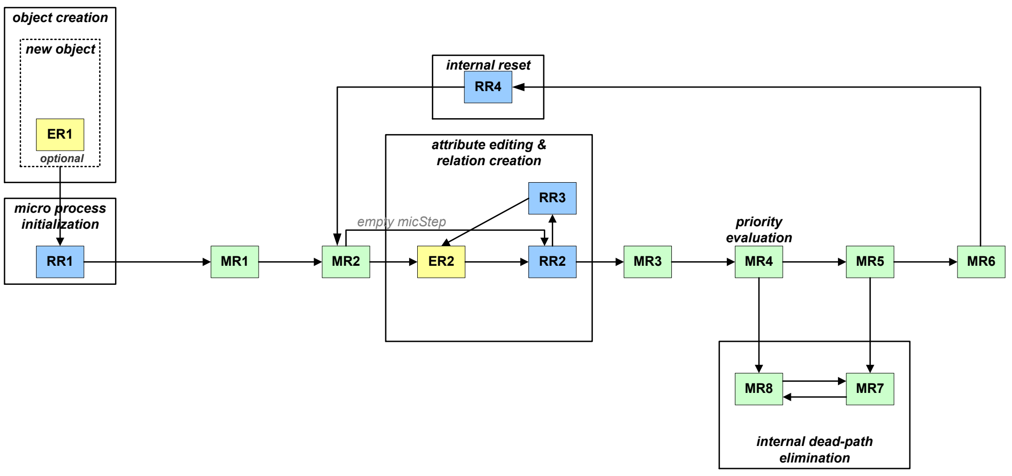


Figure 8.47: Rules for an internal reset

Example 8.28 (Applying Marking Rule MR2 after an internal reset):
Consider Fig. 8.48. When re-applying Marking Rule MR2, micro step `proposal` as well as its value steps `reject` and `invite` are first marked as ENABLED.

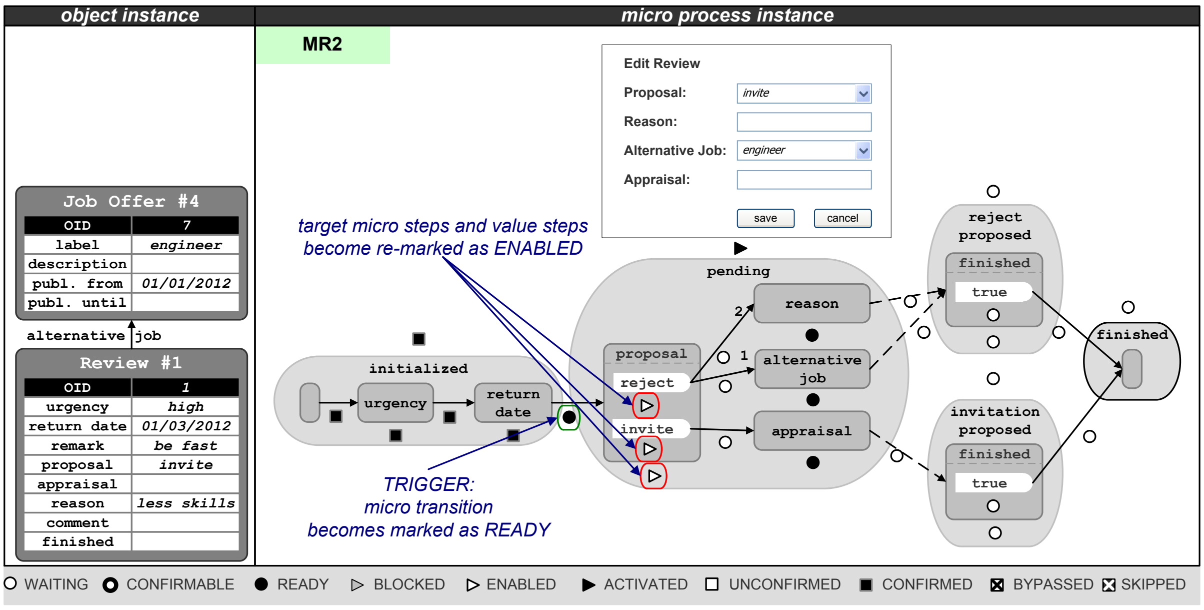


Figure 8.48: Applying Marking Rule MR2 after an internal reset

Example 8.29 (Applying Reaction Rule RR2 after an internal reset):
Consider the user form in Fig. 8.48. Since value "invite" is now assigned to attribute `proposal`, Reaction Rule RR2 is immediately triggered marking the corresponding value step `invite` as ACTIVATED (cf. Fig. 8.49). Since at least one of its value steps is marked as ACTIVATED, in addition, micro step `proposal` changes its marking from ENABLED to ACTIVATED as well.

Note that the re-execution of a particular region of a micro process instance might lead a dynamic adaption of the corresponding user form.

Example 8.30 (Applying Execution Rule ER2 for adapting user forms):
When applying the rules for a state-internal execution of a micro process instance, instead of micro steps `reason` and `alternative job`, micro step `appraisal` becomes ENABLED. In turn, this marking triggers Execution Rule ER2 leading to a dynamic adaption of the corresponding user form. Consider Fig. 8.50, where the input field belonging to micro step `appraisal` is marked as mandatory (using a red star).

8.5 State Changes

As discussed in Chapt. 7, a micro process comprises different states to coordinate the processing of an object instance among different users. More precisely, in different states the user forms generated for assigning required attribute values or relations are assigned to different users (depending on the specified user role assignment). Sect. 8.4 described the rules driving the state-internal executions of a micro process instance at run-time; i.e., the generation of state-specific user forms as well as their internal process logic. Opposed to this, this

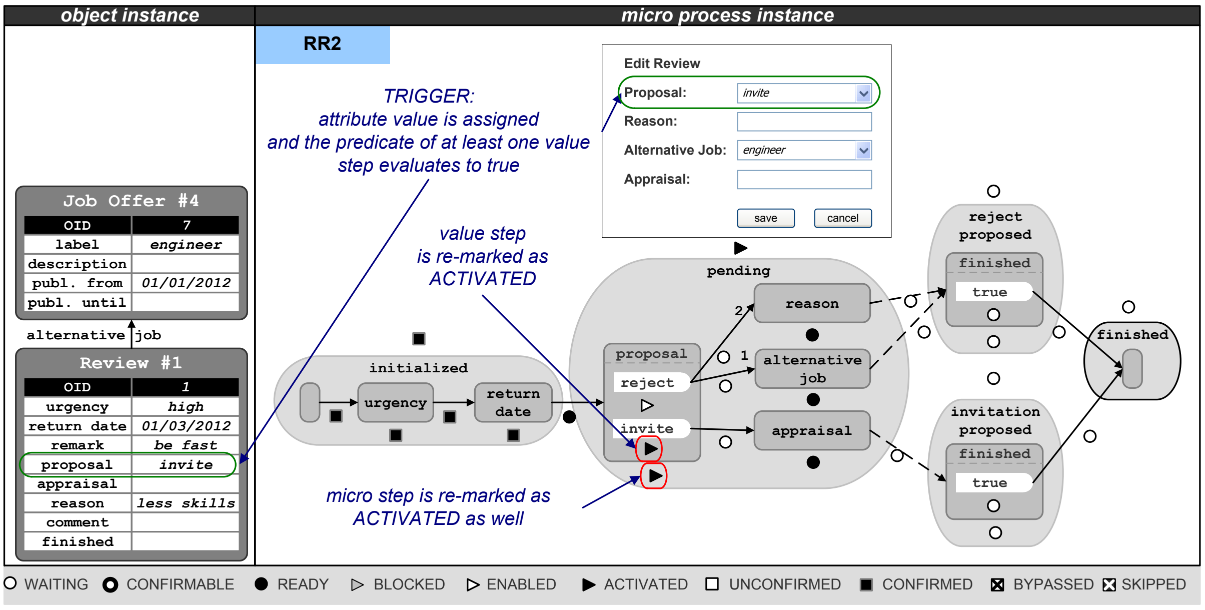


Figure 8.49: Applying Reaction Rule RR2 after an internal reset

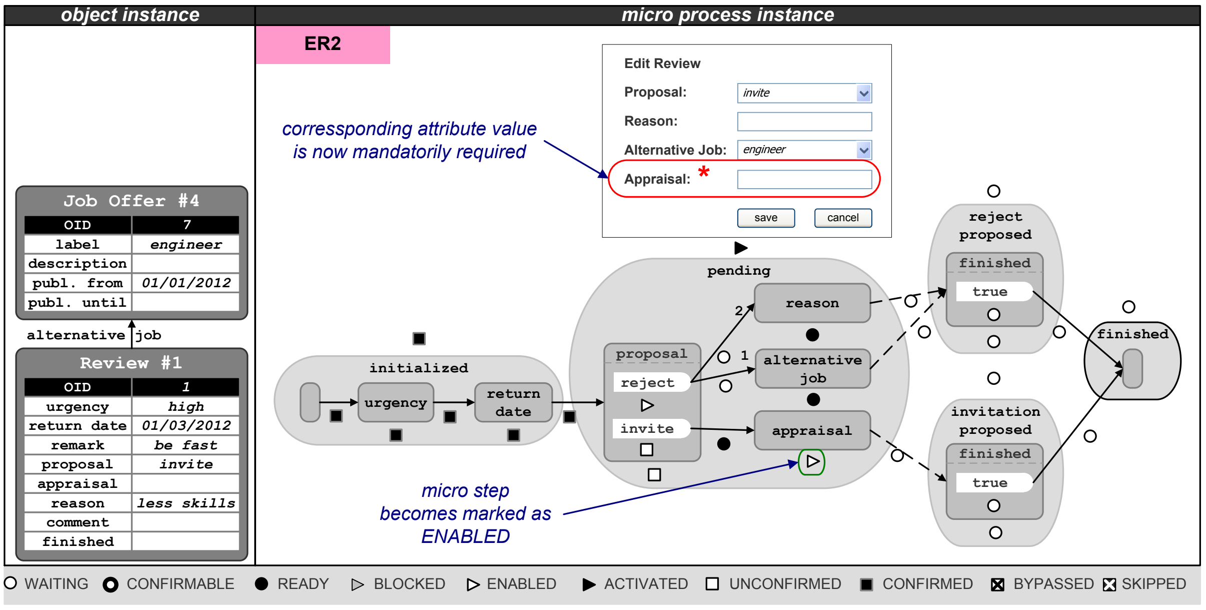


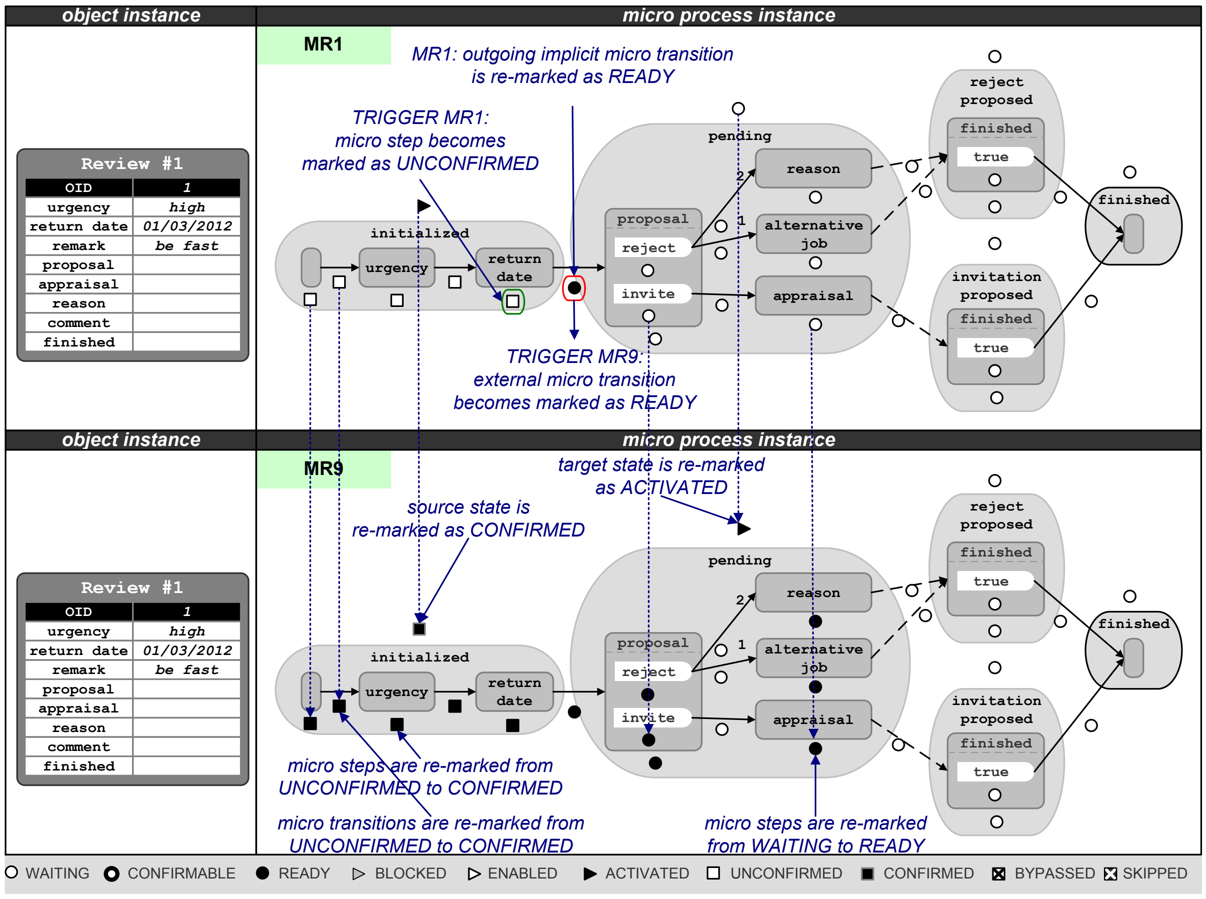
Figure 8.50: Applying Execution Rule ER2 for adapting user forms

section discusses when a *state change* occurs and how it is performed. In this context, we must differentiate between *implicit* and *explicit state changes*. The latter additionally require the commitment of a responsible user, whereas implicit state changes are automatically performed. Moreover, in certain situations users may decide which of the subsequent states shall be activated.

8.5.1 Implicit Micro Transitions

External micro transitions are either categorized as *implicit* (like internal micro transitions) or *explicit*. According to Marking Rule MR1, implicit micro transitions will be marked as READY when their source micro step (or value step) is re-marked from ACTIVATED to UNCONFIRMED.

Example 8.31 (Applying Marking Rule MR1 to external micro transitions):
Consider Fig. 8.51. Since micro step `return date` is marked as UNCONFIRMED, its outgoing implicit micro transition, which targets at micro step `proposal`, is marked as READY.



object instance
micro process instance
MR1
MR1: outgoing implicit micro transition is re-marked as READY
TRIGGER MR1: micro step becomes marked as UNCONFIRMED
TRIGGER MR9: external micro transition becomes marked as READY
Review #1
OID 1
urgency high
return date 01/03/2012
remark be fast
proposal
appraisal
reason
comment
finished
initialized
urgency
return date
pending
proposal
reject
invite
reason
alternative job
appraisal
reject proposed
invitation proposed
finished
true
MR9
target state is re-marked as ACTIVATED
source state is re-marked as CONFIRMED
micro steps are re-marked from UNCONFIRMED to CONFIRMED
micro transitions are re-marked from UNCONFIRMED to CONFIRMED
micro steps are re-marked from WAITING to READY
○ WAITING ◯ CONFIRMABLE ● READY ▷ BLOCKED ▷ ENABLED ▶ ACTIVATED □ UNCONFIRMED ■ CONFIRMED ⊠ BYPASSED ⊠ SKIPPED

Figure 8.51: Applying Marking Rules MR1 and MR9 to external micro transitions

A state change will be triggered, when an external micro transition becomes marked as READY indicating that its source micro step is reached (cf. *Marking Rule MR9*). First, the state currently marked as ACTIVATED is re-marked as CONFIRMED. This indicates that this state has been reached during micro process execution. In addition, all UNCONFIRMED micro steps, value steps, and micro transitions belonging to this state are re-marked as CONFIRMED. Opposed to this, all BYPASSED micro steps, value steps, and micro transitions of this state are re-marked as SKIPPED. Following this, the subsequent state (i.e., the state, to which the target micro step of the external micro transition, which is currently marked as READY, belongs becomes

ACTIVATED; i.e., the target state changes its marking from WAITING to ACTIVATED).
Finally, all micro and value steps of this newly ACTIVATED state are re-marked from WAITING to READY.

Marking Rule (MR9: State Change):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Further, let $state_1$ = ($name_1$, $sMicStepSet_1$) $\in$ StateSet with source $\in$ $sMicStepSet_1$ $\wedge$ $M_{State}(state_1)$ = ACTIVATED be the currently ACTIVATED state and $state_2$ = ($name_2$, $sMicStepSet_2$) $\in$ StateSet with target $\in$ $sMicStepSet_2$ $\wedge$ $M_{State}(state_2)$ = WAITING be its successor (i.e., the subsequent state that shall be activated next). Then:

micTrans $\in$ MicTransSet with $M_{MicTrans}$(micTrans) = READY $\wedge$ isexternal(micTrans) = TRUE, $\Rightarrow$

a) $M_{State}(state_1)$:= CONFIRMED;
i.e., the currently ACTIVATED state is re-marked as CONFIRMED.

b) $\forall$ micStep = (ref, ValueSteps) $\in$ $sMicStepSet_1$ with $M_{MicStep}$(micStep) = UNCONFIRMED:

- $M_{MicStep}$(micStep) := CONFIRMED;
i.e., all micro steps of the newly CONFIRMED state, which are currently marked as UNCONFIRMED, are re-marked as CONFIRMED.

- $\forall$ valueStep $\in$ ValueSteps:

$$M_{MicStep}(valueStep) := \begin{cases} \text{CONFIRMED} & \text{if } M_{MicStep}(valueStep) = \text{UNCONFIRMED} \\ \text{SKIPPED} & \text{if } M_{MicStep}(valueStep) = \text{BYPASSED} \end{cases}$$

i.e., all value steps of the newly CONFIRMED state, which are currently marked as UNCONFIRMED, are re-marked as CONFIRMED. In turn, all value steps currently marked as BYPASSED are re-marked as SKIPPED.

c) $\forall$ micStep = (ref, ValueSteps) $\in$ $sMicStepSet_1$ with $M_{MicStep}$(micStep) = BYPASSED:

- $M_{MicStep}$(micStep) := SKIPPED;
i.e., all micro steps of the newly CONFIRMED state, which are currently marked as BYPASSED, are re-marked as SKIPPED.

- $\forall$ valueStep $\in$ ValueSteps: $M_{MicStep}$(valueStep) := SKIPPED;
i.e., all value steps of the newly CONFIRMED state, which are currently marked as BYPASSED, are re-marked as SKIPPED.

d) $\forall$ micTrans $\in$ MicTransSet with {micTrans.source, micTrans.target} $\subseteq$ $sMicStepSet_1$:

$$M_{MicTrans}(micTrans) := \begin{cases} \text{CONFIRMED} & \text{if } M_{MicTrans}(micTrans) = \text{UNCONFIRMED} \\ \text{SKIPPED} & \text{if } M_{MicTrans}(micTrans) = \text{BYPASSED} \end{cases}$$

i.e., all micro transitions of the newly CONFIRMED state, which are currently marked as UNCONFIRMED, are re-marked as CONFIRMED. In turn, all micro transitions currently marked as BYPASSED are re-marked as SKIPPED.

e) $M_{State}(state_2)$:= ACTIVATED;
i.e., the subsequent state changes its marking from WAITING to ACTIVATED.

f) $\forall$ micStep = (ref, ValueSteps) $\in$ $sMicStepSet_2$ with $M_{MicStep}$(micStep) = WAITING:

- $M_{MicStep}$(micStep) := READY;
i.e., all micro steps of the newly ACTIVATED state, which are currently marked as WAITING, are re-marked as READY.

- $\forall$ valueStep $\in$ ValueSteps with $M_{MicStep}$(valueStep) = WAITING: $M_{MicStep}$(valueStep) := READY;
i.e., all value steps of the newly ACTIVATED state, which are currently marked as WAITING, are re-marked as READY.

Example 8.32 (Applying Marking Rule MR9 to implicit state changes):
Consider states `initialized` and `pending` in Fig. 8.51. If the external micro transition between micro steps `return date` and `proposal` becomes marked as READY (according to Marking Rule MR1), state `initialized` is re-marked as CONFIRMED and state `pending` changes its marking from WAITING to ACTIVATED. In addition, all micro steps and micro transitions of state `initialized` are re-marked as CONFIRMED as well; i.e., there exists no alternative execution path within this state. In addition, all micro and value steps of state `pending` are re-marked as READY.

As illustrated in Fig. 8.52, a state change is triggered as soon as an external (implicit) micro transition becomes marked as READY. Following this, the rules already introduced can be applied.

Example 8.33 (Continuing execution after a state change):
Since the micro transition between micro steps `return date` and `proposal` is marked as READY, Marking Rule MR2 is triggered remarking the target micro step (i.e., `proposal`) as well as its value steps `reject` and `invite` from READY to ENABLED. In turn, this indicates that a value for attribute `proposal` is now required; i.e., Execution Rule ER2 is triggered.

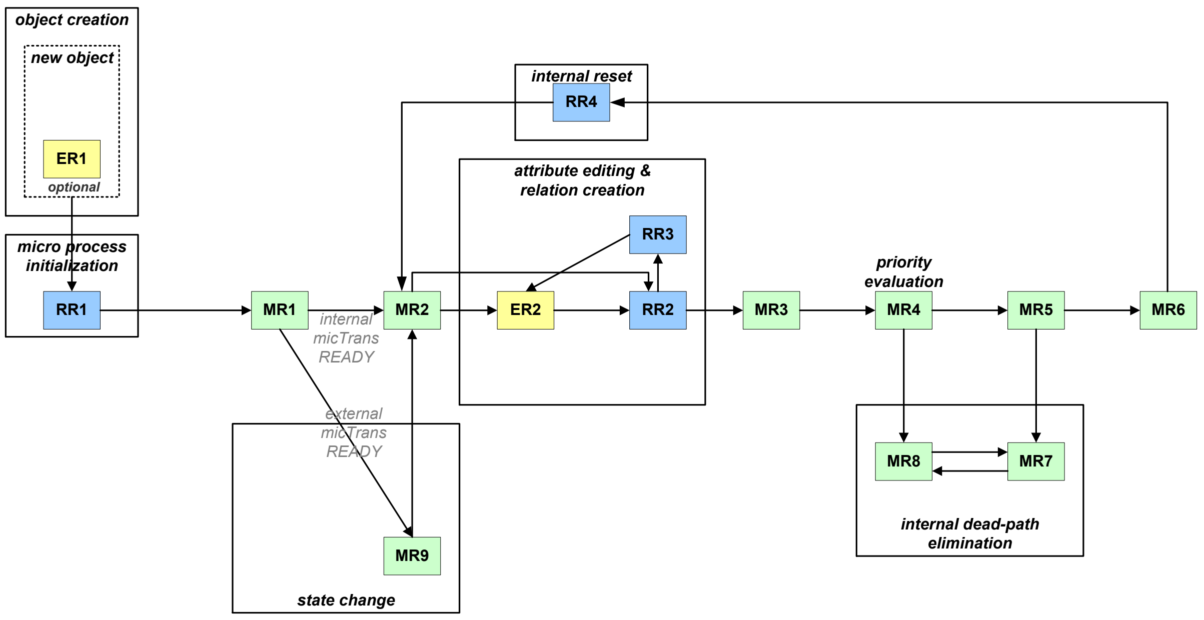


Figure 8.52: Rules for state changes

8.5.2 Explicit Micro Transitions

As opposed to implicit micro transitions, which are immediately marked as READY when their source micro step (or value step) becomes marked as UNCONFIRMED, explicit micro transitions additionally require a user commitment. Hence, respective micro transitions are first marked as CONFIRMABLE to indicate that a commitment of the responsible user is required. More precisely, if the source micro step (or value step) of a micro transition changes its marking from ACTIVATED to UNCONFIRMED, outgoing explicit micro transitions are marked as CONFIRMABLE.[6] To realize this behavior, Marking Rule MR1 is extended accordingly (cf. Marking Rule MR1”).

Marking Rule (MR1”: Marking explicit micro transitions as CONFIRMABLE):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Then:

a) see MR1 in Sect. 8.4.1

b) see MR1’ in Sect. 8.4.2

c) ∀ micStep=(ref, ValueSteps) ∈ MicStepSet with $M_{MicStep}$(micStep) = UNCONFIRMED:

- ∀ micTrans ∈ outtrans(micStep) with explicit(micTrans) = TRUE:
  $M_{MicTrans}$(micTrans) := CONFIRMABLE;
  i.e., if a micro step becomes marked as UNCONFIRMED, the explicit micro transitions originating from it will be marked as CONFIRMABLE.

- ∀ valueStep ∈ ValueSteps with $M_{MicStep}$(valueStep) = UNCONFIRMED:
  ∀ micTrans ∈ outtrans(valueStep) with explicit(micTrans) = TRUE:
  $M_{MicTrans}$(micTrans) := CONFIRMABLE;
  i.e., if a value step becomes marked as UNCONFIRMED, the explicit micro transitions originating from it are marked as CONFIRMABLE.

Example 8.34 (Applying Marking Rule MR1” to explicit micro transitions):
Consider Fig. 8.53. After reaching micro step `alternative job` (i.e., after marking this step as UNCONFIRMED), its outgoing micro transition, which targets at micro step `finished`, is re-marked from WAITING to CONFIRMABLE.

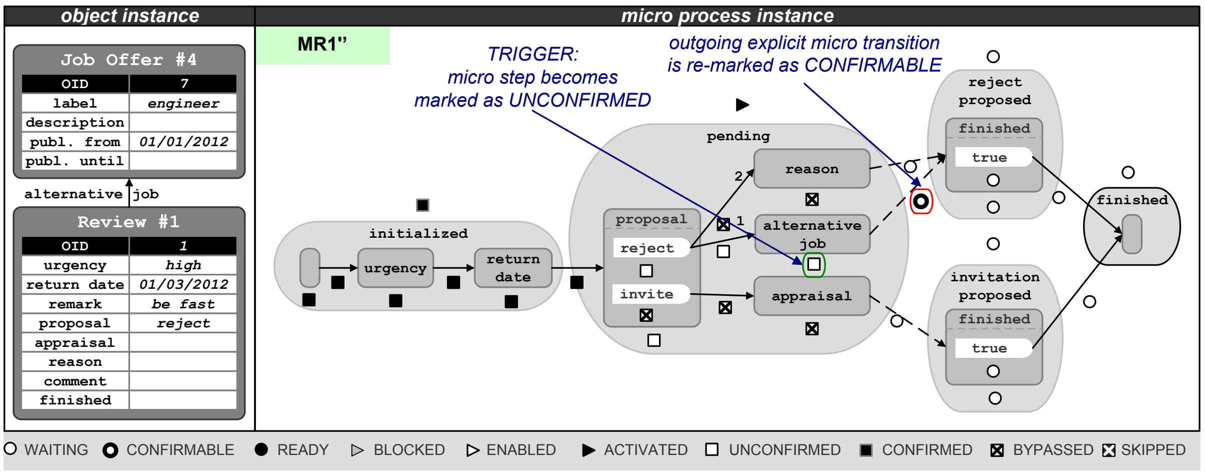


Figure 8.53: Applying Marking Rule MR1” to explicit micro transitions

If an external micro transition becomes marked as CONFIRMABLE, the subsequent state may be ACTIVATED when the responsible user commits the state transition. To commit a state transition, a mandatory activity is assigned to the worklist of the responsible users. Then, a commit button is dynamically added to the respective state-specific user form (cf. Fig. 8.54). According to *Execution Rule ER3*, as long as no such commitment has been made, however, respective user input is mandatorily required to proceed with micro process execution.

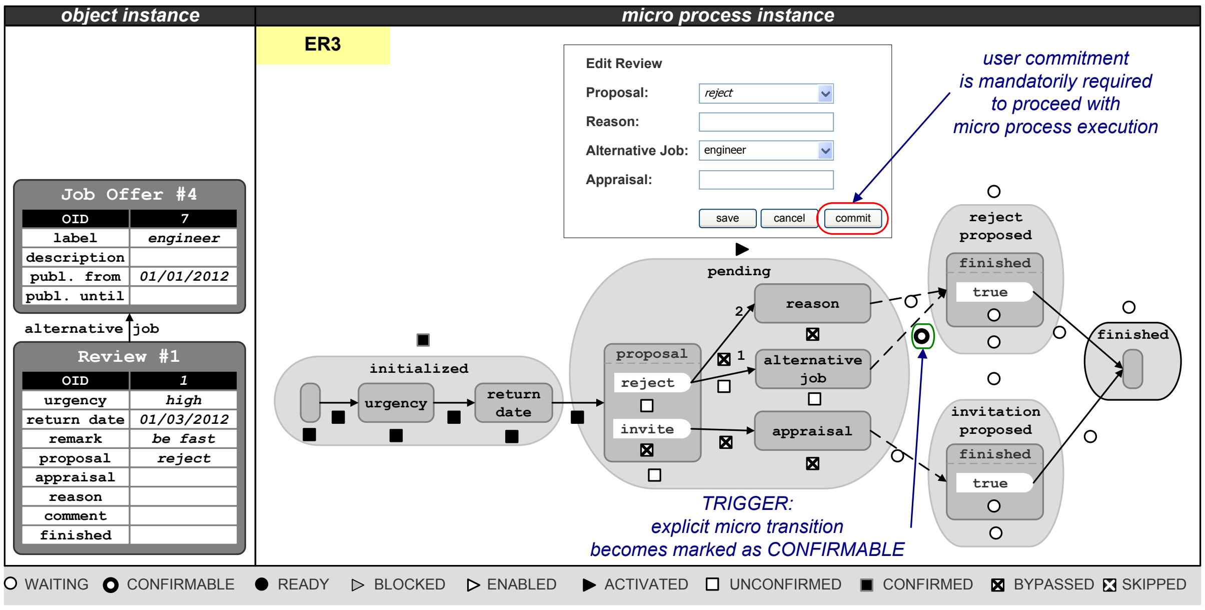
[6]Note that this may only apply to external micro transitions.

Execution Rule (ER3: User commit required):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

$\forall$ micTrans $\in$ MicTransSet with $M_{MicTrans}$(micTrans) = CONFIRMABLE:
A user commitment is required;

i.e., if an external micro transition is marked as CONFIRMABLE, a commitment of the responsible user will be required.

Example 8.35 (Applying Execution Rule ER3):
Consider Fig. 8.54. Before state `reject proposed` may be ACTIVATED, a user commitment is mandatorily required. Therefore, a corresponding commit button is dynamically added to the user form and offered as long as state `pending` is ACTIVATED.


object instance
micro process instance
ER3
user commitment is mandatorily required to proceed with micro process execution
alternative
job
initialized
urgency
return date
pending
proposal
reject
invite
reason
alternative job
appraisal
2
1
reject proposed
finished
true
invitation proposed
finished
true
finished
TRIGGER: explicit micro transition becomes marked as CONFIRMABLE
○ WAITING ◉ CONFIRMABLE ● READY ▷ BLOCKED ▷ ENABLED ▶ ACTIVATED ☐ UNCONFIRMED ■ CONFIRMED ⊠ BYPASSED ⊠ SKIPPED
save
cancel
commit

Edit Review

Proposal:	reject
Reason:	
Alternative Job:	engineer
Appraisal:	

Job Offer #4

OID	7
label	engineer
description	
publ. from	01/01/2012
publ. until	

Review #1

OID	1
urgency	high
return date	01/03/2012
remark	be fast
proposal	reject
appraisal	
reason	
comment	
finished	

Figure 8.54: Applying Execution Rule ER3

When a user commits the transition, the respective micro transition is re-marked as READY (cf. *Reaction Rule RR5*). To indicate whether a user commitment is made, we introduce function *committed*$_{\textit{state-change}}$ (cf. Def. 27).

Definition 27 (Commitment for micro transitions):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Further, let $MicTrans_{explicit}$:= micTrans $\in$ MicTransSet | explicit(micTrans) = TRUE. Then:

$committed_{state\text{-}change}$: $MicTrans_{explicit}$ $\mapsto$ **BOOLEAN** returns whether or not the required user commitment has been made for any particular explicit micro transition.

Reaction Rule (RR5: Marking explicit micro transitions as READY):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

micTrans $\in$ MicTransSet with $M_{MicTrans}$(micTrans) = CONFIRMABLE $\wedge$ $\text{committed}_{\text{state-change}}$(micTrans) = TRUE, $\Rightarrow$
$M_{MicTrans}$(micTrans) = READY;

i.e., when a user commits an external micro transition, it will be marked as READY.

Example 8.36 (Applying Reaction Rule RR5):
Consider Fig. 8.55. The micro transition between micro steps `alternative job` and `finished` is re-marked from CONFIRMABLE to READY.

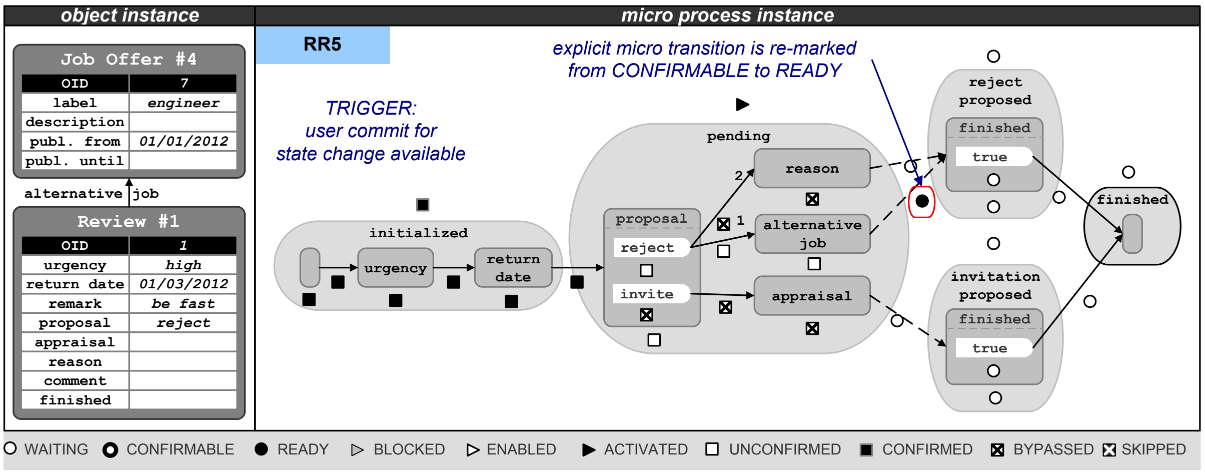


Figure 8.55: Applying Reaction Rule RR5

As illustrated in Fig. 8.56, when applying Marking Rule MR1, all explicit micro transitions are marked as CONFIRMABLE. For each of these micro transitions, a user commitment is mandatory. Hence, Execution Rule ER3 is triggered. When a commitment for a micro transition is made, in turn, the micro transition changes its marking from CONFIRMABLE to READY (cf. Reaction Rule RR5). Compared to implicit micro transitions, the corresponding state change takes place (i.e., Marking Rule MR9 is triggered), if an explicit micro transition is marked as READY, .

8.5.3 User Decisions

In certain situations, there exist multiple explicit micro transitions originating from the same micro step (cf. Fig. 8.57). In this case, we already ensure during micro process modeling that the target micro steps of these micro transitions belong to different states (cf. Def. 18). If the source micro step becomes marked as UNCONFIRMED, Marking Rule MR1 is triggered. It then re-marks all micro transitions, originating from this micro step, as CONFIRMABLE.

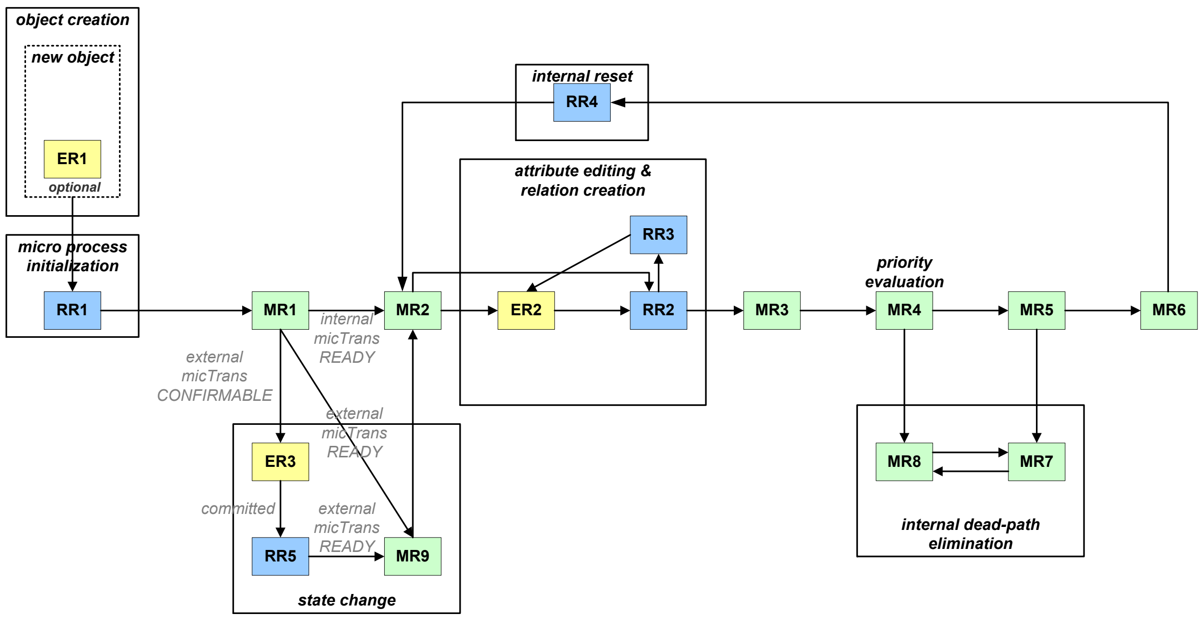


Figure 8.56: Rules for explicit micro transitions

Example 8.37 (Applying Marking Rule MR1 to user decisions):
Consider Fig. 8.57. If micro step A becomes marked as UNCONFIRMED, the micro transition between micro steps A and B becomes marked as CONFIRMABLE. In addition, the micro transition between micro steps A and C is re-marked as CONFIRMABLE.

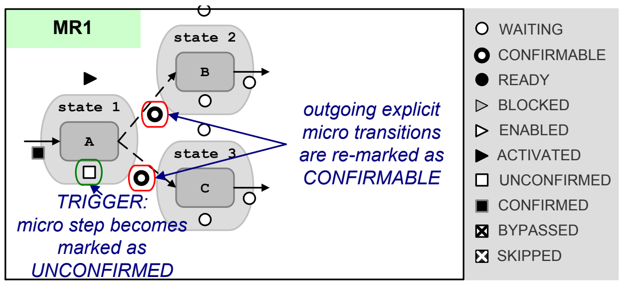

MR1
state 1
A
state 2
B
state 3
C
TRIGGER: micro step becomes marked as UNCONFIRMED
outgoing explicit micro transitions are re-marked as CONFIRMABLE
WAITING
CONFIRMABLE
READY
BLOCKED
ENABLED
ACTIVATED
UNCONFIRMED
CONFIRMED
BYPASSED
SKIPPED

Figure 8.57: Applying Marking Rule MR1 to user decisions

If multiple micro transitions become marked as CONFIRMABLE, alternative state transitions are possible. In this context, a user may select the subsequent state as he prefers; i.e., the user must decide which of these alternative states shall be activated. In this context, Execution Rule ER3 is used, requiring a user commitment for one of the respective micro transitions (e.g., using a combo box).

Example 8.38 (Applying Execution Rule ER3 to user decisions):
For example, consider Fig. 8.58. The responsible user may either commit the transition to state 2 or state 3.

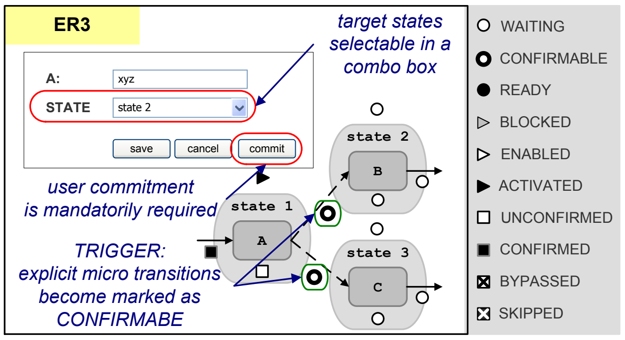

ER3
A:
xyz
STATE
state 2
save
cancel
commit
target states selectable in a combo box
user commitment is mandatorily required
TRIGGER: explicit micro transitions become marked as CONFIRMABE
state 1
A
state 2
B
state 3
C
WAITING
CONFIRMABLE
READY
BLOCKED
ENABLED
ACTIVATED
UNCONFIRMED
CONFIRMED
BYPASSED
SKIPPED

Figure 8.58: Applying Execution Rule ER3 to user decisions

In principle, several users maybe assigned to the same explicit micro transition type. Consider Ex. 8.39. At run-time, as soon as one of the users commits a state transition, this transition takes place.

Example 8.39 (User decisions with different responsible users):
Assume that user `1` is assigned to the micro transition linking micro steps `A` and `B`, while user `2` is assigned to the micro transition between micro steps `A` and `C`. Then, each of the two users may commit the respective state transition (cf. Sect. 7.3 for details).

If a micro transition, which is currently marked as CONFIRMABLE, is committed, conflicting external micro transitions must be skipped; i.e., their transitions must be marked as SKIPPED to indicate that the respective execution path of the micro process instance has not been selected. Hence, we must extend Reaction Rule RR5, which will be triggered as soon as one out of several conflicting micro transitions is committed by the respective user. All other micro transitions, which are still marked as CONFIRMABLE, change their marking to SKIPPED (cf. Reaction Rule RR5').

Reaction Rule (RR5': User Decisions):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Further, let $MicTransSet_{CONFIRMABLE}$:= {micTrans $\in$ MicTransSet | $M_{MicTrans}$(micTrans) = CONFIRMABLE}. Then:

$\exists$ a user commitment for micTrans $\in$ MicTransSet*, $\Rightarrow$

$$\forall \text{ micTrans} \in \text{MicTransSet: } M_{MicTrans}(\text{micTrans}) := \begin{cases} \text{READY}, & \text{a user commitment is available} \\ \text{SKIPPED}, & \text{else} \end{cases}$$

i.e., if a user commitment becomes available for an external explicit micro transition, the latter is marked as READY and all other external explicit micro transitions, currently marked as CONFIRMABLE, are now marked as SKIPPED .

Example 8.40 (Applying Reaction Rule RR5' to user decisions):
Consider Fig. 8.59. Since the responsible user has committed the change to state `2`, the external explicit micro transition between micro steps `A` and `B` is re-marked from CONFIRMABLE to READY. Opposed to this, the micro transition between micro steps `A` and `C` (currently marked as CONFIRMABLE) is re-marked as SKIPPED.

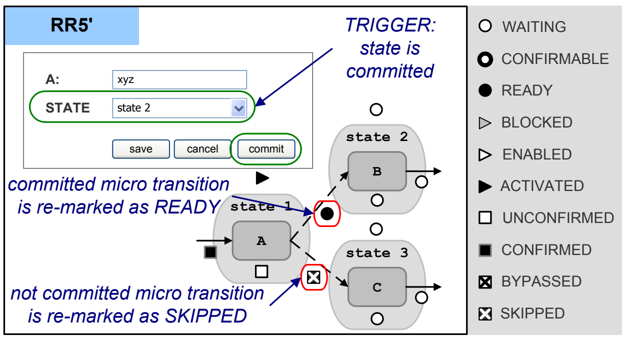


Figure 8.59: Applying Reaction Rule RR5' to user decisions

If an explicit micro transition is marked as READY, the corresponding state change may be executed (i.e., Marking Rule MR9 is triggered). By contrast, according to Reaction Rule RR5, when a certain state becomes skipped, the non-chosen execution path as well as the respective external micro transition become marked as SKIPPED.

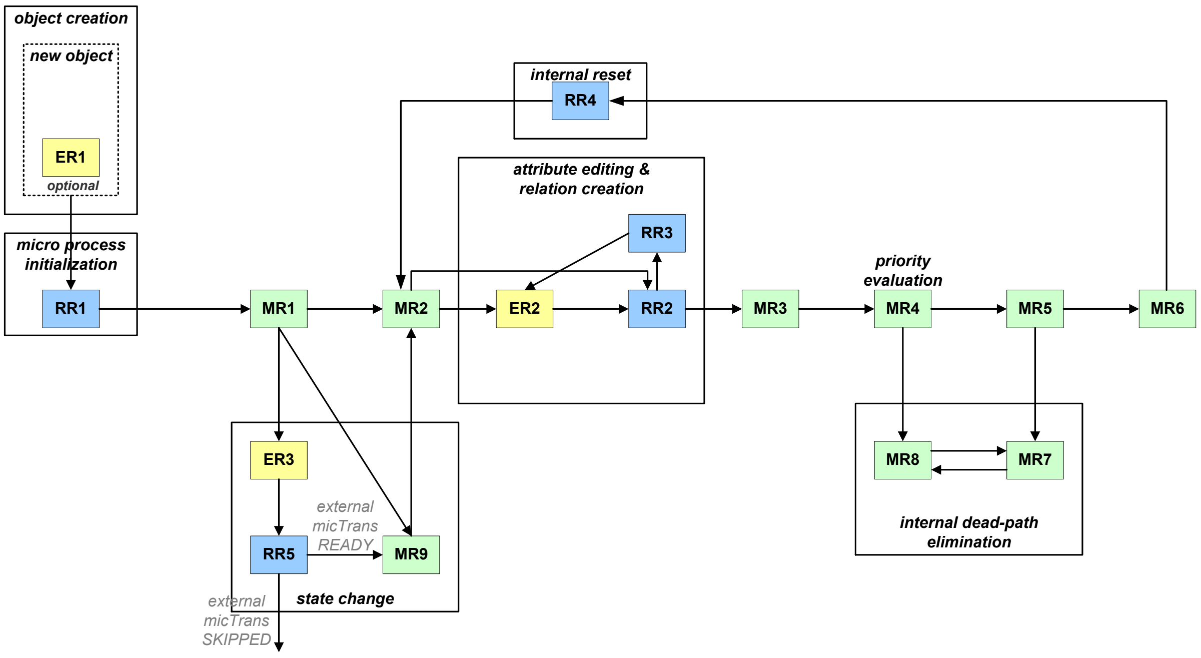


Figure 8.60: Rules considering user decisions

8.5.4 External Dead-path Elimination

In Sect. 8.4.4, we introduced an *internal dead-path elimination* for micro steps, value steps, and micro transitions of the currently activated state. Based on it, we can identify which input fields of the related user form; i.e., attribute values or relations) are not required anymore to proceed with micro process execution. Similarly, we must identify all states that are no longer reachable due to the selection of an alternative execution path. Especially, identifying these states is crucial for coordinating the execution of inter-dependent micro process instances (cf. Chapts. 10, 11, and 12). For this purpose, we introduce an *external dead-path elimination*. It marks all states, micro steps, value steps, and micro transitions as SKIPPED if they are not reachable any longer and do not belong to the currently activated state.

An *external dead-path elimination* is triggered when a micro step, value step, or micro transition becomes marked as SKIPPED. Consider a state change (i.e., Marking Rule MR9 and Reaction Rule RR5') as starting point for it. According to Reaction Rule RR5', a micro transition is marked as SKIPPED when a commitment for any other micro transition, which is currently marked as CONFIRMABLE, becomes available (cf. Sect. 8.5.3 for details).

Example 8.41 (Applying Marking Rule MR9 to trigger an external dead-path elimination):
Consider Fig. 8.61. After the user commitment for activating state `reject proposed` is made, the state change takes place; i.e., Marking Rule MR9 is triggered. Then, the previous state `pending` is re-marked from ACTIVATED to CONFIRMED. In addition, all micro steps, value steps, and micro transitions belonging to this state and currently marked as BYPASSED are re-marked as SKIPPED. In turn, this triggers an external dead-path elimination.

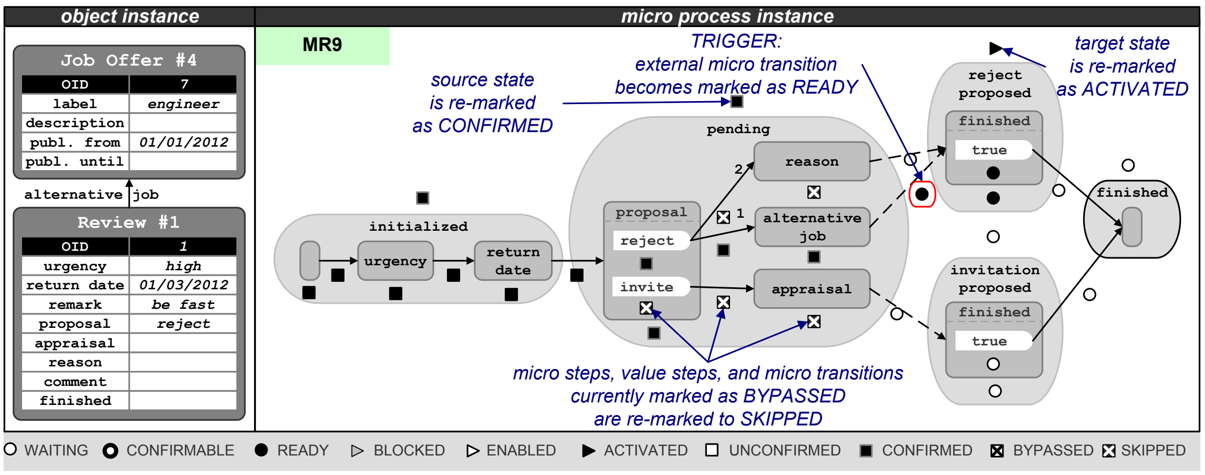


Figure 8.61: Applying Marking Rule MR9 to trigger an external dead-path elimination

Regarding the example from Fig. 8.61, micro steps `reason` and `appraisal` are now marked as SKIPPED. In turn, this triggers *Marking Rule MR10*. In particular, if a micro or value step is marked as SKIPPED, its outgoing micro transitions are re-marked from WAITING to SKIPPED. Following this, if all incoming external micro transitions of a micro step are marked as SKIPPED, this micro step itself must be marked as SKIPPED. Further, its value steps are then also re-marked as SKIPPED (cf. *Marking Rule MR11*).

Note that Marking Rules MR10 and MR11 may trigger each other; i.e., in the context of an external dead-path elimination these rules are iteratively invoked as long as a micro step is found whose incoming micro transition are all marked as SKIPPED. Further, the external dead-path elimination terminates in any case, when reaching an end micro step. Finally, if all micro steps of a state are marked as SKIPPED, the respective state must be marked as SKIPPED as well (cf. *Marking Rule MR12*).

Marking Rule (MR10: Marking micro transitions as SKIPPED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Then:

∀ micStep=(ref, ValueSteps) ∈ MicStepSet with $M_{MicStep}$(micStep) = SKIPPED:

- ∀ micTrans ∈ outtrans(micStep) with $M_{MicTrans}$(micTrans) = WAITING:
  $M_{MicTrans}$(micTrans) := SKIPPED;
  i.e., if a micro step becomes marked as SKIPPED, its outgoing micro transitions are re-marked from WAITING to SKIPPED.
- ∀ valueStep ∈ ValueSteps with $M_{MicStep}$(valueStep) = SKIPPED:
  ∀ micTrans ∈ outtrans(valueStep) with $M_{MicTrans}$(micTrans) = WAITING:
  $M_{MicTrans}$(micTrans) := SKIPPED;
  i.e., if a value step becomes marked as SKIPPED, its outgoing micro transitions are re-marked from WAITING to SKIPPED.

Marking Rule (MR11: Marking micro steps and value steps as SKIPPED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Then:

∀ micStep=(ref, ValueSteps) ∈ MicStepSet with $M_{MicStep}$(micStep) = WAITING:
∀ micTrans ∈ intrans(micStep): $M_{MicTrans}$(micTrans) = SKIPPED, ⇒

- $M_{MicStep}$(micStep) := SKIPPED;
  i.e., if all incoming micro transitions of a micro step are marked as SKIPPED, the micro step is marked as SKIPPED as well.
- ∀ valueStep ∈ ValueSteps with $M_{MicStep}$(micStep) = SKIPPED ∧ $M_{MicStep}$(valueStep) = WAITING:
  $M_{MicStep}$(valueStep) := SKIPPED;
  i.e., if a micro step is marked as SKIPPED, all corresponding value steps will be marked as SKIPPED as well.

Marking Rule (MR12: Marking states as SKIPPED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Further, let state = (name, sMicStepSet) ∈ StateSet be a state. Then:

∀ micStep ∈ sMicStepSet: $M_{MicStep}$(micStep) = SKIPPED, ⇒ M_{State}(state) := SKIPPED;

i.e., a state will be marked as SKIPPED if its micro steps (and hence value steps and micro transitions) are all marked as SKIPPED.

Example 8.42 (External dead-path elimination):
Consider Fig. 8.62. All micro transitions originating from micro steps `reason` and `appraisal` are re-marked as SKIPPED. Following this, consider micro step `finished` of state `invitation proposed` in Fig. 8.63. Since all incoming external micro transitions are marked as SKIPPED, this micro step as well as its value steps (i.e., value step `true`) are re-marked as SKIPPED. Further, since value step `true`, which belongs to state `invitation proposed`, becomes marked as SKIPPED, Marking Rule MR10 is re-triggered. Regarding Fig. 8.64, the micro transition between value step `true` of state `invitation proposed` and the end micro step is marked as SKIPPED as well. Opposed to micro step `finished` in state `invitation proposed` (cf. Fig. 8.64), micro step `finished` in state `reject proposed` has an incoming micro transition that is currently marked as READY. Hence, it must not be marked as SKIPPED; i.e., the external dead-path elimination terminates at this point. Furthermore, the end micro step has another incoming micro transition that is currently marked as WAITING. Hence, the end micro step is also not marked as SKIPPED;

i.e., it is still possible to reach it (and the end state respectively). Finally, it is no longer possible to activate state `invitation proposed`; i.e., all micro steps of this state are marked as SKIPPED. Hence, `invitation proposed` itself must be marked as SKIPPED as well (cf. Fig. 8.65).

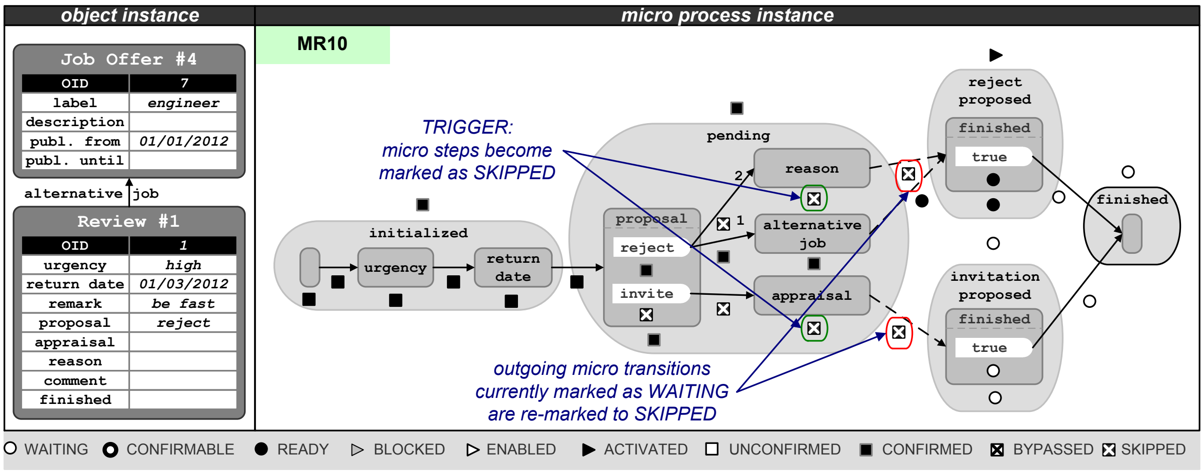


Figure 8.62: Applying Marking Rule MR10

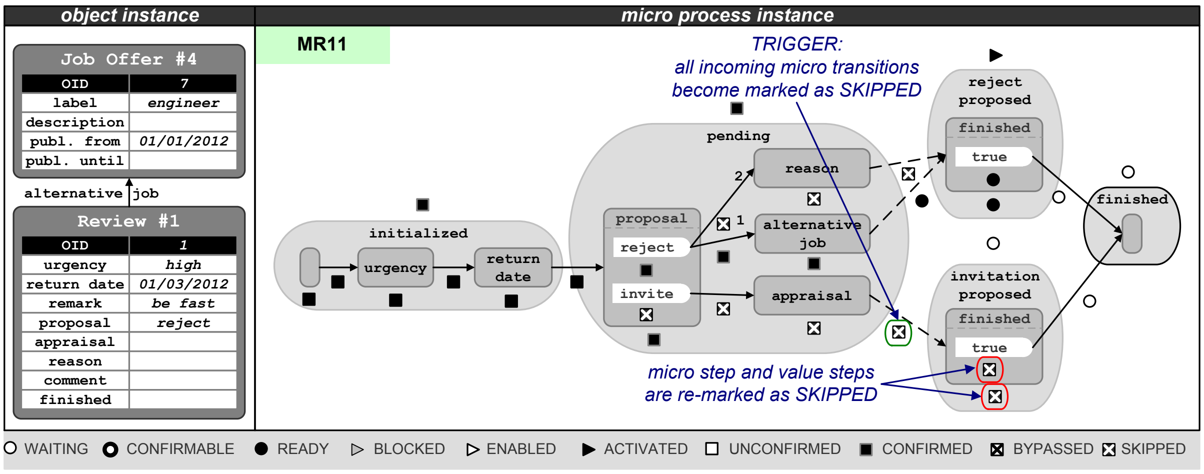


Figure 8.63: Applying Marking Rule MR11

Altogether, Fig. 8.66 illustrates the rules needed for performing an external dead-path elimination (i.e., Marking Rules MR10, MR11 and MR12). An external dead-path elimination is triggered in the following two cases:

a) The execution of Marking Rule MR9 is followed by marking a micro or value step as SKIPPED.

b) The execution of Reaction Rule RR5 is followed by re-marking an external micro transition from CONFIRMABLE to SKIPPED.

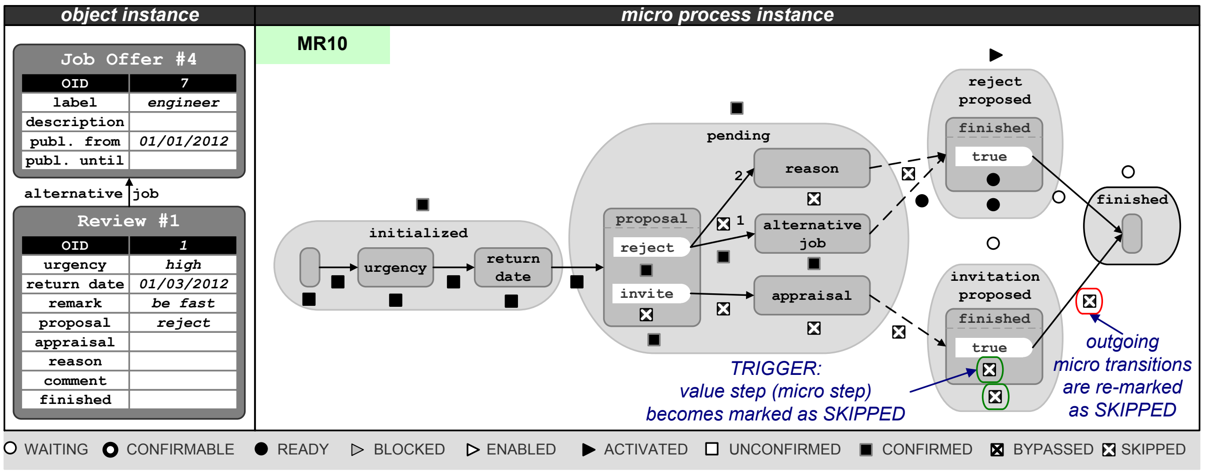

object instance
micro process instance
MR10
Job Offer #4
Review #1
alternative job
initialized
pending
reject proposed
invitation proposed
finished
TRIGGER: value step (micro step) becomes marked as SKIPPED
outgoing micro transitions are re-marked as SKIPPED
WAITING CONFIRMABLE READY BLOCKED ENABLED ACTIVATED UNCONFIRMED CONFIRMED BYPASSED SKIPPED

Figure 8.64: Applying Marking Rule MR10 iteratively

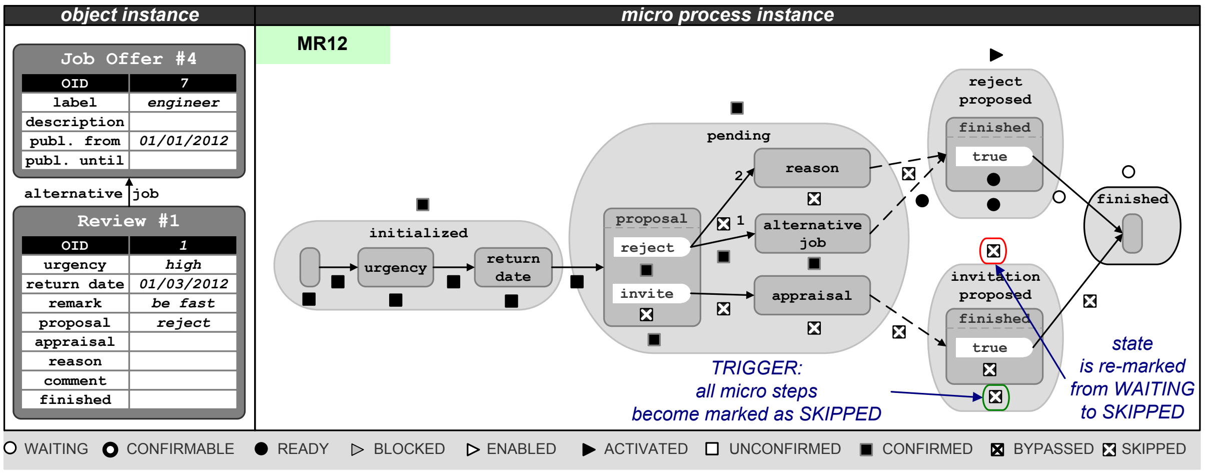

object instance
micro process instance
MR12
Job Offer #4
Review #1
alternative job
initialized
pending
reject proposed
invitation proposed
finished
TRIGGER: all micro steps become marked as SKIPPED
state is re-marked from WAITING to SKIPPED
WAITING CONFIRMABLE READY BLOCKED ENABLED ACTIVATED UNCONFIRMED CONFIRMED BYPASSED SKIPPED

Figure 8.65: Applying Marking Rule MR12

8.6 Backward Jumps

Sect. 7.5 has introduced *backward transitions*, which allow rolling back (i.e., resetting) a micro process instance if necessary (i.e., to jump back to a previous state). When the source state of a backward transition becomes activated, authorized users may reset the execution of the micro process instance to the target state of the backward transition (which is a predecessor of its source state in the normal flow of control). Initially, all backward transitions are marked as WAITING (cf. Reaction Rule RR1). As example, consider the backward transition connecting state `reject proposed` and state `pending` in Fig. 8.67.

Altogether, after a state becomes activated (i.e., Marking Rule MR9), Marking Rule MR13 is triggered (cf. Fig. 8.68). According to this rule, all backward transitions originating from the newly activated state are re-marked as CONFIRMABLE if their target state is currently marked as CONFIRMED. In turn, if the target state of a backward transition is currently marked as SKIPPED,

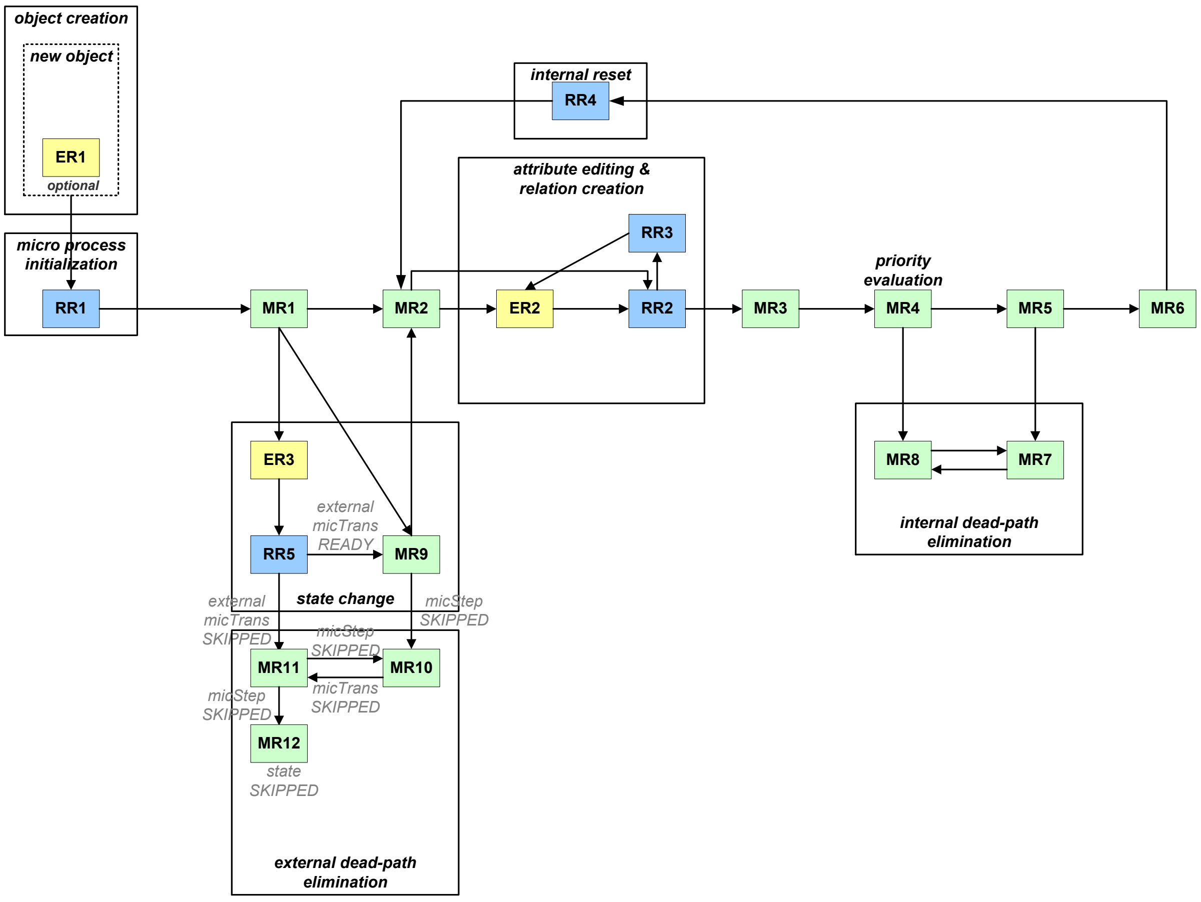


Figure 8.66: Rules for an external dead-path elimination

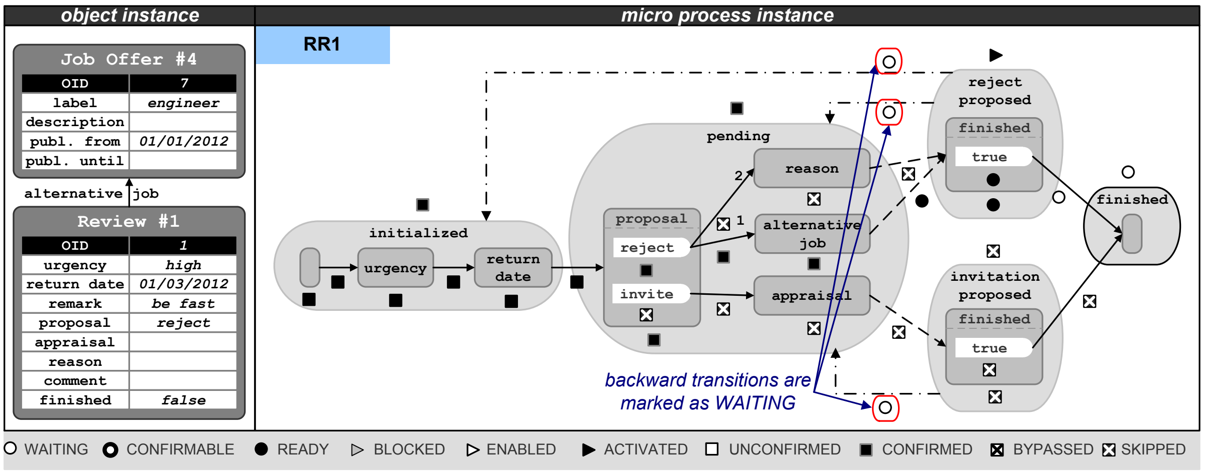


Figure 8.67: Applying Reaction Rule RR1 to backward transitions

this backward transition is re-marked as SKIPPED.
Following this, an authorized user may optionally commit a backward jump whose backward transition is currently marked as CONFIRMABLE (cf. Execution Rule ER4). If a commitment is made, the respective backward transition will be marked as READY (cf. Reaction Rule RR6). However, if a subsequent state becomes activated before initiating any backward jump, the corresponding backward transitions will be re-marked as SKIPPED (cf. Marking Rule MR9).

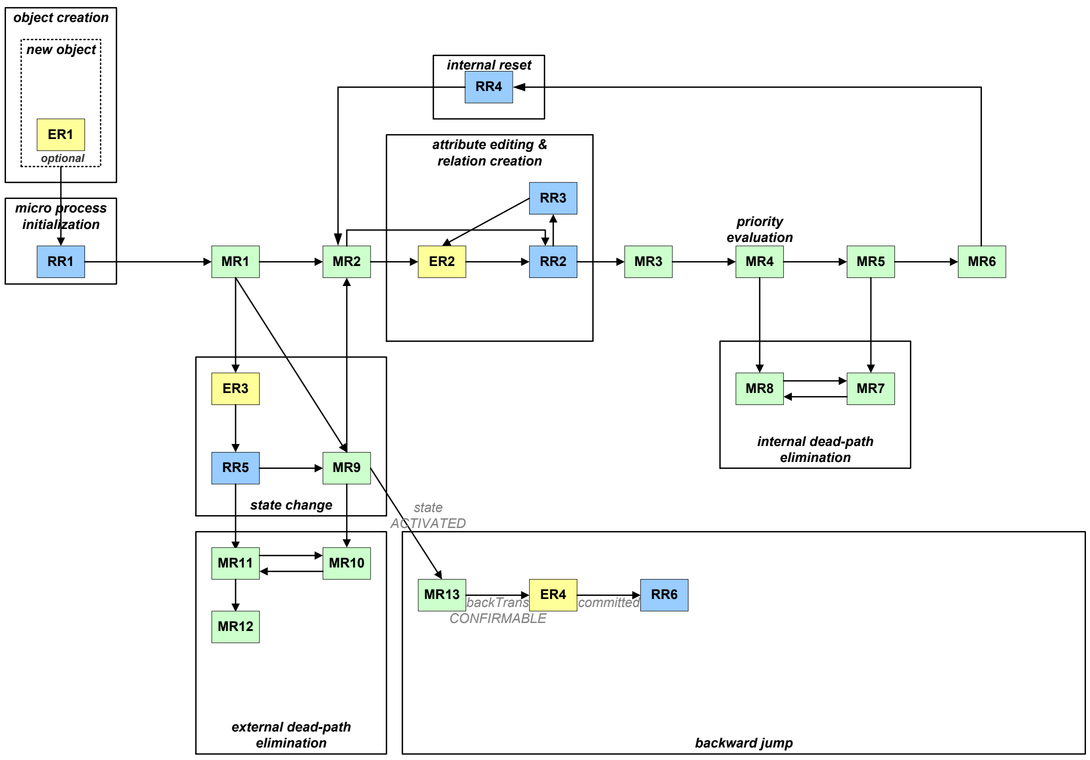


Figure 8.68: Rules for committing backward jumps

8.6.1 Committing Backward Jumps

Since we treat *backward transitions* are treated like explicit ones, a commitment of a responsible user[7] is required to perform the backward jump. First of all, all selectable backward transitions are marked as CONFIRMABLE; i.e., their source state becomes marked as ACTIVATED (cf. Marking Rule MR13). However, a backward jump to a previous state will be only possible if the previous state (i.e., the target state of the backward transition) was reached before during micro process execution; i.e., the target state of the backward transition must be currently marked as CONFIRMED. By contrast, if the target state is marked as SKIPPED, the backward transition must not be performed and therefore be also marked as SKIPPED.

Marking Rule (MR13: Marking backward transitions as CONFIRMABLE):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Further, let state $\in$ StateSet be the currently activated state; i.e., M_{State}(state) = ACTIVATED.

[7]The responsible user is defined by a corresponding backward responsibility (cf. Sect. 7.5).

Then:

∀ backTrans=(source, target) ∈ BackTransSet with source = state:

$$M_{\text{BackTrans}}(\text{backTrans}) := \begin{cases} \text{CONFIRMABLE}, & M_{\text{State}}(\text{target}) = \text{CONFIRMED} \\ \text{SKIPPED}, & M_{\text{State}}(\text{target}) = \text{SKIPPED} \end{cases}$$

i.e., if a state becomes marked as ACTIVATED, its outgoing backward transitions will be re-marked as CONFIRMABLE on condition that their target state is currently marked as CONFIRMED. In turn, if the target state is currently marked as SKIPPED, the backward transition will be re-marked as SKIPPED.

Example 8.43 (Applying Marking Rule MR13 to skip backward jumps):
Consider Fig. 8.69. When state 3 becomes marked as ACTIVATED, Marking Rule MR13 is triggered. The backward transition connecting states 3 and 1 is then marked as CONFIRMABLE since its target state (i.e., state 1) is currently marked as CONFIRMED (i.e., this state was previously reached). By contrast, the target state of the backward transition linking states 3 and 2 is currently marked as SKIPPED. Hence, this backward transition is also marked as SKIPPED.

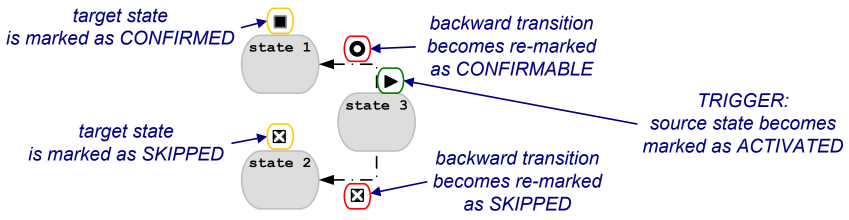


Figure 8.69: Applying Marking Rule MR13 to skip backward jumps

Example 8.44 (Applying Marking Rule MR13 for enabling backward jumps):
Consider Fig. 8.70. State `reject proposed` becomes marked as ACTIVATED. In turn, this triggers the execution of Marking Rule MR13. Accordingly, the backward transitions connecting state `reject proposed` with states `pending` and `initialized` are both re-marked as CONFIRMABLE. Note that none of the two backward transitions is marked as SKIPPED since both states `pending` and `initialized` are marked as CONFIRMED.

If a backward transition is marked as CONFIRMABLE, the micro process instance may be reset to the target state of this backward transition. However, a backward transition is not automatically executed, but has to be committed by a responsible user. Therefore, we introduce function $committed_{backTrans}$ for differentiating between committed and non-committed backward transitions (cf. Def. 28).

Definition 28 (Commitment for backward transitions):
Let micProcInstance = (micProc, oid, M_{State}, M_{MicStep}, M_{MicTrans}, $M_{\text{BackTrans}}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Then:

$\text{committed}_{\text{backTrans}}$: BackTransSet ↦ BOOLEAN defines whether or not the required user commitment has been made for a particular backward transition.

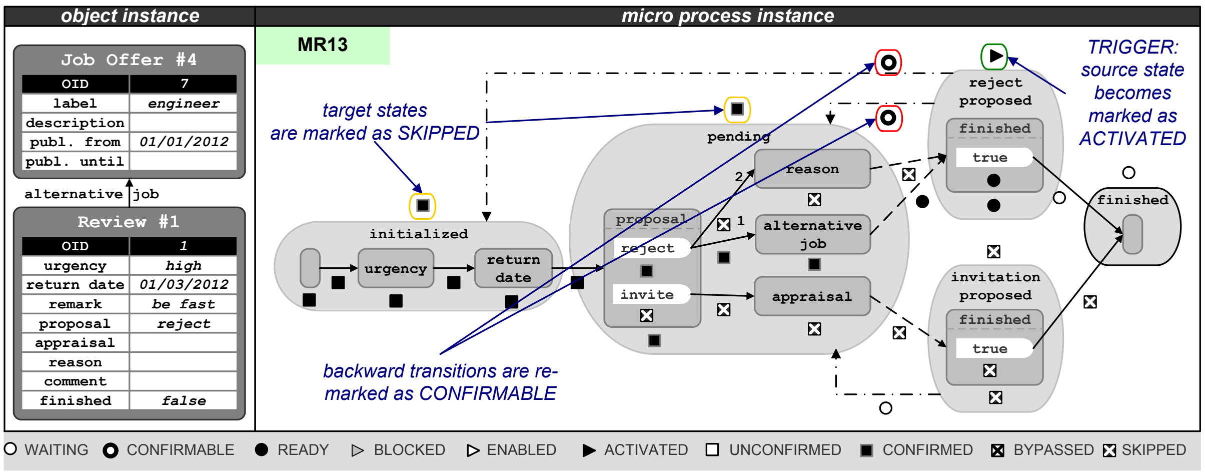


Figure 8.70: Applying Marking Rule MR13 to enable backward jumps

As opposed to explicit micro transitions, which must be executed in order to proceed with a micro process, the execution of backward transitions is optional. For example, a backward transition may be executed to handle an exceptional situation: To be more precise, a backward jump may be initiated when processing the form-based activity related to the source state of its backward transition. For this purpose, the target state is added to a respective combo box. In particular, all states that may be activated due to a backward transition are then added to the combo box (see Fig. 8.71).

Altogether, a responsible user may optionally commit a backward transition, which is marked as CONFIRMABLE, in order to reset micro process execution to a previous state (i.e., to its target state) (cf. *Execution Rule ER4*).

Execution Rule (ER4: Committing backward jumps):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

$\forall$ backTrans $\in$ BackTransSet with $M_{BackTrans}$(backTrans) = CONFIRMABLE $\wedge$ $committed_{backTrans}$(backTrans) = FALSE:
An authorized user may optionally commit backTrans.

i.e., if a backward transition changes its marking from WAITING to CONFIRMABLE, a commitment of the responsible user may be optionally made.

Example 8.45 (Applying Execution Rule ER4):
In Fig. 8.71, state `initialized` is selected. Using the commit button, the selected state is then activated.

If a backward transition is marked as CONFIRMABLE and the required commitment is made, while the source state is activated, the backward jump may be performed. To indicate this, the respective backward transition is marked as READY (cf. Reaction Rule RR6). However, we must consider that more than one backward transition, originating from the source state,

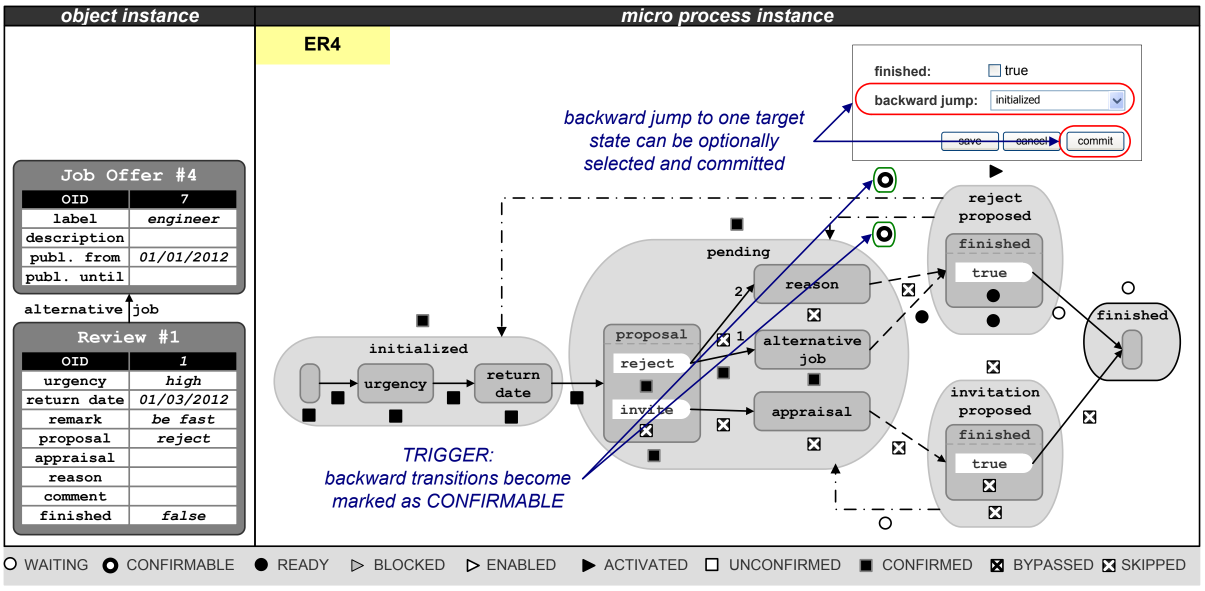


Figure 8.71: Applying Execution Rule ER4

may be currently marked as CONFIRMABLE; i.e., several backward jumps to different previous states may be performed (cf. Fig. 8.71). Accordingly, if one backward transition is marked as READY, all other ones (still marked as CONFIRMABLE) must be re-marked as WAITING. Note that since them might be executed later on, respective backward transitions are not marked as SKIPPED.

Reaction Rule (RR6: Marking backward transitions as READY):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Then:

$\forall$ backTrans $\in$ BackTransSet with $M_{BackTrans}$(backTrans) = CONFIRMABLE:

$$M_{BackTrans}(backTrans) := \begin{cases} \text{READY}, & \text{if } committed_{backTrans}(backTrans) = \text{TRUE} \\ \text{WAITING}, & \text{else} \end{cases}$$

i.e., if a user commits a backward transition, it is marked as READY, while all other backward transitions, which are currently marked as CONFIRMABLE, are then re-marked as WAITING.

Example 8.46 (Applying Reaction Rule RR6):
Consider Fig. 8.72. The backward jump from state `reject proposed` to the preceding state `initialized` was committed. In turn, this has triggered Reaction Rule RR6 according to which the backward transition from state `reject proposed` to state `initialized` is re-marked from CONFIRMABLE to READY. At the same time, the backward transition from state `reject proposed` to state `pending` is re-marked as WAITING.

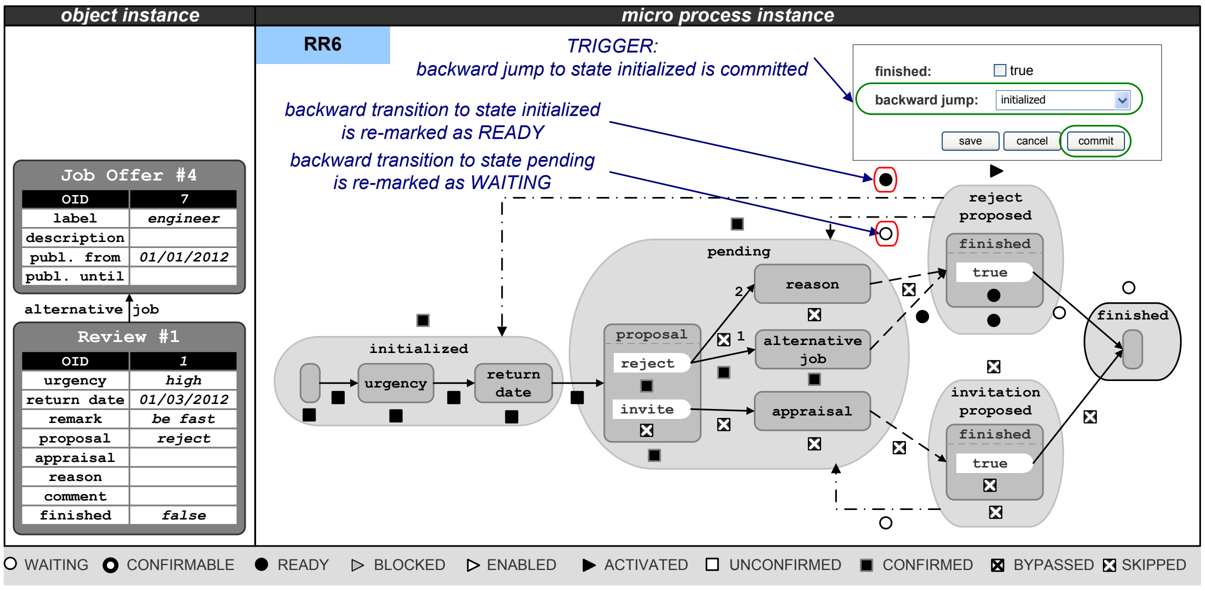


Figure 8.72: Applying Reaction Rule RR6

8.6.2 Backward Transitions during an External Dead-path Elimination

When completing the source state of a backward transition (i.e., this state is marked as CONFIRMED using Marking Rule MR9) and the respective backward jump was not initiated, the latter can no longer be performed unless the source state is re-activated later due to another backward jump. In this situation, according to *Marking Rule MR14*, all backward transitions originating from the respective state are re-marked as SKIPPED.

Marking Rule (MR14: Deactivating backward transitions):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Then:

```
∀ state ∈ StateSet with $M_{State}$(state) = CONFIRMED:
    ∀ backTrans=(source, target) ∈ BackTransSet with source = state:
        $M_{BackTrans}$(backTrans) := SKIPPED;
```

i.e., if a state becomes marked as CONFIRMED, its outgoing backward transitions are re-marked to SKIPPED.

We further extend the rules enabling an *external dead-path elimination* to consider backward transitions as well. If a state becomes marked as SKIPPED, according to *Marking Rule MR15*, its outgoing backward transitions must be marked as SKIPPED as well.

Marking Rule (MR15: Marking backward transitions as SKIPPED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Then:

```
∀ state ∈ StateSet with $M_{State}$(state) = SKIPPED:
```

∀ backTrans=(source, target) ∈ BackTransSet with source = state:
$M_{BackTrans}$(backTrans) := SKIPPED;

i.e., if a state becomes marked as SKIPPED, its outgoing backward transitions are re-marked to SKIPPED.

Example 8.47 (Applying Marking Rule MR15):
Consider Fig. 8.73. Since state `invitation proposed` is marked as SKIPPED, its outgoing backward transition, which targets at state `pending`, will be also marked as SKIPPED.

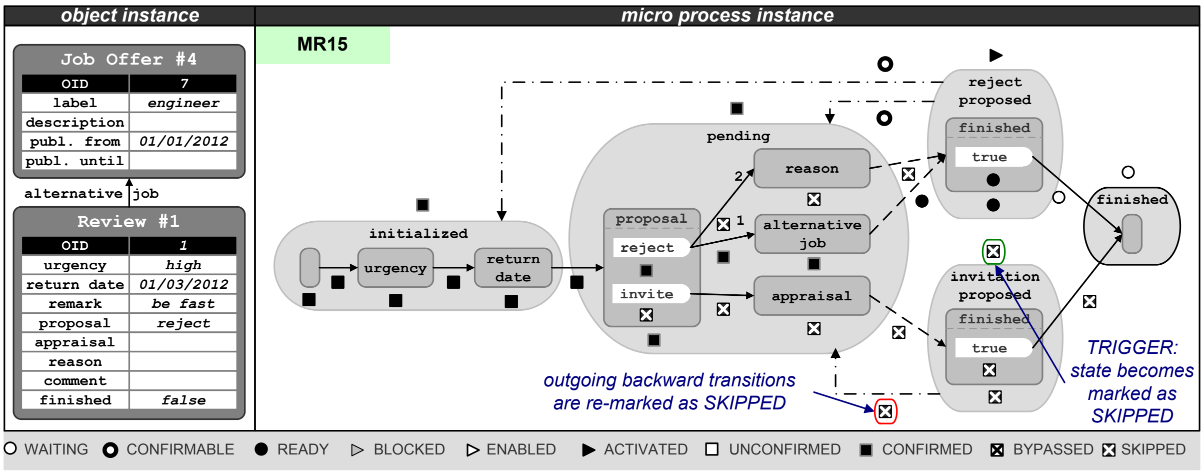


Figure 8.73: Applying Marking Rule MR15

Fig. 8.74 illustrates the rules required for the comprehensive external dead-path elimination described; i.e., Marking Rules MR14 and MR15 need to be added to properly deal with backward transitions in the context of an external dead-path elimination.

8.6.3 Executing Backward Jumps

If a backward transition is marked as READY, the respective backward jump to a preceding state must be executed (cf. Marking Rule MR16). According to this rule, the target state of the backward transition is re-marked from CONFIRMED to ACTIVATED (cf. Marking Rule MR16a). In addition, all micro and value steps belonging to the target state are re-marked as READY, regardless from whether these micro steps are currently marked as CONFIRMED or SKIPPED (cf. Marking Rule MR16b and c). Internal micro transitions, whose source and target micro steps (value steps) belong to the target state of the backward transition, in turn, are re-marked to WAITING (cf. Marking Rule MR16d). Moreover, each state (except the start state) that was reached during micro process execution was activated through exactly one external micro transition. In particular, there is exactly one external micro transition whose target micro step belongs to the target state of the backward transition and which is currently marked as CONFIRMED. Note that this marking indicates that the state has been activated using this micro transition. After resetting a micro process instance, execution must continue from this point. For this purpose, this micro transition is re-marked from CONFIRMED to READY (cf. Marking Rule MR16e). However,

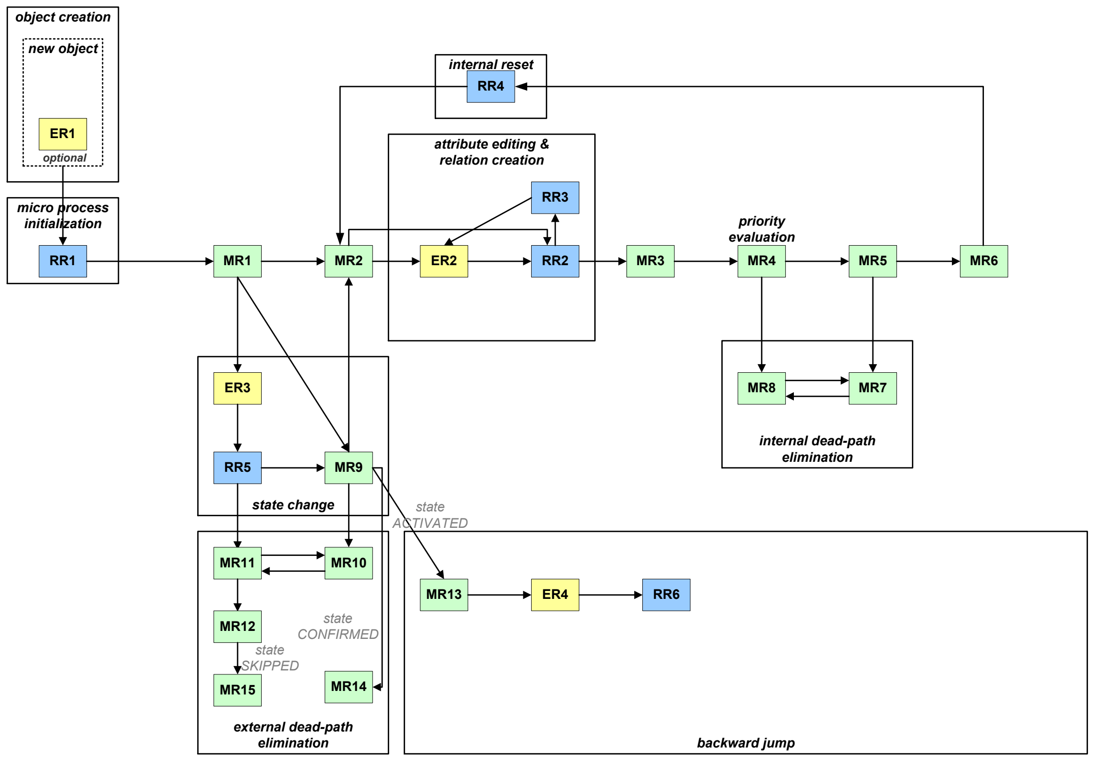


Figure 8.74: Rules considering backward jumps during an external dead-path elimination

if the target state of the backward transition corresponds to the start state of the micro process instance, the start micro step will be marked as UNCONFIRMED and all micro transitions originating from the start micro step will be re-marked as READY (cf. Marking Rule MR16f). Since it is possible to activate the backward transition once again, the backward transition itself is re-marked from READY to WAITING (cf. Marking Rule MR16g). Further, all external micro transitions originating from the state, which is now activated (i.e., the target state of the backward transition), are re-marked as WAITING (cf. Marking Rule MR16h). Finally, the source state of the backward transition is re-marked as SKIPPED (cf. Marking Rule MR16i).

Marking Rule (MR16: Backward Jump):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Further, let $\text{startState}_{micProcInstance}$ = (name, sMicStepSet) ∈ StateSet be the start state of micProcInstance and $\text{startMicStep}_{micProcInstance}$ ∈ sMicStepSet be the start micro step of micProcInstance. Then:

backTrans = (source, target) ∈ BackTransSet
with $M_{BackTrans}$(backTrans) = READY ∧ target = (name, sMicStepSet) ∈ StateSet:

i.e., if a backward transition becomes marked as READY, the following re-markings apply:

a) M_{State}(target) := ACTIVATED;
i.e., the target state of the executed backward transition is re-marked from CONFIRMED to ACTIVATED.

b) ∀ micStep=(ref, ValueSteps) ∈ sMicStepSet: $M_{MicStep}$(micStep) := READY;
i.e., all micro steps of the target state are re-marked as READY.

c) $\forall$ micStep=(ref, ValueSteps) $\in$ sMicStepSet:
 $\forall$ valueStep $\in$ ValueSteps: M_{MicStep}(valueStep) := READY;
 i.e., all value steps of the target state are re-marked as READY.

d) $\forall$ micTrans_1 $\in$ MicTransSet with {micTrans_1.source, micTrans_1.target} $\in$ sMicStepSet:
 $M_{\text{MicTrans}}(\text{micTrans}_1)$:= WAITING;
 i.e., all internal micro transitions of the target state are re-marked as WAITING.

e) target != $\text{startState}_{\text{micProcInstance}}$: $\exists!$ micTrans_2 = (s,t,prio) $\in$ MicTransSet with
 s $\notin$ sMicStepSet $\wedge$ t $\in$ sMicStepSet $\wedge$ $M_{\text{MicTrans}}(\text{micTrans}_2)$ = CONFIRMED:
 M_{MicTrans}(micTrans) := READY;
 i.e., if the target state of the backward transition differs from the start state of the micro process instance, the external micro transition, whose target micro step belongs to the target state and which is currently marked as CONFIRMED, will be re-marked to READY.

f) target = $\text{startState}_{\text{micProcInstance}}$:
 $M_{\text{micStep}}(\text{startMicStep}_{\text{micProcInstance}})$:= UNCONFIRMED $\wedge$
 $\forall$ micTrans_3 $\in$ MicTransSet with micTrans_3.source = $\text{startMicStep}_{\text{micProcInstance}}$:
 $M_{\text{MicTrans}}(\text{micTrans}_3)$:= READY;
 i.e., if the target state of the backward transition corresponds to the start state of the micro process instance, the start micro step will be re-marked as UNCONFIRMED and the micro transitions originating from it will be re-marked as READY.

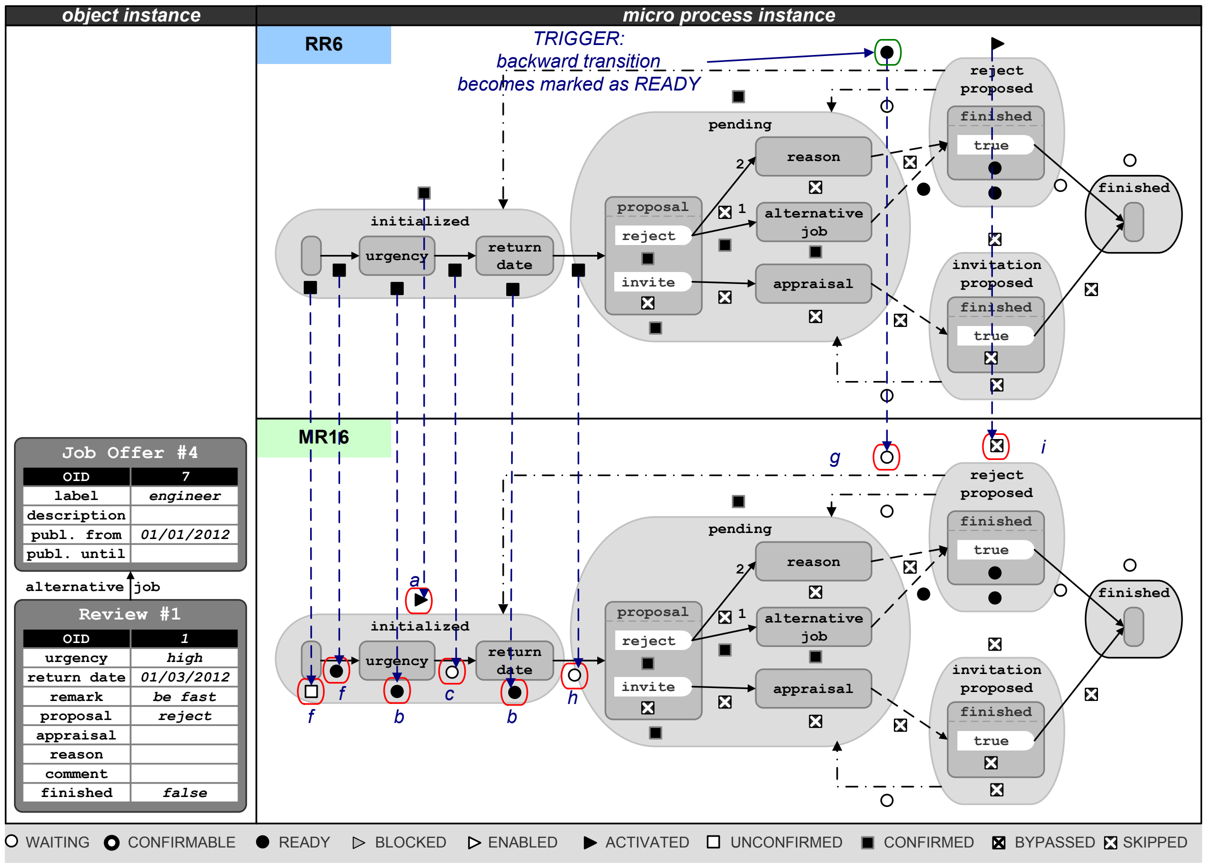
g) $M_{\text{BackTrans}}$(backTrans) := WAITING;
 i.e., the backward transition itself is re-marked from READY to WAITING.

h) micTrans_4 $\in$ MicTransSet with micTrans_4.source $\in$ sMicStepSet $\wedge$ micTrans_4.target $\notin$ sMicStepSet:
 $M_{\text{MicTrans}}(\text{micTrans}_4)$:= WAITING;
 i.e., external micro transitions, whose source micro step (or value step) belongs to the currently activated state, are re-marked as WAITING.

i) M_{State}(source) := SKIPPED;
 i.e., the source state of the backward transition is re-marked from ACTIVATED to SKIPPED.

Example 8.48 (Backward jump to the start state):
Consider the backward jump to state `initialized` as illustrated in Fig. 8.75. First, according to Marking Rule MR16, state `initialized` is re-marked from CONFIRMED to ACTIVATED (i.e., this state is re-activated). Second, all micro and value steps of state `initialized` are re-marked as READY (cf. Marking Rule MR16b and c), while all internal micro transitions are re-marked as WAITING (cf. Marking Rule MR16d). Since state `initialized` corresponds to the start state of the micro process instance, opposed to all other micro steps of state `initialized`, the respective start micro step is re-marked as UNCONFIRMED (cf. Marking Rule MR16f) and the micro transition connecting the start micro step with micro step `urgency` is re-marked as READY (cf. Marking Rule MR16f). Note that this marking triggers Marking Rule MR2, according to which the subsequent micro steps are enabled when required attribute values (or relations) become available. Furthermore, the backward transition between states `reject proposed` and `initialized` which enables the backward jump, is re-marked from READY to WAITING (cf. Marking Rule MR16g). Finally, the external micro transition, which originates from micro step `return date`, is re-marked as WAITING (cf. Marking Rule MR16h) and the (source) state `reject proposed` is marked as SKIPPED (cf. Marking Rule MR16i).

Example 8.49 (Backward jump to a state that differs from the start state):
Consider now Fig. 8.76. Here, a backward jump is not targeting at state `initialized`, but at state `pending`. Like in Example 8.48, the target state (i.e., state `pending`) is marked as ACTIVATED (cf. Marking Rule MR16a) and the backward transition is re-marked from READY to WAITING (cf. Marking Rule MR16g). All micro and value steps of state `pending` are re-marked as READY (cf. Marking Rule MR16b and c). Furthermore, all internal micro transitions are re-marked as WAITING (cf. Marking Rule MR16d). Opposed to Ex. 8.48, however, state `pending` is not the start state of the micro process instance. Hence, the external micro transition, which was used to activate state `pending`, must be marked as READY. Consider the micro transition connecting micro steps `return date` (belonging



Job Offer #4	
OID	7
label	*engineer*
description	
publ. from	*01/01/2012*
publ. until	

Review #1	
OID	1
urgency	*high*
return date	*01/03/2012*
remark	*be fast*
proposal	*reject*
appraisal	
reason	
comment	
finished	*false*

○ WAITING ◉ CONFIRMABLE ● READY ▷ BLOCKED ▷ ENABLED ▶ ACTIVATED □ UNCONFIRMED ■ CONFIRMED ⊠ BYPASSED ⊠ SKIPPED

Figure 8.75: Applying Marking Rule MR16 to a backward jump to the start state

to state `initialized`) and `proposal` (belonging to state `pending`). It is re-marked from CONFIRMED to READY. Note that this triggers Marking Rule MR2 to re-execute the micro process instance. Finally, the external micro transitions originating from micro steps `reason`, `alternative job`, and `appraisal` are re-marked as WAITING (cf. Marking Rule MR16h) and the (source) state `reject proposed` is re-marked as SKIPPED (cf. Marking Rule MR16i).

In summary, a backward jump will be initiated if a backward transition is marked as READY (cf. Fig. 8.77).

8.6.4 Re-setting Micro Process Instances

Before a particular region of a micro process instance may be re-executed, all states succeeding the target state of the respective backward jump (as well as all their micro steps, value steps, micro transitions, and backward transitions) must be reset. First, micro steps whose incoming micro transitions are all marked as WAITING, are then re-marked as WAITING as well. Thereby, only micro steps not belonging to the currently activated state are considered. In turn, micro steps of the currently activated state (i.e., the target state of the backward transition) must be further marked as READY (cf. Marking Rules MR16b + c). In addition, if a micro step is marked

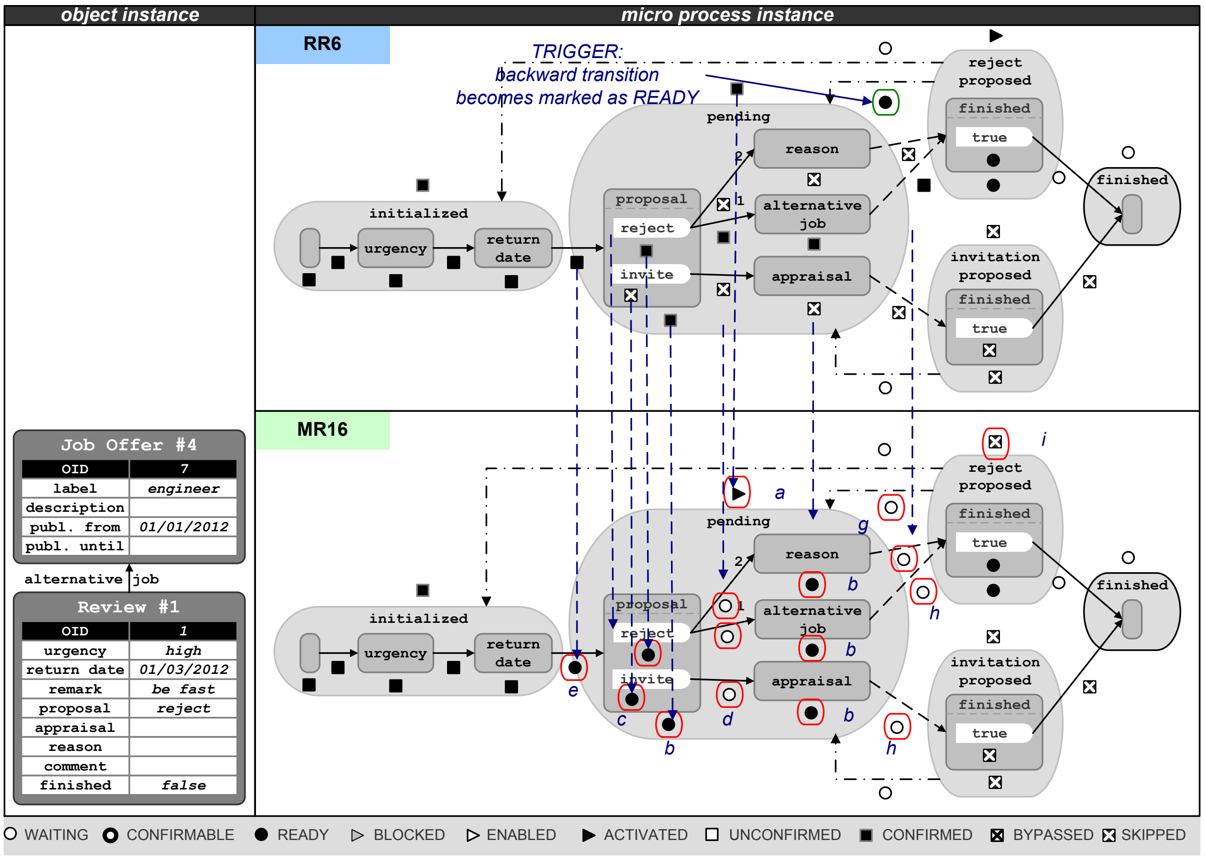


Figure 8.76: Applying Marking Rule MR16 to a backward jump that targets at a state differing from the start state

as WAITING, according to *Marking Rule MR17*, its value steps will be marked as WAITING as well.

Marking Rule (MR17: Resetting micro steps and value steps):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

$\forall$ micStep=(ref, ValueSteps) $\in$ sMicStepSet with state $\in$ StateSet $\wedge$ M_{State}(state) $\neq$ ACTIVATED:

a) $\forall$ micTrans $\in$ intrans(micStep) with $M_{MicTrans}$(micTrans) = WAITING: $M_{MicStep}$(micStep) := WAITING;
i.e., if all incoming micro transitions of a micro step, which do not belong to the currently activated state, are marked as WAITING, the micro step will be also marked as WAITING.

b) $\forall$ valueStep $\in$ ValueSteps with $M_{MicStep}$(micStep) = WAITING: $M_{MicStep}$(valueStep) := WAITING;
i.e., if a micro step is marked as WAITING, all corresponding value steps will be also marked as WAITING.

Example 8.50 (Applying Marking Rule MR17 to reset micro and value steps):
Consider Fig. 8.78 which illustrates the markings after executing the backward jump. When jumping back to state `initialized`, all subsequent states (i.e., `pending`, `reject proposed`, `invitation proposed`, and `finished`) as well as corresponding micro steps, value steps, micro transitions, and backward transitions must be reset. For this

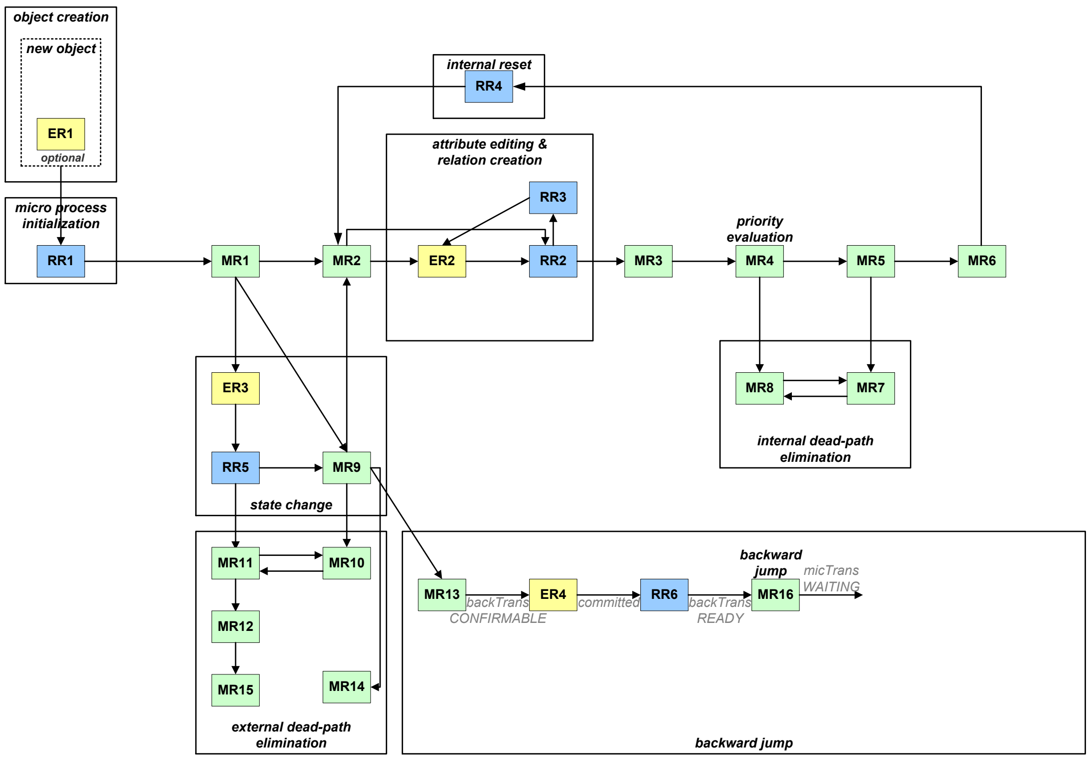


Figure 8.77: Rules for performing a backward jumps

purpose, Marking Rule MR17 is applied. Since the micro transition between micro steps `return date` and `proposal` is marked as WAITING, micro step `proposal` itself is also marked as WAITING. In addition, value steps `reject` and `invite` of micro step `proposal` are marked as WAITING.

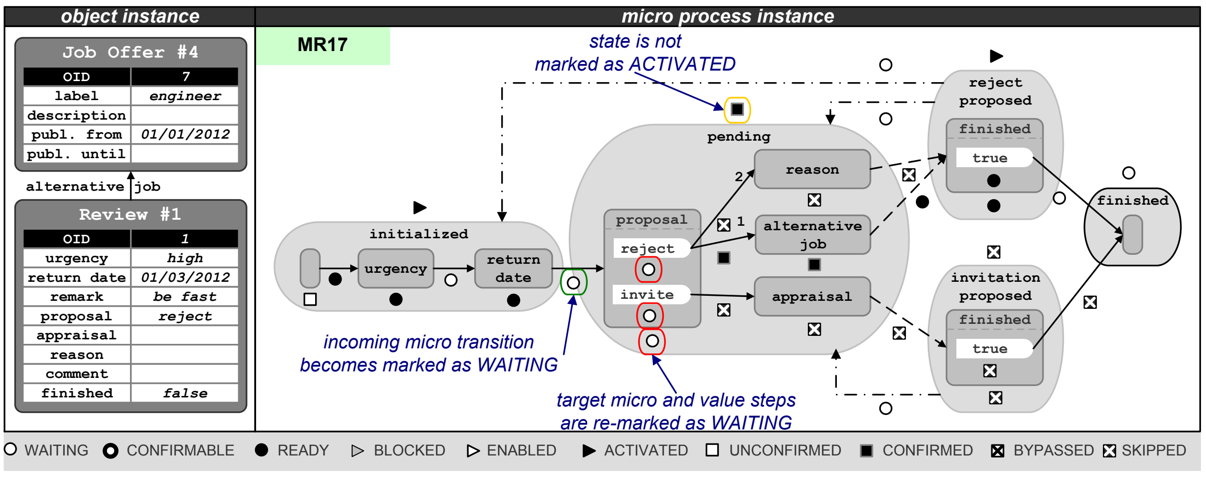


Figure 8.78: Applying Marking Rule MR17 to reset micro and value steps

According to *Marking Rule MR18*, all outgoing micro transitions are also marked as WAITING when a micro step or value step becomes marked as WAITING.

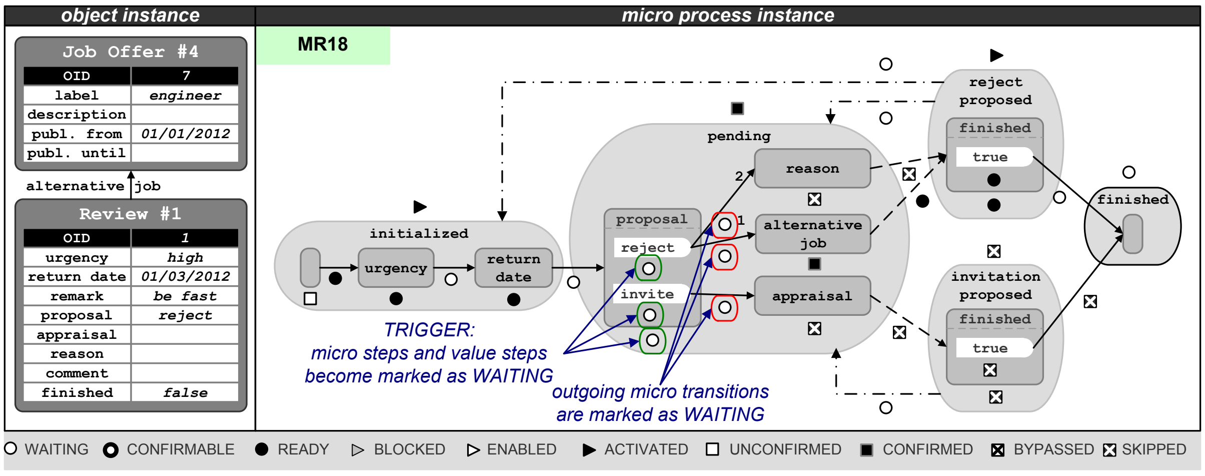
Marking Rule (MR18: Resetting micro transitions):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Then:

∀ micStep=(ref, ValueSteps) ∈ sMicStepSet with $M_{MicStep}$(micStep) = WAITING:

a) ∀ micTrans ∈ outtrans(micStep): $M_{MicTrans}$(micTrans) := WAITING;
i.e., if a micro step is marked as WAITING, its outgoing micro transitions will be also marked as WAITING.

b) ∀ valueStep ∈ ValueSteps with $M_{MicStep}$(valueStep) = WAITING:
∀ micTrans ∈ outtrans(valueStep): $M_{MicTrans}$(micTrans) := WAITING;
i.e., if a value step is marked as WAITING, its outgoing micro transitions will be also marked as WAITING.

Example 8.51 (Applying Marking Rule MR18 to reset micro transitions):
Consider Fig. 8.79. According to Marking Rule MR18, all micro transitions originating from micro step `proposal` are marked as WAITING; i.e., the micro transitions between value step `reject` and micro step `reason`, between value step `reject` and micro step `alternative job`, and between value step `invite` and micro step `appraisal`.



Job Offer #4

OID	7
label	*engineer*
description	
publ. from	*01/01/2012*
publ. until	

Review #1

OID	*1*
urgency	*high*
return date	*01/03/2012*
remark	*be fast*
proposal	*reject*
appraisal	
reason	
comment	
finished	*false*

Figure 8.79: Applying Marking Rule MR18 to reset micro transitions

Note that Marking Rules MR17 and MR18 trigger each other iteratively similar to internal and external dead-path eliminations. Therefore, all subsequent execution paths are reset until reaching an end micro step (cf. Fig. 8.80).

Furthermore, in the given context, states and backward transitions must be reset as well. This is accomplished by *Marking Rules MR19 and MR20*. A state is marked as WAITING if its corresponding micro steps are all marked as WAITING. In turn, a backward transition is marked as WAITING when its source state becomes marked as WAITING.

Marking Rule (MR19: Resetting states):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocin-

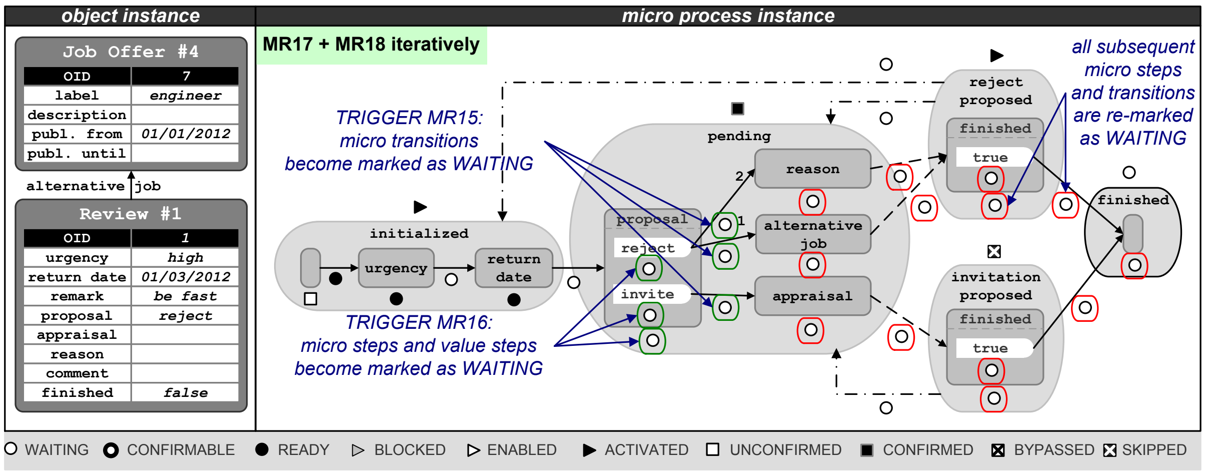


Figure 8.80: Applying Marking Rule MR17 and MR18 iteratively

stances(micProc). Further, let state = (name, sMicStepSet) ∈ StateSet be a state. Then:

∀ micStep ∈ sMicStepSet with $M_{MicStep}$(micStep) = WAITING: M_{State}(state) := WAITING;

i.e., a state is marked as WAITING if its micro steps are all marked as WAITING.

Marking Rule (MR20: Resetting backward transitions):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance ∈ micprocinstances(micProc). Then:

∀ backTrans=(source, target) ∈ BackTransSet with M_{State}(source) = WAITING: $M_{backTrans}$(backTrans) := WAITING;

i.e., a backward transition is marked as WAITING when its source state becomes marked as WAITING.

Example 8.52 (Resetting states and backward transitions):
Consider Fig. 8.81. States `pending`, `reject proposed`, `invitation proposed`, and `finished` are re-marked as WAITING when applying Marking Rule MR19. In addition, according to Marking Rule MR20, the backward transitions between states `reject proposed` and `initialized`, between states `reject proposed` and `pending`, and between `invitation proposed` and `pending` are marked as WAITING.

Overall, in the context of a backward jump, an external reset of the micro process instance is applied starting from the currently activated state (i.e., the target state of the backward transition). For this purpose, Marking Rules MR17, MR18, MR19, and MR20 are applied (cf. Fig. 8.82).

8.6.5 Re-executing Micro Process Instances

After performing a backward jump, at least one micro transition is marked as READY. More precisely, if the newly activated state corresponds to the start state of the micro process instance,

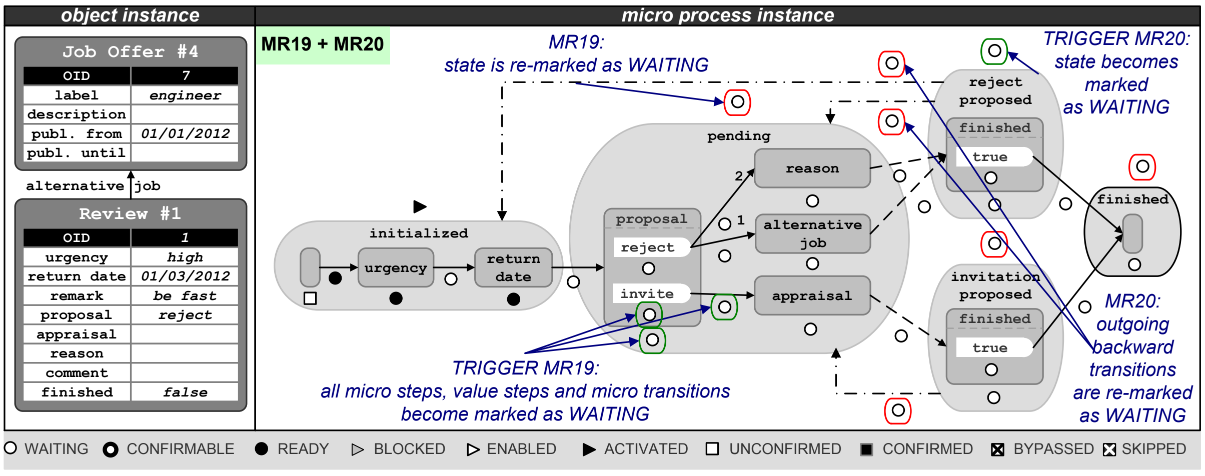


Figure 8.81: Applying Marking Rules MR19 and MR20 to reset states and backward transitions

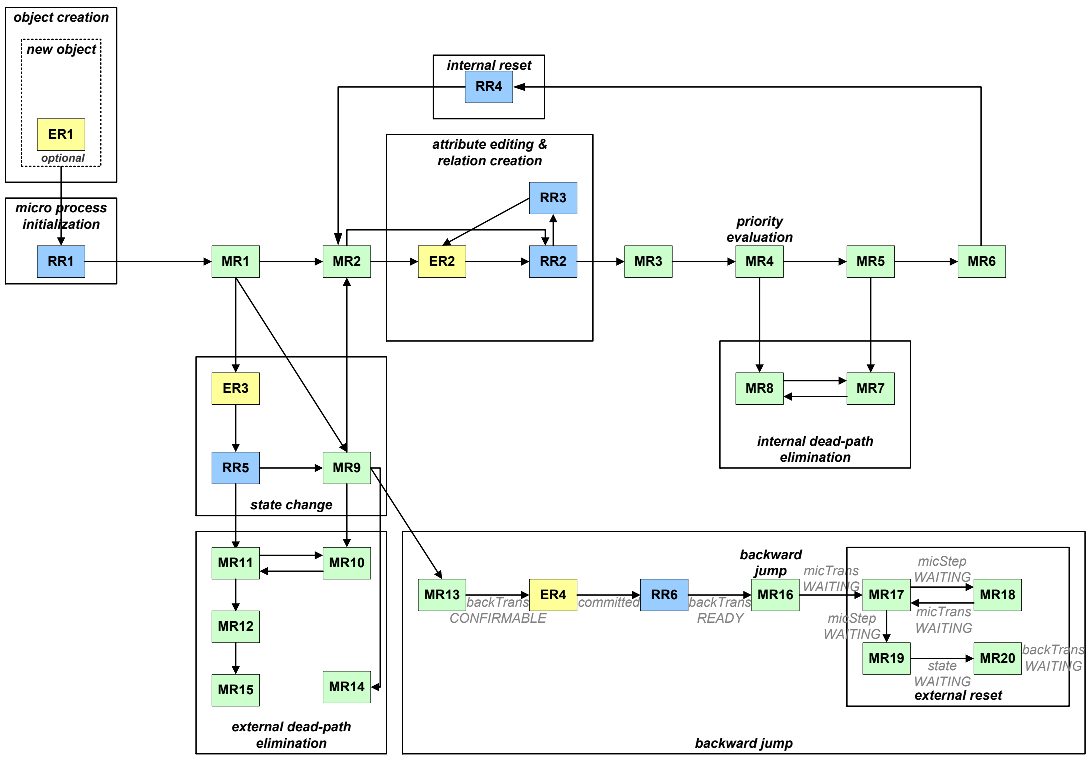


Figure 8.82: Rules for resetting the micro process instance

all micro transitions originating from the corresponding start micro step are marked as READY (cf. Marking Rule MR16f). By contrast, if the currently activated state does not correspond to the start state, exactly one external micro transition (whose target micro step belongs to the currently activated state) is marked as READY (cf. Marking Rule MR16e). Starting from the micro transitions, which are currently marked as READY, the region of the micro process instance that was reset is now re-executed again (cf. Marking Rule MR2). As already explained, if required attribute values or relations a particular micro step refers to, are already available, micro process execution automatically proceeds. More precisely, if the required attribute value or relation is available, Execution Rule ER2 is automatically skipped. Instead, Reaction Rule RR2 is executed. Consequently, micro process execution automatically proceeds until an explicit external micro transition is reached that requires a respective user commitment. Until this point, no user intervention for changing already assigned attribute values or relations will be possible (i.e., required attribute values are already assigned in the first iteration). In particular, if a subsequent state is reachable using an implicit external micro transition, this state will be immediately activated without allowing for any user intervention. In particular, it is not possible to revise data inputs.

Example 8.53 (Automatic re-execution):
Consider Fig. 8.81. State `initialized` becomes activated due to the backward jump outgoing from state `reject proposed`. Since values for attributes `urgency` and `return date` are already available, state `initialized` will be automatically marked as CONFIRMED and its subsequent state `pending` as ACTIVATED.

One option to prevent such automatic re-execution of micro process instances without resetting attribute values would be to delete the respective attribute values before. However, in this case, we loose all work already done (i.e., already assigned attribute values must then be reset even if they shall not be changed). Hence, this is not a feasible solution. To enable more user interventions, we introduce user commitments for already set attribute values and relations. In this context, consider data markings UNASSIGNED, ASSIGNED, and CONFIRMABLE (cf. Def. 29). Each user, who is allowed to set required attribute values or relations, may commit respective values.

Definition 29 (Data Markings):
DataMarkings := {UNASSIGNED, ASSIGNED, CONFIRMABLE} corresponds to the set of all markings defined for attributes and relations (cf. Tab. 8.7).

Marking	Label	Description
○	UNASSIGNED	For this attribute or relation, no value is currently assigned.
■	ASSIGNED	For this attribute or relation, a value is currently assigned.
◉	CONFIRMABLE	For this attribute or relation, a value is currently assigned, but must be committed by an authorized user.

Table 8.7: Data Markings

Data markings are only important for attributes and relations referenced by a corresponding micro step. In this context, we must consider that more than one micro step may refer to the same attribute or relation (i.e., these micro steps belong to different states). For this reason, data markings are assigned to micro steps rather than to attributes and relations. Accordingly, we extend the definition of micro steps as specified in Def. 30.

Definition 30 (Extending Micro Steps with Data Markings):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Further, let micStep = (ref, ValueSteps) $\in$ MicStepSet be a micro step. Then:

M_{Data}: MicStepSet $\mapsto$ DataMarking assigns to each micro step ms its marking M_{Data}(ms) $\in$ DataMarkings.

We extend the definition of a micro process instance accordingly. In the following, therefore, a micro process instance represents a tuple **micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data})**.

When performing a backward jump, the data markings of affected micro steps (i.e., the micro steps marked as WAITING or READY) are changed to CONFIRMABLE. For these micro steps, attribute values must be committed before proceeding with micro process execution. Note that this allows responsible users to change already set attribute values and relations when re-executing the micro process instance. More precisely, users may delete these attribute values and relations. In this case, data marking are changed to UNASSIGNED. When a new value is assigned, however, the respective data marking is changed to ASSIGNED and micro process execution may proceed. Alternatively, users may commit already assigned values. In this case, respective data markings are changed from CONFIRMABLE to ASSIGNED, triggering the corresponding reaction rules.

Fig. 8.83 illustrates possible data markings and their transitions:

1. If a value is set for the referenced attribute or relation, the data marking of the respective micro step changes from UNASSIGNED to ASSIGNED.
2. If the value of a referenced attribute or relation is deleted, the data marking of the respective micro step changes from ASSIGNED to UNASSIGNED.
3. After a backward jump, all data markings of the micro steps corresponding to the backward region are changed from ASSIGNED to CONFIRMABLE.
4. If a user commitment is made for a micro step after a backward jump, the data marking of this micro step changes CONFIRMABLE to ASSIGNED.
5. If an attribute value or relation (which is referenced by a micro step with data marking CONFIRMABLE) is deleted, the data marking changes from   to UNASSIGNED.

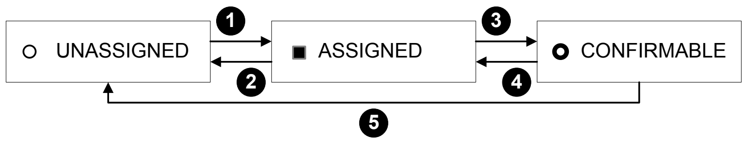


Figure 8.83: Data markings and their transitions

Initially, for each micro step its data marking corresponds to UNASSIGNED; i.e., when creating a micro process instance, no attribute value or relation is assigned. For this purpose, we extend Reaction Rule RR1 for initializing micro process instances (cf. Marking Rule RR1').

Marking Rule (RR1': Initializing micro steps with data markings):
Let dm = (name, OTypeSet, RelTypeSet) $\in$ DM be a data model and ds = (dm, OSet, RelSet) be a corresponding data structure. Further, let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data}) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Finally, let startState = (name, sMicStepSet) $\in$ StateSet be the start state of micProcInstance and startMicStep $\in$ MicStepSet the start micro step.

Assume that a new object instance o is created:
OSet = OSet $\cup$ {o} with o = (oid, oType, attrval) $\wedge$ o.oid = micProcInstance.oid.
Then the initial markings of micProcInstance are set as follows:

a)-j) see RR1 in Sect. 8.3

k) $\forall$ micStep $\in$ MicStepSet: M_{Data}(micStep) := UNASSIGNED;
i.e., for each micro step, its initial data marking is set to UNASSIGNED.

If a value for the attribute or relation the micro step refers to becomes available, the data marking changes from UNASSIGNED to ASSIGNED. If the attribute value or relation is deleted, in turn, data marking UNASSIGNED is re-allocated (cf. Reaction Rule RR7).

Reaction Rule (RR7: Assigning Data Markings):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data}) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

$\forall$ micStep=(ref, ValueSteps) $\in$ MicStepSet with ref $\neq$ NULL:

- ref=(attrType, attrValue) $\in$ Attr, $\Rightarrow$
$$M_{Data}(\text{micStep}) := \begin{cases} \text{ASSIGNED}, & \text{attrValue} \neq \text{NULL} \\ \text{UNASSIGNED}, & \text{attrValue} = \text{NULL} \end{cases}$$
i.e., when an attribute value becomes available for a micro step, its data marking changes to ASSIGNED. By contrast, if no attribute value is available, the data marking corresponds to UNASSIGNED.
- ref=(relType, soid, toid) $\in$ RelSet, $\Rightarrow$
$$M_{Data}(\text{micStep}) := \begin{cases} \text{ASSIGNED}, & \text{toid} \neq \text{NULL} \\ \text{UNASSIGNED}, & \text{toid} = \text{NULL} \end{cases}$$
i.e., when a relation becomes available for a micro step, its data marking changes to ASSIGNED. By contrast, if no relation is available, its data marking corresponds to UNASSIGNED.

When performing a backward jump, the data markings of the micro steps belonging to the reset region of the micro process instance are changed to CONFIRMABLE. For this purpose, the micro steps belonging to the target state are reset first (cf. Marking Rule MR16').

Marking Rule (MR16': Marking data markings as CONFIRMABLE):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data}) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

backTrans = (s, t) $\in$ BackTransSet with $M_{BackTrans}$(backTrans) = READY $\wedge$ t = (name, sMicStepSet) $\in$ StateSet, $\Rightarrow$
i.e., if a backward transition becomes marked as READY, the following re-markings apply:

a)-i) see MR16 in Sect. 8.6.3

j) $\forall$ backTrans=(s, t) $\in$ BackTransSet with $M_{BackTrans}$(backTrans) = READY $\wedge$ t=(name, sMicStepSet) $\in$ StateSet:
$\forall$ micStep=(ref, ValueSteps) $\in$ sMicStepSet: M_{Data}(micStep) := CONFIRMABLE;
i.e., data markings of all micro steps belonging to the target state are changed to CONFIRMABLE.

Second, the data markings of micro steps belonging to any subsequent state, which must be reset due to the backward jump, must be changed to CONFIRMABLE as well. For this purpose, we extend Marking Rule MR17. In particular, if a micro step is marked as WAITING (i.e., its incoming micro transitions are all marked as WAITING), the corresponding data marking must be changed to CONFIRMABLE (cf. Marking Rule MR17').

Marking Rule (MR17': Re-assigning data markings when resetting micro steps):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data}) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

$\forall$ micStep=(ref, ValueSteps) $\in$ sMicStepSet with state=(name, sMicStepSet) $\in$ StateSet $\wedge$ M_{State}(state) $\neq$ ACTIVATED:

a)-b) see MR17 in Sect. 8.6.4

c) $\forall$ micTrans $\in$ intrans(micStep) with $M_{MicTrans}$(micTrans) = WAITING: M_{Data}(micStep) := CONFIRMABLE;
i.e., if all incoming micro transitions of a micro step not belonging to the currently activated state are marked as WAITING, the data marking of this micro step changes to CONFIRMABLE.

Assume the data marking of a micro step is CONFIRMABLE. Then, this micro step may only be activated, when a user commitment becomes available for the attribute value or relation the micro step refers to. This is defined by *Execution Rule ER5*. For this purpose, we introduce function *committed*$_{data}$ which indicates whether or not a commitment for the respective attribute value or relation has been made (cf. Def. 31).

Definition 31 (Commitment for data markings):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data}) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

committed$_{data}$: MicStepSet $\mapsto$ BOOLEAN evaluates whether or not the required user commitment has been made for a particular micro step type.

Execution Rule (ER5: Committing data):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data}) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

$\forall$ micStep $\in$ MicStepSe with M_{Data}(micStep) = CONFIRMABLE $\wedge$ $committed_{data}$(micStep) = FALSE:
A user commitment is required;

i.e., if the data marking of a micro step corresponds to CONFIRMABLE, a commitment of any responsible user is required.

In turn, when a user provides the required commitment, the data marking is changed to ASSIGNED (cf. *Reaction Rule RR8*). Accordingly, the respective micro step can now be activated. To accomplish the latter, Reaction Rule RR2 needs to be extended; i.e., micro steps may only be marked as ACTIVATED, if their data marking corresponds to ASSIGNED (cf. Reaction Rule RR2”””).

Reaction Rule (RR8: Data committed):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data}) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

$\forall$ micStep=(ref, ValueSteps) $\in$ MicStepSet with M_{Data}(micStep) = CONFIRMABLE $\wedge$ committed_{data}(micStep) = TRUE:
    M_{Data}(micStep) := ASSIGNED;

i.e., the data marking of a micro step changes to ASSIGNED when a user commitment for its attribute value (or relation) becomes available.

Reaction Rule (RR2”””: Considering data markings when activating micro steps):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data}) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

a) $\forall$ micStep=(ref, VSteps) $\in$ MicStepSet with $M_{MicStep}$(micStep) = ENABLED $\wedge$ ref $\neq$ NULL $\wedge$ VSteps = $\emptyset$:
    (ref=(attrT, attrV) $\in$ Attr $\wedge$ attrV $\neq$ NULL) $\vee$ (ref=(relT, soid, toid) $\in$ RelSet $\wedge$ toid $\neq$ NULL)
    $\wedge$ M_{Data}(micStep) = ASSIGNED:
        $M_{MicStep}$:= ACTIVATED;
   i.e., all atomic micro steps will be marked as ACTIVATED if for the corresponding attribute (or relation) a value becomes available and the respective data marking corresponds to ASSIGNED.

b) $\forall$ micStep=(ref, VSteps) $\in$ MicStepSet with $M_{MicStep}$(micStep) = ENABLED $\wedge$ ref $\neq$ NULL:
    $\forall$ vStep $\in$ VSteps with constraint(valueStep) = TRUE $\wedge$ M_{Data}(micStep) = ASSIGNED:
        $M_{MicStep}$(vStep) := ACTIVATED;
   i.e., all value steps that evaluate to True will be marked as ACTIVATED when the data marking of the micro step they belong to corresponds to ASSIGNED.

c) $\forall$ micStep=(ref, VSteps) $\in$ MicStepSet with $M_{MicStep}$(micStep) = ENABLED $\wedge$ ref $\neq$ NULL $\wedge$ VSteps $\neq$ $\emptyset$:
    $\exists$ vStep $\in$ VSteps with $M_{MicStep}$(vStep) = ACTIVATED $\wedge$ M_{Data}(micStep) = ASSIGNED:
        $M_{MicStep}$(micStep) := ACTIVATED;
   i.e., a value-specific micro step will be re-marked from ENABLED to ACTIVATED if at least one of its value steps becomes marked as ACTIVATED and its data marking corresponds to ASSIGNED.

d) $\forall$ micStep=(ref, VSteps) $\in$ MicStepSet with
    $M_{MicStep}$(micStep) = ENABLED $\wedge$ ref $\neq$ NULL $\wedge$ VSteps $\neq$ $\emptyset$ $\wedge$ outtransCount(micStep) $\geq$ 1:
    (ref=(attrT, attrV) $\in$ Attr $\wedge$ attrV $\neq$ NULL) $\vee$ (ref=(relT, s, t) $\in$ RelSet $\wedge$ t $\neq$ NULL)
    $\wedge$ M_{Data}(micStep) = ASSIGNED, $\Rightarrow$
        $M_{MicStep}$:= ACTIVATED;
   i.e., a value-specific micro step will be re-marked from ENABLED to ACTIVATED when a value for its corresponding attribute becomes available and the micro step itself has at least one outgoing micro transition (in addition to the micro transitions outgoing from value steps of this micro step) and its data marking corresponds to ASSIGNED.

e) $\forall$ micStep=(ref, VSteps) $\in$ MicStepSet with $M_{MicStep}$(micStep) = ENABLED $\wedge$ ref $\neq$ NULL $\wedge$ VSteps $\neq$ $\emptyset$:
    ((ref=(attrT, attrV) $\in$ Attr $\wedge$ attrV $\neq$ NULL) $\vee$ (ref=(relT, s, t) $\in$ RelSet $\wedge$ t $\neq$ NULL)):
        outtransCount(micStep) = 0 $\wedge$ M_{Data}(micStep) = ASSIGNED $\wedge$
        $\nexists$ vStep $\in$ ValueSteps with $M_{MicStep}$(vStep) = ACTIVATED:
            $M_{MicStep}$(micStep) := BLOCKED;
   i.e., a value-specific micro step will be re-marked from ENABLED to BLOCKED if it has no outgoing micro

transition, none of its value steps is currently marked as ACTIVATED and its data marking corresponds to ASSIGNED.

f) $\forall$ micStep=(ref, ValueSteps) $\in$ MicStepSet with
ref = NULL $\wedge$ $M_{MicStep}$(micStep) = ENABLED $\wedge$ M_{Data}(micStep) = ASSIGNED:
$M_{MicStep}$(micStep) := ACTIVATED;
i.e., all empty micro steps that become marked as ENABLED will be immediately re-marked as ACTIVATED if their data marking corresponds to ASSIGNED.

Consider Execution Rule ER5 as well as Reaction Rules RR7 and RR8 as illustrated in Fig. 8.84. In order to re-execute a micro process instance after a backward jump, already assigned attribute values and relations, which are mandatorily required to proceed with the control flow, must be committed by responsible users. This enables authorized users to reuse already assigned attribute values and relations when re-executing a micro process instance.

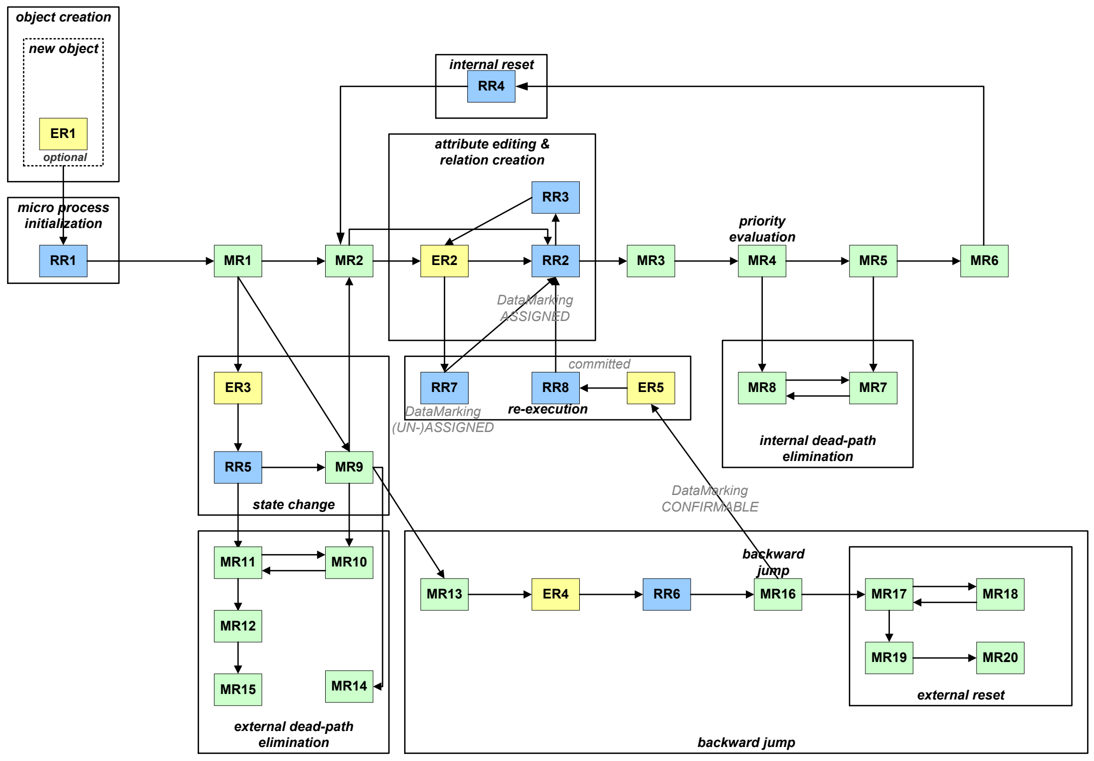


Figure 8.84: Rules for re-executing a micro process instance

8.7 Terminating Micro Process Instances

According to *Marking Rule MR21*, a micro process instance will terminate when one of its end states becomes activated (i.e., is marked as ACTIVATED).

Marking Rule (MR21: Termination of a micro process instance):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data}) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Further, let EndStates $\subset$ StateSet comprise all end states of micProc. Then:

$\exists$ state $\in$ EndStates with M_{State}(state) = ACTIVATED: $M_{MicProc}$(micProcInstance) := FINISHED;
i.e., a micro process instance will be re-marked from ACTIVATED to FINISHED as soon as one of its end states becomes marked as ACTIVATED.

Note that it should be even possible to re-activate a micro process instance if a backward jump from its end state to a previous state is performed; i.e., backward transitions originating from an end state may fire even if the micro process instance has been marked as CONFIRMED. For this purpose, the micro process instance must be re-marked as ACTIVATED when a respective backward transition is triggered (cf. Marking Rule MR16").

Example 8.54 (Re-activating micro process instances):
After having received a rejection, the `applicant` object instances reaches end state `rejected`. Consider now the case when an `applicant` applies for another `job` after having received the rejection. In this case, the `applicant` micro process instance changes from state `rejected` to state `pending`. Since state `pending` does not belong to the set of end states, the micro process instance corresponding to the `applicant` object instance must become re-activated.

Marking Rule (MR16": Re-activating micro process instances):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data}) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet); i.e., micProcInstance $\in$ micprocinstances(micProc). Further, let EndStates $\subset$ StateSet comprise all end states of micProcInstance. Then:

backTrans $\in$ BackTransSet with $M_{BackTrans}$(backTrans) = READY and backTrans.target $\in$ StateSet;
i.e., if a backward transition becomes marked as READY, the following re-markings apply:

a)-i) see MR16 in Sect. 8.6.3

j) see MR16' in Sect. 8.6.5

k) $\forall$ backTrans $\in$ BackTransSet with $M_{BackTrans}$(backTrans) = READY $\wedge$ source $\in$ EndStates:
$M_{MicProc}$(micProcInstance) := ACTIVATED;
i.e., if a backward jump originates from an end state, the micro process instance is re-activated and therefore marked as ACTIVATED.

Altogether, Fig. 8.85 lists all rules needed for the correct execution of a micro process instance. The rather high number is rules is required to provide the required flexibility for process execution, to realize the required data-driven execution paradigm, and to precisely specify the corresponding execution semantics at a fine-grained level needed to implement the approach. In particular, these rules have provided the foundation for implementing parts of the PHILharmonicFlows prototype (cf. Chapt. 14).

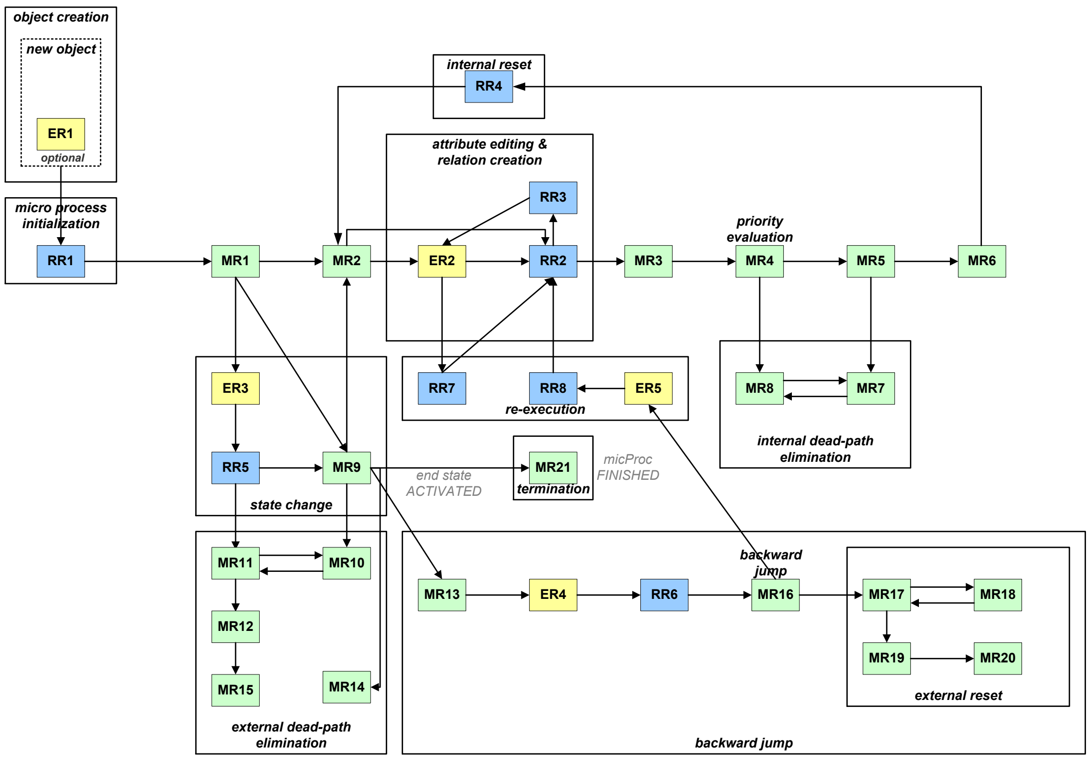


Figure 8.85: Comprehensive set of rules for executing a micro process instance

8.8 Task-oriented User View

As illustrated in Fig. 8.86, the main components relevant in the context of a micro process execution include states (and their execution responsibilities) as well as the micro steps and micro transitions. Based on the operational semantics introduced in this chapter, we are able to provide a generic approach for automatically generating *process-oriented views* at run-time.

In addition to the *data-oriented view* (cf. Sect. 6.5), in which users may access business data and involve business functions at any point in time, PHILharmonicFlows provides a *process-oriented view* as well. Based on the latter, upcoming activities can be assigned to the right users at the right point in time. In particular, for each user, PHILharmonicFlows automatically generates a *worklist* that comprises all mandatory activities this user is responsible for. Opposed to worklists in traditional PrMS (cf. Chapt. 4), however, the following issues must be considered:

- For each micro process type, usually, a high number of micro process instances exists (i.e., a large amount of object instances must be handled for a particular object type).
- Activities (generic and black-box ones) are aligned in respect to particular object types.
- Which activities are executable and which input fields are displayed when editing a particular object instance depends on the currently activated state of the corresponding micro process instance (as well as respective user permissions).

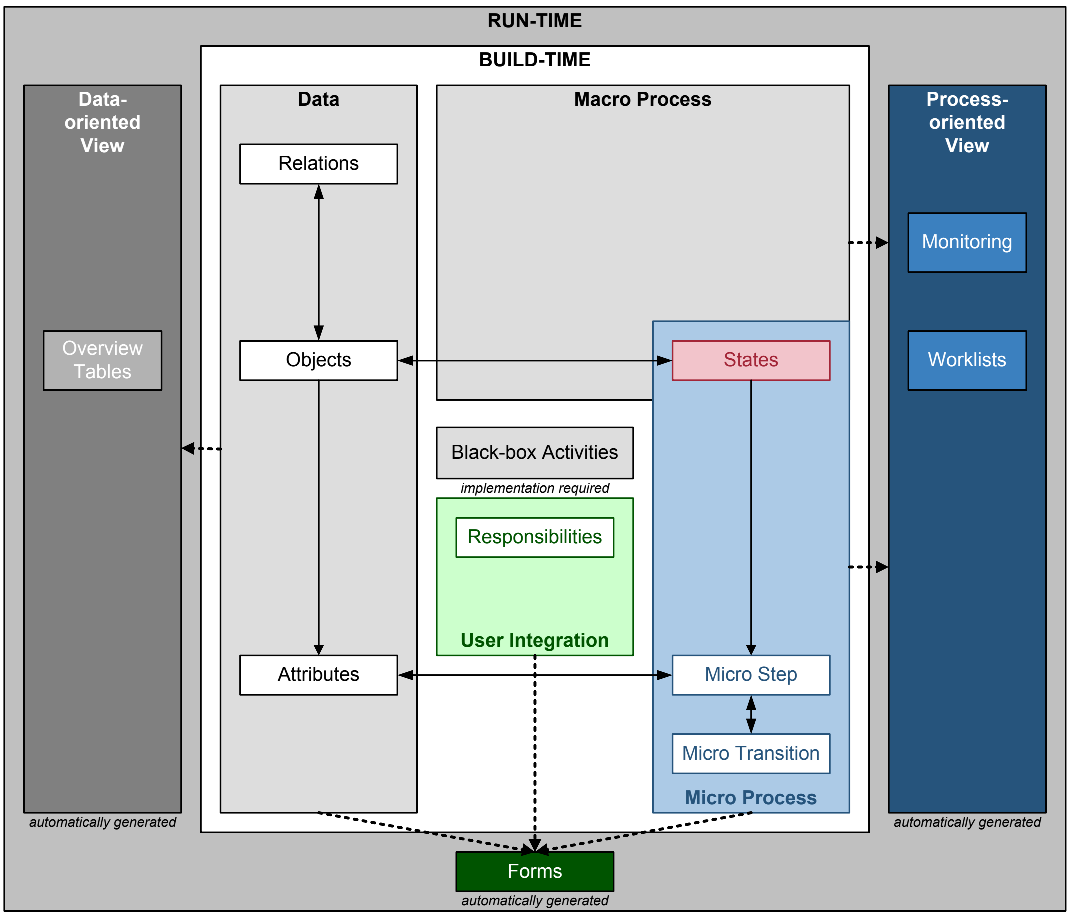
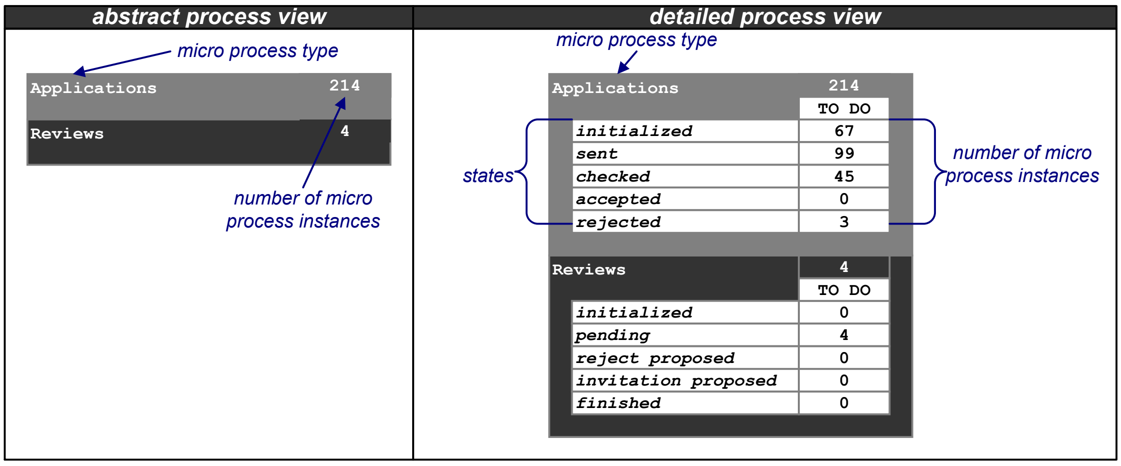


Figure 8.86: Process-oriented view in PHILharmonicFlows

Consequently, each user must execute one and the same activity for a potentially large number of micro process instances of same type. As a consequence, it is not always useful to directly list all activities, one entry following the other. Instead, all micro process instances and related tasks should be displayed in a more aggregated and comprehensive way. This aggregation (which is generated for each particular user) comprises two steps: First, a process-oriented view is generated for each object type. Second, for each of these views corresponding mandatory activities are then further categorized in respect to the states defined by the corresponding micro process types. Altogether, this results in process views that allow users to get a quick overview on their different working areas (i.e., the different micro process types) for which they must execute upcoming activities. Based on this aggregated view, corresponding object instances for which mandatory activities must be executed are then automatically selected and displayed in a corresponding overview table. This way, the process-oriented view works as filter selecting the affected object instances.

An example of a process-oriented user view is depicted in Fig. 8.87. Note that the micro process types and their user assignments (i.e., execution and transition responsibilities) constitute the fundamental basis for generating such views. Initially, for each micro process type, the corresponding number of micro process instances, for which the respective user must execute at least one mandatory activity, is listed. In this context, consider the abstract process view as illustrated in Fig. 8.87, according to it the respective user must execute tasks related to 214 applications.



Applications	214
Reviews	4

Applications	214
	TO DO
initialized	67
sent	99
checked	45
accepted	0
rejected	3

Reviews	4
	TO DO
initialized	0
pending	4
reject proposed	0
invitation proposed	0
finished	0

Figure 8.87: Process-oriented user view

As discussed, there exist two kinds of mandatory activities: on one hand, form-based activities for providing required attribute values (and relations) are required, on the other, as well as user commitments for activating subsequent states must be supported. Which activities are mandatory for a particular micro process instance again depends on its currently activated state; i.e., the current markings of the corresponding micro process instance. Therefore, the aggregated process view additionally allows categorizing micro process instances according to the states defined for the respective micro process type. For this purpose, a detailed process view can be realized for each object type (cf. Fig. 8.87).

Example 8.55 (Process-oriented user view):
Consider Fig. 8.88. The user role `employee` is assigned to state `pending` as well as to the explicit external micro transitions originating from the micro step types `reason`, `alternative job`, and `appraisal`. Since there exist four `review` instances, for which state `pending` is currently activated (cf. Fig. 8.90), PHILharmonicFlows summarizes them in the aggregated view of the responsible employee (cf. Fig. 8.87).

Using this aggregated view, users may list all corresponding instances. More precisely, when selecting a specific state, a respective overview table is automatically generated listing the associated object instances (cf. Fig. 8.89). This overview table can then be considered as worklist; i.e., the aggregated view acts like a filter in order to select all instances being in the specified state and for which the user must execute a mandatory activity. In this context, also note that for these object instances the required activities may be also executed in one go if desired by the user (cf. Sect. 9.3 for details).

Example 8.56 (Worklist):
Consider `review` instance `#1` in Fig. 8.89. For this instance, a value for attribute `proposal` is still missing. Hence, micro step `proposal` is marked as ENABLED (cf. Fig. 8.90). In turn, this requires the execution of a mandatory form-based activity to assign the missing value. This activity may be invoked using the edit-icon (cf. the pencil in Fig. 8.89). `Review` instance `#2`, in turn, either requires a value for attribute `reason` or a relation `alternative job`. Note that this attribute (or relation) may be also assigned by invoking a respective form (like the one for instance `#1`). In

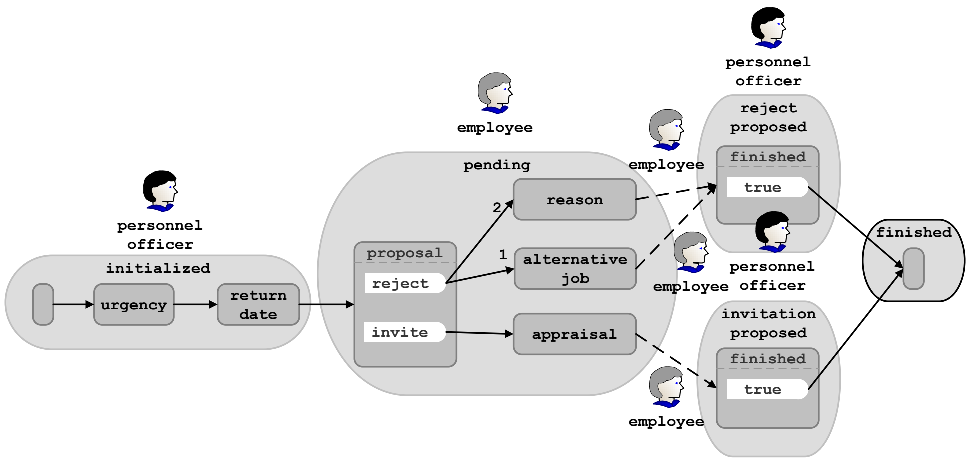
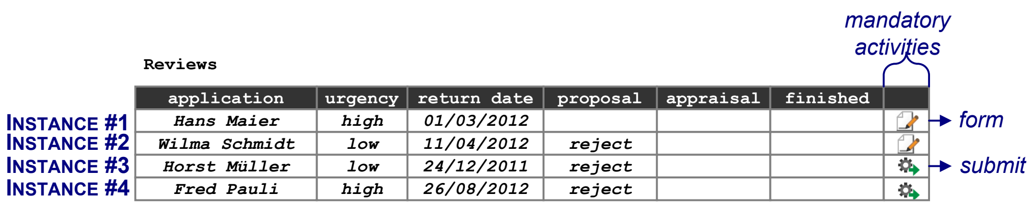


Figure 8.88: Examples of execution and transition responsibilities


mandatory activities
form
submit

Reviews

	application	urgency	return date	proposal	appraisal	finished	
INSTANCE #1	*Hans Maier*	*high*	*01/03/2012*				📝
INSTANCE #2	*Wilma Schmidt*	*low*	*11/04/2012*	*reject*			📝
INSTANCE #3	*Horst Müller*	*low*	*24/12/2011*	*reject*			⚙
INSTANCE #4	*Fred Pauli*	*high*	*26/08/2012*	*reject*			⚙

Figure 8.89: Worklist

turn, for `review` instances #3 and #4, respective attribute values have been already assigned. For these instances, the responsible user (i.e., the employee) must commit the activation of the subsequent state. More precisely, regarding instance #3 (cf. Fig. 8.90), the explicit external micro transition originating from micro step `reason` is marked as CONFIRMABLE. In turn, this indicates that a user commitment is mandatorily required. Opposed to this, for instance #4, a value for relation `alternative job` is selected (instead of setting attribute `reason`). Consequently, the explicit micro transition originating from micro step `alternative job` is re-marked as CONFIRMABLE (cf. Fig. 8.90).

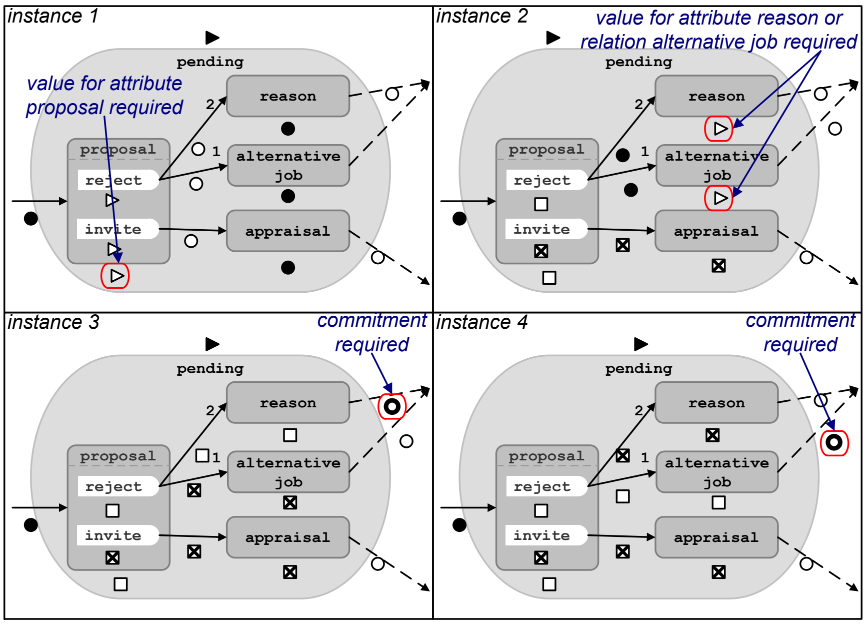


Figure 8.90: Review micro process instances in state pending

8.9 Summary

For properly initiating and executing micro process instances at run-time, PHILharmonicFlows is based on well-defined marking, execution, and reaction rules. In particular, these rules evaluate the attribute values of the corresponding object instances and enable the automatic generation of end-user components at run-time.

Opposed to traditional PrMS, each activity is related to a particular object instance and micro process instance respectively. In particular, this constitutes the foundation for providing aggregated views based on the different object and micro process types. Further, this allows for the proper handling of a large number of instances. Note that this does not impose any restriction with respect to the quick processing of upcoming activities. As will be shown in Sect. 9.3, PHILharmonicFlows additionally provides features for processing a particular activity for multiple object instances in one go. Furthermore, we do not disallow executing optional activities like traditional PrMS. Instead, in our approach these may be invoked for particular object instances based on the data-oriented perspective. Overall, PHILharmonicFlows provides data- and function-oriented features as known from (database) applications; i.e., we do not treat the process-oriented view completely independent from the data- and function-oriented ones. Instead, the process-oriented view acts as filter guiding users to find mandatorily required activities relevant for quickly processing running process instances.

9 Activities

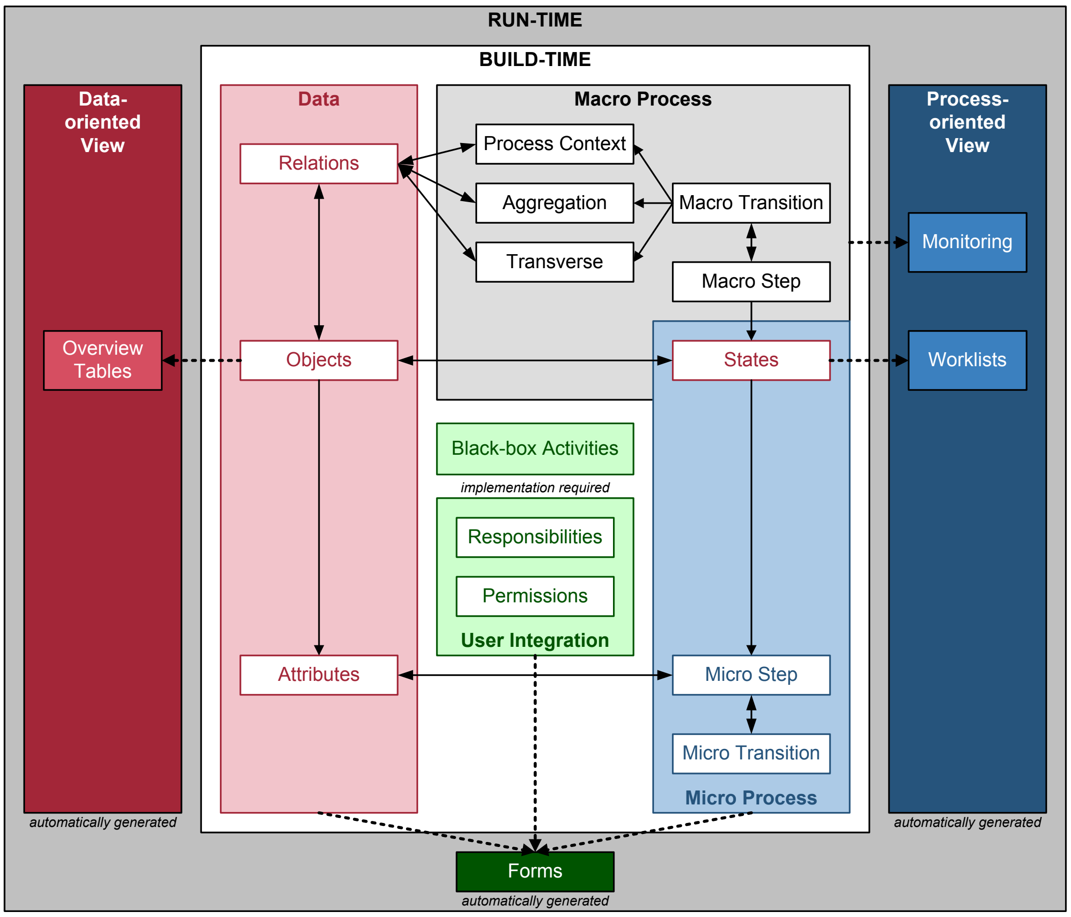


Figure 9.1: Activities in PHILharmonicFlows

Opposed to existing PrMS, this thesis does not pursue enterprise application integration [Kel02] as goal. Instead, it targets at the support of processes that are based on functions for managing and accessing data as stored in a DBMS. The thesis focuses on generic functionality for automatically generating data-oriented views (i.e., overview tables, data reports), process-oriented views (i.e., worklists), and form-based activities (i.e., user forms). However, one may also integrate common business functions, which are not form-based. For example, consider complex computations or the integration of legacy applications. However, the latter one restricted to corresponding business functions must be assignable to one particular business object; i.e., changes on additional business objects are possible but then out of control like in existing PrMS.

As discussed in Chapt. 4, in contemporary PrMS a business process is defined in terms of a set of activities and their execution constraints. When executing an activity, the attribute values of object instances may be changed. However, activity-centric PrMS do not control which object instances may be accessed and updated during activity execution. More precisely, the relationship between activities and object types is not made transparent; i.e., application data is managed by the invoked application components themselves. We denote respective activities as *black-box*. Thereby, each black-box activity requires a specific implementation.
As opposed to activity-centric PrMS, PHILharmonicFlows defines processes based on data rather than on black-box activities (cf. Chapt. 7). In particular, this allows for a well-defined granularity enabling the definition of process types in tight accordance with the one of object types and their relations. At build-time, each activity is put into the context of a particular object type. This enables us to automatically generate form-based activities (i.e., user forms) during run-time; i.e., activities for creating, editing, reading, and deleting object instances.
Generally, users should be allowed to choose the granularity of a particular activity. For example, when processing a particular object instance, users may want to additionally process related object instances. Furthermore, the progress of a micro process may not only depend on the availability of data (i.e., data-driven process execution), but also on explicit user decisions; i.e., even though required data is available, the execution of a particular micro process instance must not proceed until a user explicitly commits this.
PHILharmonicFlows automatically generates activities enabling such a flexible process control. Examples include the commitment of state changes and the application of backward jumps. For these control activities as well as for form-based activities, a generic implementation is provided; i.e., these activities are *automatically generated* at run-time and integrated with overview tables and worklists. Accordingly, we denote them as *generic activities*. Finally, PHILharmonicFlows allows processing multiple object instances in one go. We denote this as *batch execution*.

9.1 Generic Activities

PHILharmonicFlows provides *generic activities* for editing, reading, creating, and deleting object instances (i.e., form-based activities) as well as for enabling process control. The latter includes user commitments and backward jumps.

9.1.1 Editing Object Instances

When a state becomes enabled during the processing of a particular object instance, usually, a corresponding form-based activity for entering required attribute values is automatically generated (cf. Fig. 9.2). This activity is then assigned to the worklists of authorized users (cf. Def. 19). In general, we denote such activities as *mandatory*. Mandatory activities are required to proceed with the flow of control; i.e., to set required attribute values and relations.

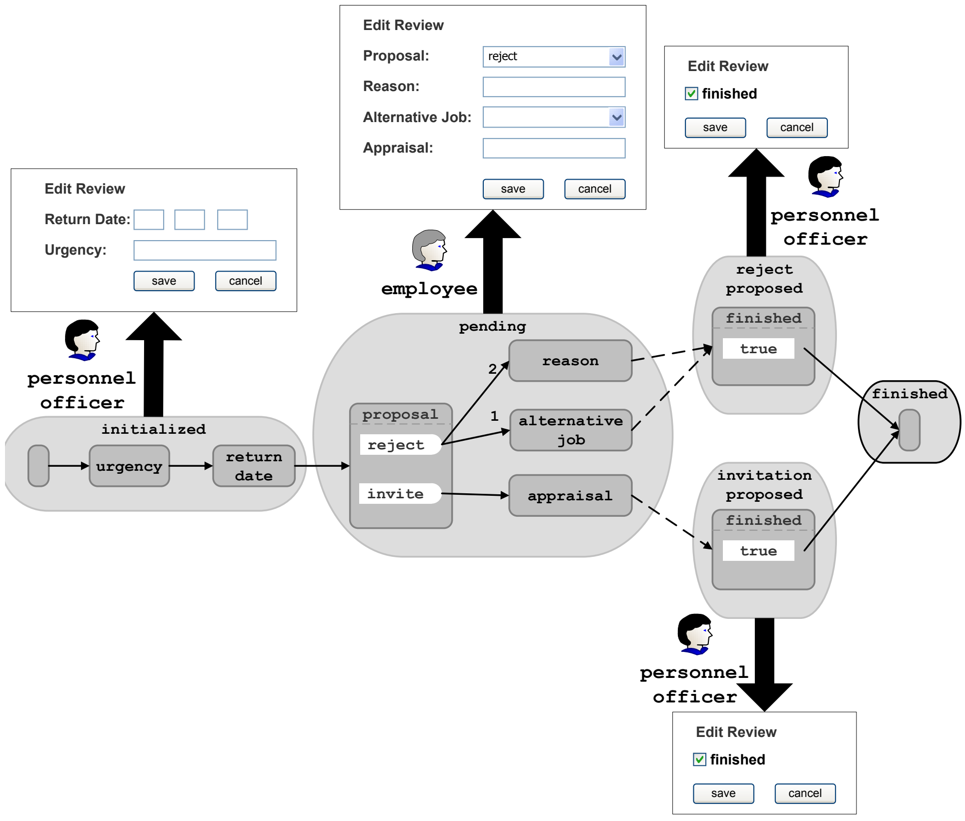


Figure 9.2: Generating mandatory activities

As illustrated in Fig. 9.3, the states of a micro process instance may be associated with different user forms. Thereby, a generated user form comprises one *input field* for each attribute referenced by any micro step of the currently considered state. Generally, we denote each of these input fields as *mandatory action*. Moreover, when processing a form, the user is guided in setting the attribute values or relations required (e.g., by highlighting selectable input fields). To allow for this guidance, the state-internal logic of the respective micro process instance is taken into account (cf. Chapt. 8).

To enable an integrated and flexible access to business data and business processes, users should be allowed to optionally read or write object attributes even if they are currently not executing any mandatory activity; i.e., users should be allowed to execute *optional activities* (cf. Prop. 8). However, which object attributes may be optionally read or written again depends on the current progress of the considered process instance as well; i.e., whether a user may access certain object attributes depends on the state of the corresponding micro process instance (cf. Prop. 19). Furthermore, optional activities may differ from user to user; i.e., different

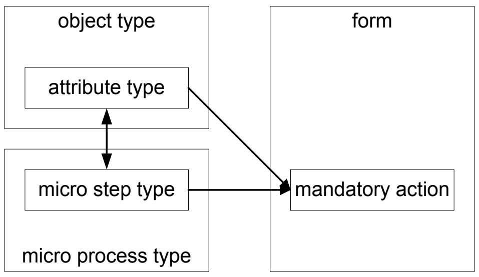


Figure 9.3: Mandatory actions

users may read/write different attribute values in a particular state of an object instance. As a consequence, the number of forms required for the processing of an object instance depends on the numbers of defined object states and involved user roles.

Example 9.1 (Variability of user forms):
Considering our example from Chapt. 8, a `review` object instance comprises five different states; i.e., `initialized`, `pending`, `reject proposed`, `invitation proposed`, and `finished`. Further, these states refer to three different user roles (i.e., `personnel officer`, `employee`, and `manager`). Altogether, this might require fifteen different user forms.

To enable context-awareness of form-based activities (cf. Prop. 14) when processing a particular object instance, it should be possible to edit attribute values of related object instances if desired. Note that this further multiplies the number of required user forms; i.e., for such related object instances, their different processing states and corresponding user roles must be considered as well. Consequently, as illustrated in Fig. 9.4, the number of required user forms depends on the number of processing states, involved user roles, and related object instances.

Example 9.2 (Vertical flexibility of form-based activities):
When processing a `review` form, users may want to concurrently access the corresponding `application` within the same form. Like the `review` object type, an `application` comprises five states. Consequently, for a particular user, up to twenty-five different user forms may have to be generated.

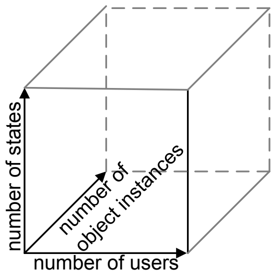


Figure 9.4: Factors determining the controls of form-based activities

Such a fine-grained access control is required by many applications. To enable it, PHILharmonicFlows provides an *authorization table* for each object type (cf. Fig. 9.5). In this context, different permissions for reading and writing object attribute values may be granted to the different user roles.[1] Opposed to existing approaches, PHILharmonicFlows additionally considers the states of micro process instances as defined by its corresponding type. More precisely, different user roles may own different read or write permissions in different states (cf. Def. 32a+b).

Definition 32 (Read and write permissions):
Let dm = (name, OTypeSet, RelTypeSet) be a data model and oType = (name, AttrTypeSet) $\in$ OTypeSet be an object type with corresponding micro process type micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet, BackTransTypeSet). Further, let UserRoles be the set of all defined user roles.
Optional read and write permissions are then defined as follows:

A **permission perm = (refType, stateType, role, mode)** is a tuple where

- refType $\in$ AttrTypeSet $\cup$ RelTypeSet is an attribute type or a relation type
  with refType $\in$ RelTypeSet, $\Rightarrow$ refType.source = oType
- stateType $\in$ StateTypeSet is a state type.
- role $\in$ UserRoles is a user role.
- mode is either r (read permission) or w (write permission).

ReadPermissions corresponds to the set of all definable read permissions (i.e., mode = r) and **WritePermissions** corresponds to the set of all definable write permissions (i.e., mode = w).

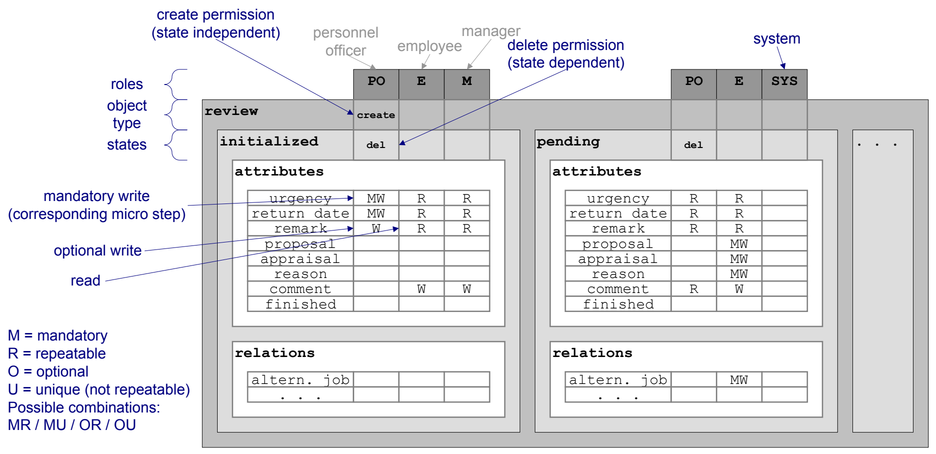


Figure 9.5: Authorization table

Example 9.3 (Read and write permissions):
As illustrated in Fig. 9.5, the `personnel officer (PO)` may still update attributes `urgency`, `return date`, and `remark` even if the subsequent state `pending` has been already activated. In addition, he may read attribute `comment`.

Generally, the permissions granted to selected attributes of an object type constitute the foundation for generating forms at run-time. As illustrated in Fig. 9.6, in this context, for each attribute

[1]If a user owns several roles, he gets the permissions of all granted roles at the same time. In addition, PHILharmonicFlows provides features to select only a subset of granted roles during the login process.

of an object type one potential *action* exists (i.e., to write or read the attribute). Whether such an action is actually added to a particular user form, however, depends on the user's permissions. If the user may write a particular attribute, a corresponding input field is added to the form allowing him or her to optionally assign or change the respective attribute value. In turn, to enable read access, the current value of the attribute is displayed in a data field to the user when processing the form. In general, depending of their permissions, users may optionally read and write attribute values of a particular object instance. According to *Execution Rule ER6*, these actions are denoted as *optional* ones.

Execution Rule (ER6: Optional data access):
Let dm = (name, OTypeSet, RelTypeSet) be a data model and ds = (dm, OSet, RelSet) be a corresponding data structure. Further, let micProcInstance be a micro process instance of type micProc; i.e., micProcInstance ∈ micprocinstances(micProc). Finally, let o = (oid, oType, attrval) be the object instance micProcInstance refers to; i.e., o.oid = micProcInstance.oid. Then:

a) ∀ readPerm ∈ ReadPermissions:
∃ readPerm.stateType ∈ StateSet with M_{State}(readPerm.stateType) = ACTIVATED, ⇒

- readPerm.refType ∈ AttrTypes, ⇒
  attrval(readPerm.refType) may be optionally read by users owning r.
- readPerm.refType ∈ RelTypeSet, ⇒
  ∃ rel ∈ RelSet with rel.soid = o.oid and rel.relType ∈ RelTypeSet:
  rel.toid.attrval(label(rel.relType.target)) may be optionally read by users owning r.

b) ∀ writePerm ∈ WritePermissions:
∃ writePerm.stateType ∈ StateSet with M_{State}(writePerm.stateType) = ACTIVATED, ⇒

- writePerm.refType ∈ AttrTypes, ⇒
  attrval(writePerm.refType) may be optionally written by users owning r.
- writePerm.refType ∈ RelTypeSet, ⇒
  ∃ rel ∈ RelSet with rel.soid = o.oid and rel.relType ∈ RelTypeSet:
  rel.toid.attrval(label(rel.relType.target)) may be optionally written by users owning r.

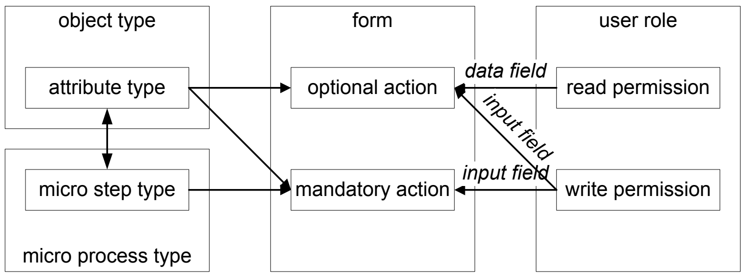


Figure 9.6: Optional actions

As illustrated in Fig. 9.7, for usability reasons, mandatory and optional actions are usually not treated separately, but combined in one and the same form. Generally, a form may comprise (1) optional actions solely, (2) mandatory actions solely, or (3) optional as well as mandatory actions. We therefore distinguish between *optional* and *mandatory activities*. More precisely, a *mandatory activity* refers to at least one object attribute for which a value must be set. In turn, an *optional activity* only comprises actions not relevant for progressing with the process.

From the viewpoint of the end-user, for a particular object instance an optional activity may be invoked by a user if he owns the permissions to optionally write at least one attribute of the

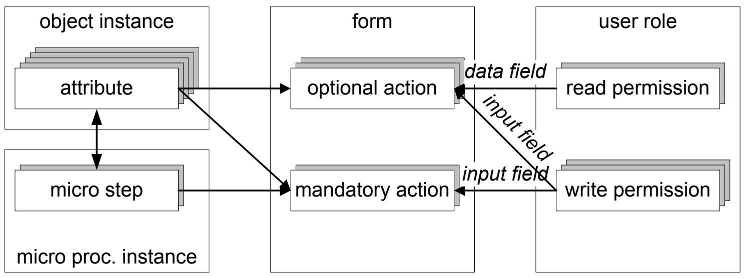


Figure 9.7: Mandatory and optional actions

object instance. In turn, a mandatory activity becomes enabled if there exist pending mandatory actions for the respective object instance and the user owns the required permissions. If a user may invoke both optional and mandatory activities for a particular object instance at a certain point in time, corresponding actions are composed in one form. Thereby, mandatory attributes are highlighted, while optional ones are not; i.e., certain actions of the activity are considered as being more important than others. This allows users to do the work they must do and the one they might want to do in a single step.

Another challenge is to ensure that *process authorization* (i.e., user assignment for execution mandatory activities) complies with *data authorization* (cf. Prop. 18). More precisely, each user who must execute a mandatory activity should own corresponding write permissions for changing the attribute values required for proceeding with process execution. This is automatically ensured by PHILharmonicFlows; i.e., each user owning the *execution responsibility* for a particular state, automatically gets write permissions for the attributes referenced by a micro step of this state. Optional permissions may then be additionally granted to the respective user role by adjusting the generated authorization table accordingly.

Example 9.4 (Automatically granted write permissions):
Consider Fig. 9.8. Since the `personnel officer` owns the execution responsibility for state `initialized`, write permissions for attributes `urgency` and `return date`, which are referenced by the micro steps of state `initialized`, are automatically granted.

Example 9.5 (Adjusted permissions):
Write permissions for attributes `urgency` and `return date` are automatically granted for the `personnel officer`. Since there is no micro step type referring to the `remark` attribute type, however, write permission for attribute `remark` must be manually granted (Fig. 9.8).

As discussed, for users owning the execution responsibility of a particular state, a mandatory activity is assigned to their worklist. In this context, we denote write permissions automatically assigned for execution responsibilities as *mandatory* (cf. Def. 33). This allows us to further differentiate between process authorization (i.e., user assignment through execution responsibilities) and data authorization (i.e., permissions). Only for users with mandatory write permissions, a mandatory activity is assigned to their worklists (cf. 8.8).

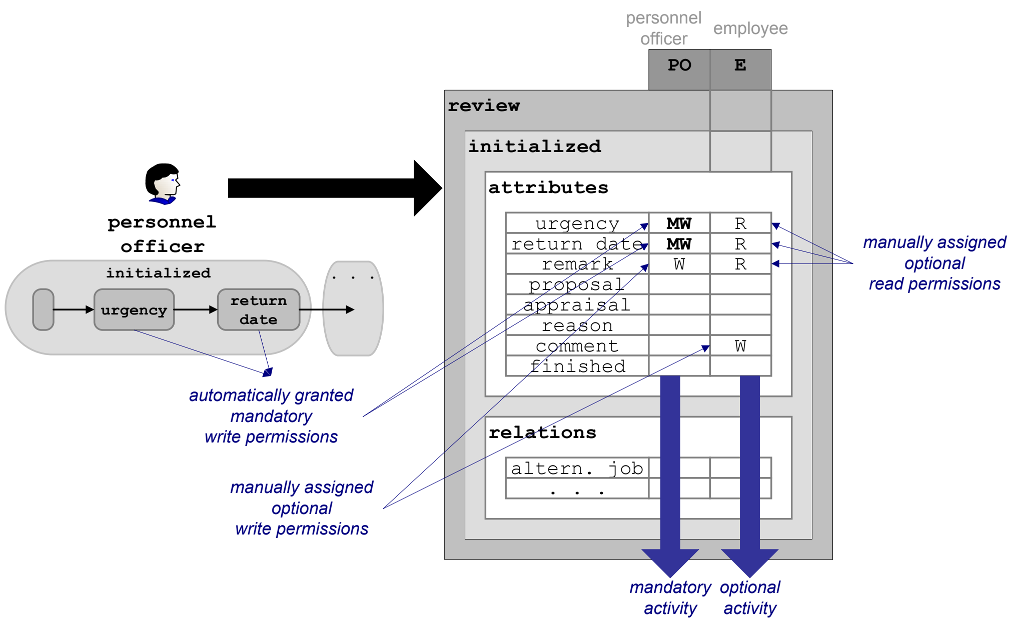


Figure 9.8: Different permissions and resulting activity types

Definition 33 (Mandatory write permissions):
Function **mandatory: WritePermissions** $\mapsto$ **BOOLEAN** defines whether a particular write permission writePerm = (refType, stateType, r, mode) $\in$ WritePermissions is mandatory for r;
i.e., users owning the specified role must then mandatorily assign a value to the attribute or relation when the respective state becomes activated at run-time.

When a state becomes activated, a mandatory activity for editing the required attribute values is automatically assigned to the worklist of users owning corresponding mandatory write permissions. When additionally considering the introduced permissions, it is possible to optionally edit additional attributes not referenced by a micro step of the currently activated state.

Example 9.6 (Optional input fields):
As illustrated in Fig. 9.9, the `personnel officer` may edit attribute `remark` when initiating a `review`.

Even users currently not executing a mandatory activity may execute optional activities.

Example 9.7 (Optional activities):
While the `personnel officer` initiates the `review`, the `employee` may optionally read attributes `urgency`, `return date`, and `remark` and optionally write attribute `comment` (cf. Fig. 9.9).

Taking the different permissions into account, overview tables are adapted accordingly. In particular, which object instances (rows) are editable and which attribute values (columns) are displayed depends on the permissions of the respective user accessing the overview table. Since

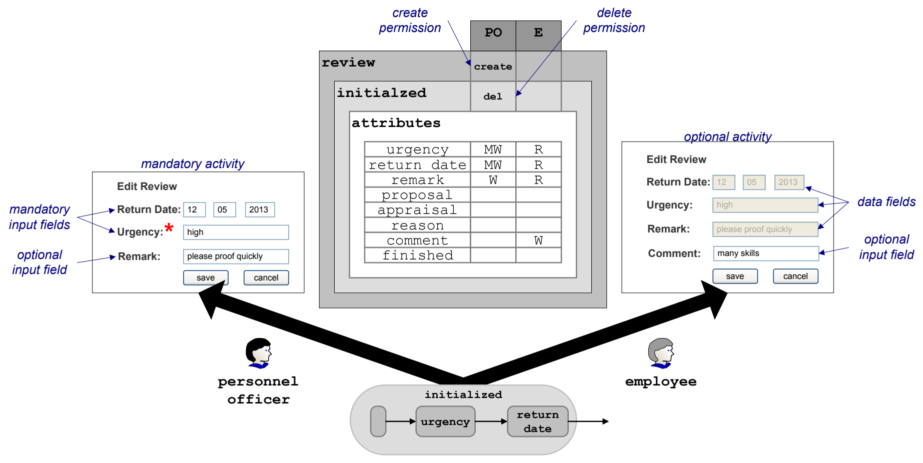


Figure 9.9: Generating form-based activities

for different object instances different states may be activated, both activities and displayed attribute values may vary from object instance to object instance. For the sake of transparency, the currently activated state of object instances should be displayed as well (cf. Fig. 9.10). Otherwise, it would not be transparent for users why a particular object instance is editable, while others are not. As illustrated in Fig. 9.10, taking the currently activated state of an object instance into account, only users owning at least one write permission may invoke the corresponding user form for editing attribute values.

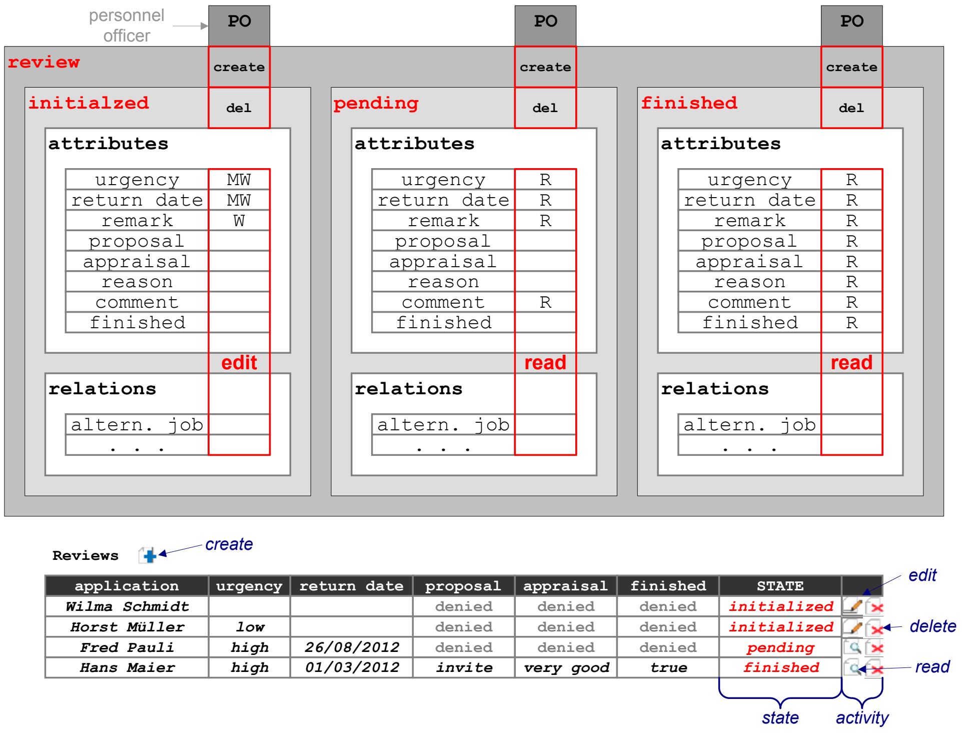
Example 9.8 (Executable activities vary from object instance to object instance):
A `personnel officer` only has write permissions in state `initialized`. Thus, only object instances for which state `initialized` is currently activated are editable.

In turn, for read permissions, it usually does not make sense to display all attribute values of an object instance in the overview table. For users owning solely read permissions in the current state of an object instance, therefore, a corresponding activity for displaying detailed object information (e.g., all attribute values) is added. In turn, if a user also owns write permissions, detailed object information is displayed when invoking the respective user form (i.e., edit activity).

Example 9.9 (Detailed view on object instances):
In states `pending` and `finished`, the `personnel officer` only owns read permissions (cf. Fig. 9.10). Hence, an activity displaying detailed object information may be invoked. In turn, in state `initialized`, write permissions are granted as well. Accordingly, an edit activity is invoked when selecting the respective object instance.

An overview table might contain columns (i.e., attribute values) for which the respective user does not own read permissions in all states. Consequently, certain attribute values need to be hidden for selected object instances in particular states. In order to differentiate them from attributes without currently assigned value, PHILharmonicFlows indicates it explicitly if access is denied (cf. Fig. 9.10).

Example 9.10 (Varying read permissions):
The `personnel officer` must not read attribute `comment` if state `initialized` is currently activated. In all other states, however, the `comment` must be read.

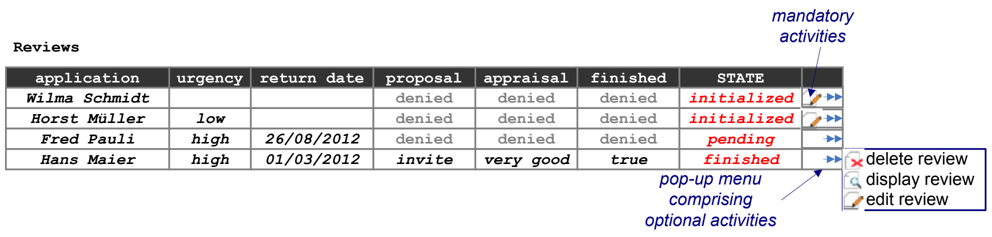


application	urgency	return date	proposal	appraisal	finished	STATE
Wilma Schmidt			denied	denied	denied	*initialized*
Horst Müller	*low*		denied	denied	denied	*initialized*
Fred Pauli	*high*	*26/08/2012*	denied	denied	denied	*pending*
Hans Maier	*high*	*01/03/2012*	*invite*	*very good*	*true*	*finished*

Figure 9.10: Adapting overview tables

As discussed in Sect. 8.8, data- and process-oriented views shall be tightly integrated. In particular, based on the task-oriented user view (i.e., by selecting a particular state), invoked object instances are listed in a corresponding overview table. More precisely, the task-oriented view acts as filter in this context. In turn, overview tables contain a column listing the currently activated state of the respective micro process instance corresponding to a particular object instance. Based on overview tables, mandatory as well as optional activities may be invoked; e.g., consider the separate column as illustrated in Fig. 9.10. In this context, PHILharmonicFlows differentiates between mandatory and optional activities. Regarding the overview table in Fig. 9.10, optional activities are displayed using a pop-up menu (see the double-arrow in Fig. 9.10).

Altogether, editing attribute values (and relations) is not only possible when the respective value is mandatorily required for micro process execution (cf. Execution Rule ER2). In addition, users may optionally edit object instances (and relations) if required permissions are available (cf. Execution Rule ER6).



Reviews

application	urgency	return date	proposal	appraisal	finished	STATE
Wilma Schmidt			denied	denied	denied	initialized
Horst Müller	low		denied	denied	denied	initialized
Fred Pauli	high	26/08/2012	denied	denied	denied	pending
Hans Maier	high	01/03/2012	invite	very good	true	finished

Figure 9.11: Differentiating between mandatory and optional activities

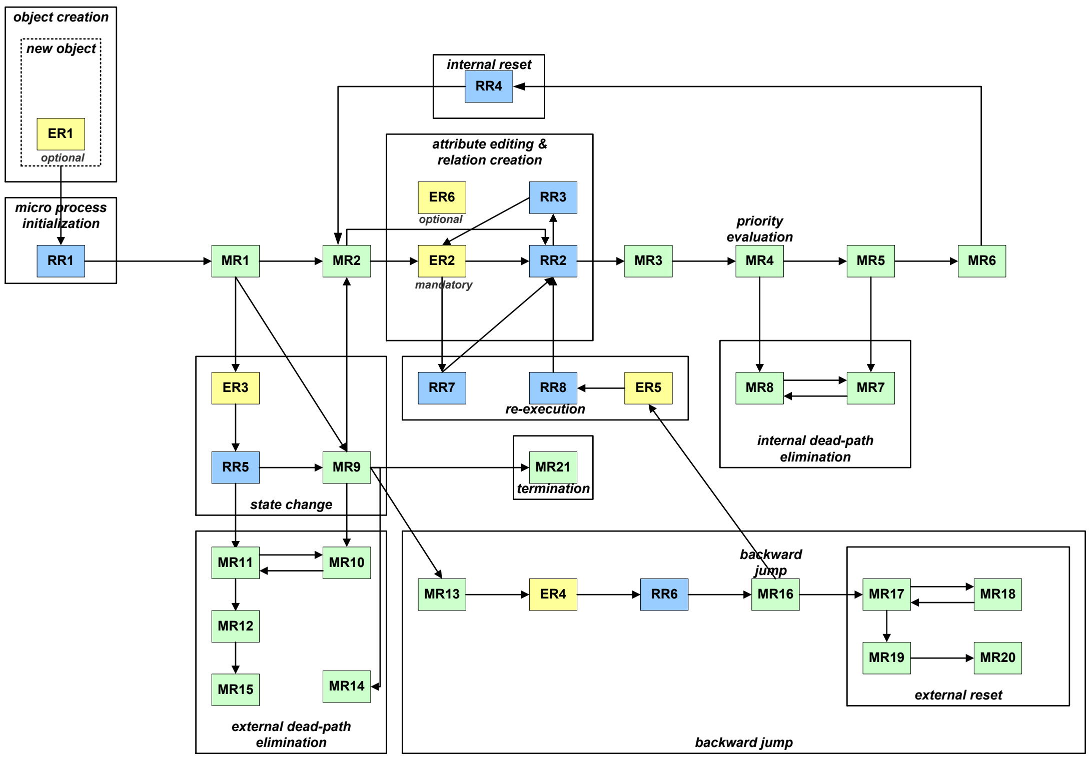


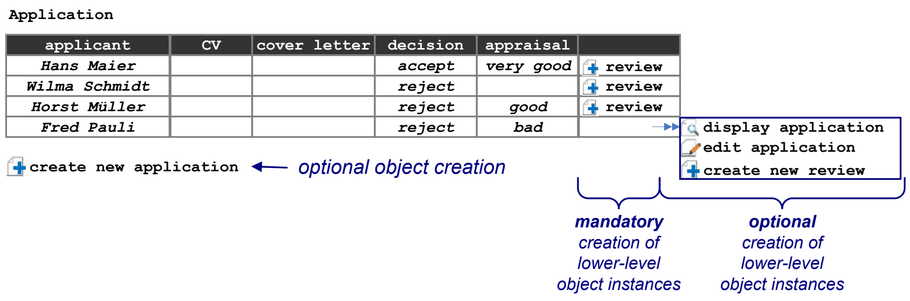
Figure 9.12: Rules for editing object instances

9.1.2 Creating Object Instances

Create permissions (cf. Def. 25) may be assigned based on authorization tables as well. However, they are independent from any state. Note that the respective object instance does not exist when applying this permission. Hence, create permissions are directly assigned on the top-level of the authorization table, the first row does not distinguishing between different states, as depicted in Fig. 9.5.

Object instances may be created by any user owning corresponding create permissions. For each newly created object instance, a corresponding micro process instance is automatically initialized (cf. Reaction Rule RR1). Thereby, the start state of the micro process instance becomes activated. When creating object instances, the respective user is enabled to assign attribute values and relations based on an automatically generated user form (cf. Sect. 9.1.1).

This user form can be invoked and executed by selecting a respective item from the overview table generated for the object type (cf. Fig. 9.13). Which input and data fields are displayed in this context depends on the read and write permissions granted for the respective user in respect to the start state of the corresponding micro process.


Application

applicant	CV	cover letter	decision	appraisal	
Hans Maier			*accept*	*very good*	review
Wilma Schmidt			*reject*		review
Horst Müller			*reject*	*good*	review
Fred Pauli			*reject*	*bad*	

display application
edit application
create new review
create new application
optional object creation
mandatory creation of lower-level object instances
optional creation of lower-level object instances

Figure 9.13: Overview table enabling the creation of new object instances

Example 9.11 (Creating object instances):
Based on the definition of the `application` object type, PHILharmonicFlows automatically generates an overview table listing existing `application` object instances (cf. Fig. 9.13). Since the `personnel officer` owns the create permission for `application` object instances (cf. Fig. 9.14), a corresponding item for involving this creation activity is displayed in the overview table (see Fig. 9.13A). Note that this business function is required, to enter `applications` send by mail. Using this item, a corresponding micro process instance is generated and its start state `initialized` becomes activated. In this state, the `personnel officer` owns write permissions for relation `job offer` as well as for attributes `cover letter`, `priority`, and `remark`. Hence, the generated user form for creating a new `application` object instance comprises input fields for assigning a relation to a `job offer` as well as for editing the attributes `cover letter`, `priority`, and `remark` (cf. Fig. 9.14).

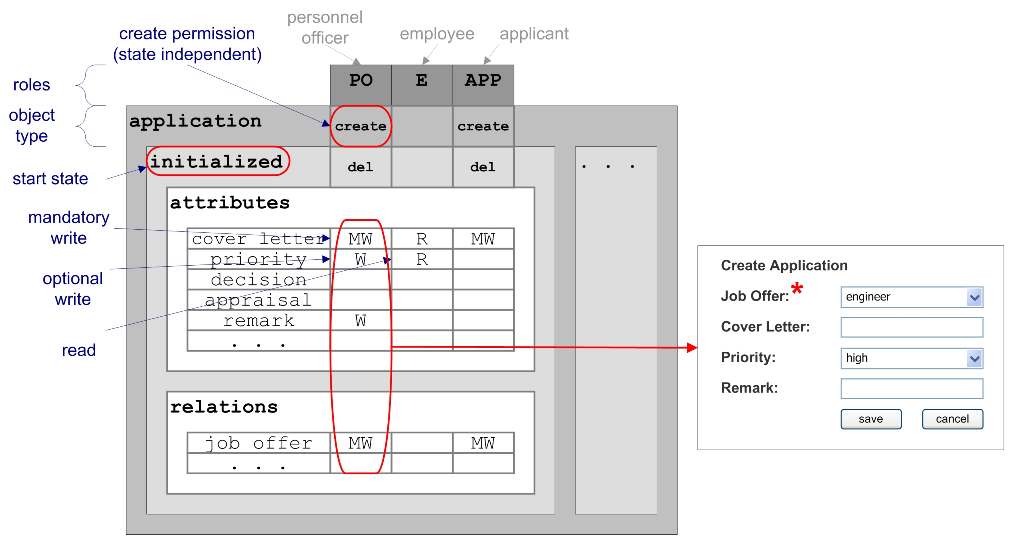


Figure 9.14: Authorization table of the application object type

In general, object instances must not be created independently from each other. Instead, lower-level object instances are required in the context of higher-level ones. For this reason, PHILharmonicFlows allows creating an object instance directly in relation to a higher-level one; e.g., to create an `application` object instance directly in relation to the `job offer` object instance the `application` shall refer to. Such activities are dynamically integrated into the overview table and may be executed for each listed, higher-level object instance (cf. Fig. 9.13). When executing this activity, a corresponding relation is automatically assigned; i.e., the relation targets at the respective higher-level object instance the activity was started with.

Example 9.12 (Creating lower-level object instances for a higher-level one):
For each `application` object instance, multiple `review` object instances must be created. As illustrated in Fig. 9.13, the corresponding activity for creating object instances may be directly executed within the overview table that lists the `application` object instances. For this purpose, a respective item is added for each `application` object instance. Since it corresponds to an optional activity, the item is added to the context menu (cf. Sect. 9.1).

When creating object instances, it must be ensured that the cardinality constraints specified for the different object relations are met. For this purpose, a *creation context* is created for each object instance of the target object type. Consider Fig. 9.15. For each relation, a source and a target object type is defined (cf. Def. 6). At run-time, for each instance of the target object type, a corresponding creation context is managed. In particular, this context counts all lower-level instances of the source object type referencing this target object instance (cf. Def. 34).

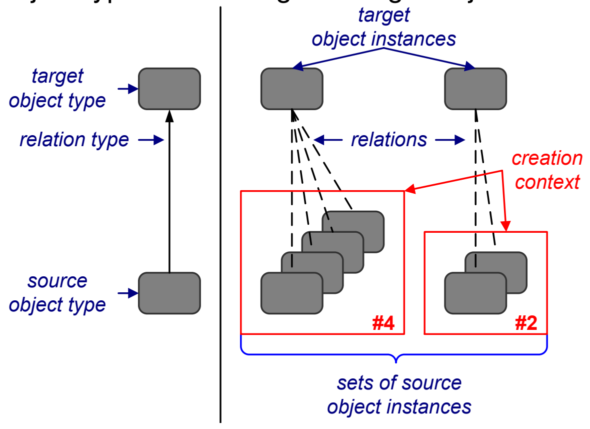


Figure 9.15: Creation contexts

Definition 34 (Creation context):
Let dm = (name, OTypeSet, RelTypeSet) be a data model and ds = (dm, OSet, RelSet) be a corresponding data structure. Then:

A **creation context cc = (relType, toid, SoidSet)** is a tuple where

- relType ∈ RelTypeSet is a relation type.
- toid is the identifier of the object instance o ∈ OSet of the target object type the creation context cc belongs to.
- SoidSet is a finite set of identifiers representing the instances of the source object type that reference the object instance identified by toid:
  ∀ soid ∈ SoidSet: ∃ rel ∈ RelSet with rel.rType = relType ∧ rel.source = soid ∧ rel.target = toid.

CreationContexts represents the set of all creation contexts. Furthermore, **CreationContextSet$_o$** corresponds to the set of creation contexts existing for a particular target object instance.
To capture respective information, in the following, an object instance correspionds to a tuple **o = (oid, oType, attrval, CreationContextSet)**; i.e., the creation contexts are added.

Whether or not additional lower-level source object instances may be created depends on the number of already existing ones. To additionally consider object creation, we must extend the already defined operational semantics accordingly. For this purpose, we introduce a number of run-time markings for creation contexts. Based on these markings, it can be expressed when

- object instances must be mandatorily created,
- object instances must not be created anymore, and
- object instances may be optionally created.

Based on respective markings we can additionally determine to which higher-level target object instances relations may be assigned afterwards. In particular, when editing existing object instances, corresponding user forms may comprise components (e.g., comboboxes) for selecting higher-level object instances to be referenced. This set must be dynamically updated when reaching the maximum number of lower-level object instances for particular object instances. Then, these object instances must be removed or deactivated from the respective form component enabling the creation of respective relations. For this purpose, each creation context is either marked as ACTIVATED, CONFIRMED, or BLOCKED (cf. Def. 35).

Definition 35 (Markings for creation contexts):
Let dm = (name, OTypeSet, RelTypeSet) be a data model and ds = (dm, OSet, RelSet) be a corresponding data structure. Then:

M_{Cc}: CreationContexts $\mapsto$ {ACTIVATED, CONFIRMED, BLOCKED} assigns to a creation context cc its current marking (see Tab. 9.1).
In the following, a data structure is represented as a tuple **ds = (dm, OSet, RelSet, M_{Cc})**.

These markings have the following meaning (cf. Tab. 9.1):

Marking	Label	Description
▶	ACTIVATED	The minimum number of required lower-level source object instances is not reached. Additional object instances must still be created.
■	CONFIRMED	The minimum number of required lower-level source object instances is reached, but not the maximum one. Hence, additional object instances may be optionally created, although this is not mandatory.
▷	BLOCKED	The maximum number of required lower-level source object instances is reached. Additional object instances must not be created.

Table 9.1: Creation context markings

At run-time, a corresponding creation context is initialized for each defined relation type when a new object instance is created. When creating an object instance, no lower-level source object instance referencing it exists (i.e., relations between object instances may only be created when both the source and the target object instance exist. The marking of respective creation contexts therefore depends on the minimum cardinality specified for the corresponding relation type. More precisely, if a minimum cardinality is defined, the creation context will be initially marked as ACTIVATED. Otherwise, it will be marked as CONFIRMED (cf. Reaction Rule RR1”).

Reaction Rule (RR1": Initializing creation contexts):
Let dm = (name, OTypeSet, RelTypeSet) ∈ DM be a data model and ds = (dm, OSet, RelSet, M_{Cc}) be a data structure of dm. Further, let micProcInstance be a micro process instance of type micProc; i.e., micProcInstance ∈ micprocinstances(micProc).

When creating a new object instance:
OSet = OSet ∪ {o} with o = (micProcInstance.oid, oType, attrval, CreationContextSet),
the initial markings of its creation context instances are as follows:

a)-j) see RR1 in Sect. 8.3

k) see RR1' in Sect. 8.6

l) ∀ cc ∈ CreationContextSet with cc.relType ∈ RelTypeSet:

$$M_{Cc} := \begin{cases} \text{ACTIVATED}, & |\text{cc.SoidSet}| < \text{cc.relType.min} \\ \text{CONFIRMED}, & \text{else} \end{cases}$$

i.e., all creation contexts are initially marked as CONFIRMED or ACTIVATED. In the latter case, for the corresponding relation type a minimum cardinality is defined.

If a relation is assigned later on, the affected creation contexts are updated. According to *Reaction Rule RR9*, the source object instance is then assigned to the creation context of the target object instance. Using *Marking Rule MR22*, the marking of the creation context is updated accordingly.

Reaction Rule (RR9: Updating creation contexts):
Let dm = (name, OTypeSet, RelTypeSet) be a data model and ds = (dm, OSet, RelSet, M_{Cc}) be a corresponding data structure.

When assigning a new relation (i.e., RelSet = RelSet ∪ {rel}) then:

∀ cc ∈ CreationContexts with cc.toid = rel.target: cc.SoidSet := cc.SoidSet ∪ {rel.source};

i.e., when assigning a new relation, its source object instance is assigned to the creation contexts (of the respective relation type) that exists for the respective target object instance.

Marking Rule (MR22: Re-marking creation contexts):
Let dm = (name, OTypeSet, RelTypeSet) be a data model and ds = (dm, OSet, RelSet, M_{Cc}) be a corresponding data structure. Further, let cc ∈ CreationContexts be a creation context. Then:

When assigning an object instance (with identifier oid) to cc.SoidSet; i.e., cc.SoidSet = SoidSet ∪ {oid}, ⇒

$$M_{Cc}(\text{cc}) := \begin{cases} \text{ACTIVATED}, & |\text{cc.SoidSet}| < \text{cc.relType.min} \\ \text{BLOCKED}_{\text{card}}, & |\text{cc.SoidSet}| = \text{cc.relType.max} \\ \text{CONFIRMED}, & \text{else} \end{cases}$$

As long as the minimum cardinality constraint of a relation is not met, the corresponding creation context is marked as ACTIVATED. In turn, this indicates that a corresponding creation activity must be mandatorily executed (cf. *Execution Rule ER7*a). This activity is invokable in the context of the respective higher-level object instance. Regarding the separate column in the overview table, that comprises the different work-items, opposed to optional activities which are integrated within a context menu, mandatory activities are directly placed in the right column (cf. Fig. 9.13). In addition, a mandatory activity for creating a lower-level object instance is considered in the process-oriented view generated for the object type of the higher-level object instance as well.

Execution Rule (ER7: Object creation and relation assignment):
Let dm = (name, OTypeSet, RelTypeSet) be a data model and ds = (dm, OSet, RelSet, M_{Cc}) be a corresponding data structure. Further, let relType $\in$ RelTypeSet be a relation type. Then:

a) $\forall$ cc $\in$ CreationContexts with M_{Cc}(cc) = ACTIVATED:
A new object instance o of type source and a new relation rel of type relType with rel.source = o $\wedge$ rel.target = cc.toid must be created.

b) $\forall$ cc $\in$ CreationContexts with M_{Cc}(cc) = CONFIRMED:
A new object instance o of type source and a new relation rel of type relType with rel.source = o $\wedge$ rel.target = cc.toid may be optionally created.

c) $\forall$ cc $\in$ CreationContexts with M_{Cc}(cc) = BLOCKED_{card}:
No new relation rel = (relType, sourceoid, targetoid) of type relType with rel.target = cc.toid may be created.

Example 9.13 (Considering a minimum cardinality constraint):
For each `application` object instance at least three `review` object instances must be created. For this purpose, a corresponding work item is dynamically added to the `application` work list of the responsible user. Consider therefore the overview table listing `application` object instances. Here, the item for starting the respective creating activity is then no longer displayed in the context menu of the respective `application` object instance but directly in the column containing the activities executable for this object instance (cf. Fig. 9.13).

In turn, taking the maximum cardinality constraint of a relation into account, the work item in the overview table will be dynamically disabled when reaching the maximum number of lower-level object instances (cf. *Execution Rule ER7c*). In turn, consider the case for creating lower-level object instances independent from a higher-level one. For this purpose, a combo box is displayed in the form for editing the lower-level object instance. In this case, the higher-level object instance for which the maximum number of lower-level object instances is reached is dynamically removed from the corresponding combo-box. In particular, only higher-level object instances are offered for selection which corresponding process context is either marked as ACTIVATED or CONFIRMED.

Example 9.14 (Considering maximum cardinality constraints):
For each `application` object instance, at most five `review` object instances may be created. If for a particular `application` object instance already five `review` object instances exist, the work item for creating corresponding `review` object instances becomes disabled in the `application` overview table of the respective user. Furthermore, when editing lower-level instances, the `application` object instance is no longer selectable in the respective combo-box.

If a creation context is marked as CONFIRMED according to *Execution Rule ER7b*, additional object instances referencing the respective higher-level one may be optionally created.

Fig. 9.16 additionally contains all rules needed for the creation of object instances.

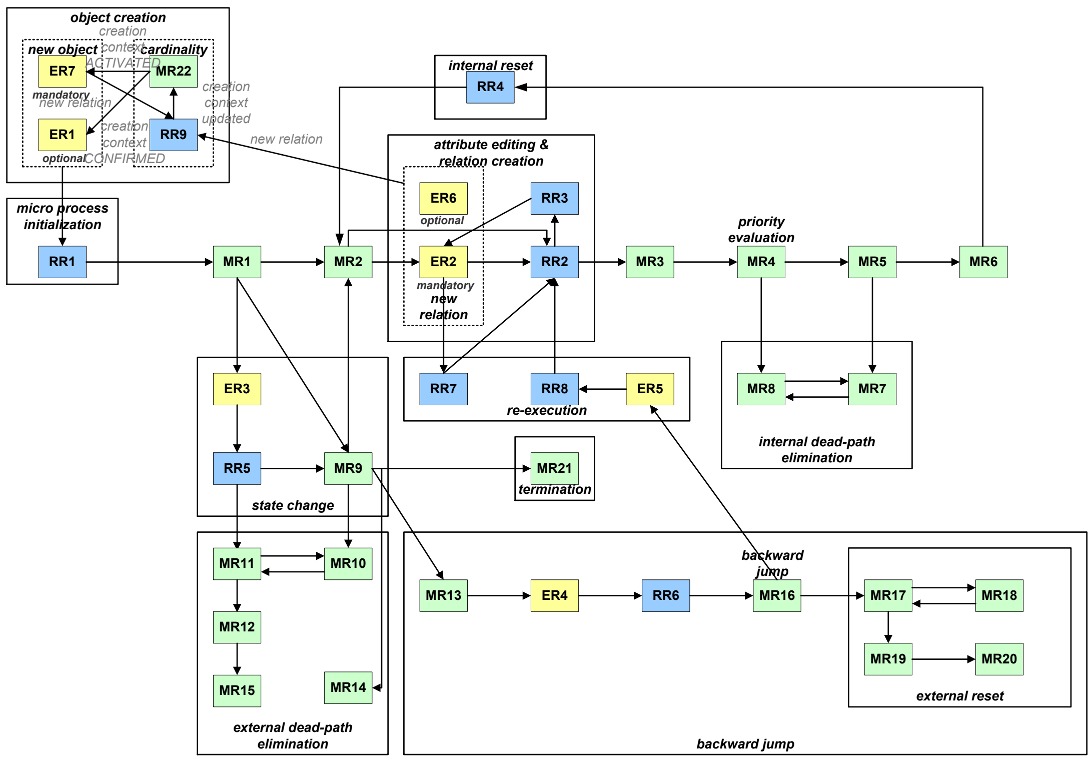
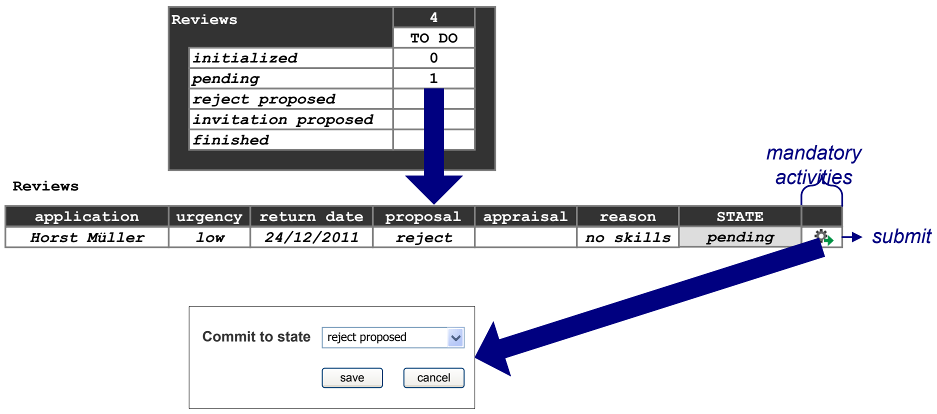


Figure 9.16: Rules for object creation

9.1.3 User Commitments and Decisions

As discussed in Sect. 7.3, external state transitions correspond to points during process execution at which a user must explicitly commit that the target state of the transition may be activated (assuming that required data is available). As a prerequisite, for all explicit micro transitions a corresponding transition responsibility (i.e., a user role) must be provided (cf. Sect. 7.4). When an external micro transition is reached during run-time (i.e., marked as CONFIRMABLE), the respective user commitment becomes mandatory to proceed with the process (cf. Sect. 7.4). Consequently, the activities for making respective commitments are mandatory as well. Like activities for writing attribute values, they are considered in the aggregated view; i.e., if for a micro process instance a user commitment is required, the corresponding counter will be incremented (cf. Fig. 9.17). Regarding the data-oriented view, which lists the invoked instances, the activity is displayed as mandatory one, directly in the table and will be excluded from the popup-menu. When invoking this activity, the required user dialog is automatically generated (cf. Fig. 9.17). If there are several explicit outgoing micro transitions, the responsible user must select the desired one. This can be realized, for example, using a combo box that lists all possible subsequent states (cf. Fig. 9.17).



Reviews	4
	TO DO
initialized	0
pending	1
reject proposed	
invitation proposed	
finished	

Reviews

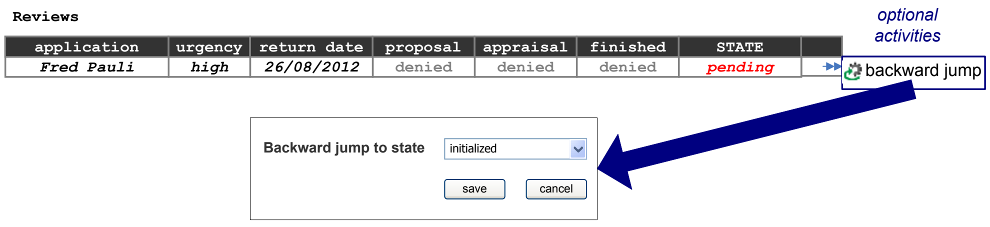
application	urgency	return date	proposal	appraisal	reason	STATE
Horst Müller	low	24/12/2011	reject		no skills	pending

Commit to state: reject proposed

Figure 9.17: User Commitments and Decisions

9.1.4 Backward Jumps

Opposed to user commitments and user decisions, which are always mandatory when they become activated, *backward jumps* are optional. Note that respective jumps are usually applied to handle exceptional situations. In particular, activities enabling backward jumps are only listed in the overview tables when corresponding backward transitions are activated (i.e., marked as CONFIRMABLE). Further, the generated user dialog is similar to the user commitment we discussed in the previous section. Since backward jumps to different previous states may exist, the desired state is also selectable using a combo box (cf. Fig. 9.18).



Reviews

application	urgency	return date	proposal	appraisal	finished	STATE
Fred Pauli	high	26/08/2012	denied	denied	denied	pending

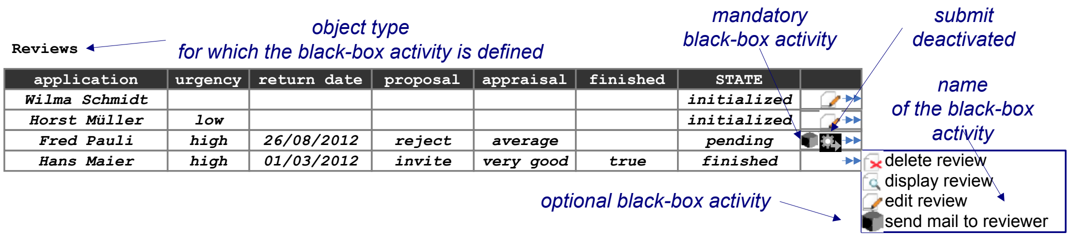
Backward jump to state: initialized

Figure 9.18: Backward jumps

9.2 Black-box Activities

In addition to *form-based activities*, PHILharmonicFlows allows for the integration of *black-box activities*, which may be assigned to different states. In particular, black-box activities realize more complex business functions for which a specific implementation is required. While form-based activities provide input fields (e.g., text-fields, combo-boxes, checkboxes, buttons, etc.) for writing and data fields for reading selected attributes of object instances, black-box activities enable computations as well as the integration of advanced functionalities (e.g., sending e-mails or invoking web services). As illustrated in Fig. 9.19, like form-based activities, black-box activities may be defined as *optional* or *mandatory*. In the latter case, their execution is mandatory in order to proceed with the control flow. In particular, the respective state may be only terminated if all mandatory black-box activities are executed. This way, PHILharmonicFlows allows differentiating between a *data- and activity-driven execution paradigm* at run-time. Black-box activities are automatically added to the overview tables and are considered when generating worklists (i.e., mandatory ones). Regarding the latter, the counters displayed in the context of an aggregated view are incremented for each object instance for which at least one mandatory activity must be executed (regardless of whether this is a form-based or black-box activity); i.e., object instances are counted and not activities. Particular activities are transparent when invoking corresponding overview tables listing the affected instances. Finally, for executing black-box activities automatically, a special user role `SYSTEM` is provided.

Example 9.15 (Integrating black-box activities):
Consider Fig. 9.19 which illustrates an overview table comprising four `review` object instances. For the `review` object instance displayed in the third row, state `pending` is currently activated. In this state, a mandatory black-box activity needs to be executed. For this purpose, a corresponding icon is directly displayed in the right column. Since this black-box activity is defined as mandatory, it is not possible to commit the transition to a subsequent state. Hence, the corresponding submit-icon is deactivated. In turn, the fourth `review` object instance is in state `finished`. Here, the black-box activity `send mail to reviewer` may be optionally executed and is therefore assigned in the context menu.



Reviews

application	urgency	return date	proposal	appraisal	finished	STATE	
Wilma Schmidt						*initialized*	
Horst Müller	*low*					*initialized*	
Fred Pauli	*high*	*26/08/2012*	*reject*	*average*		*pending*	
Hans Maier	*high*	*01/03/2012*	*invite*	*very good*	*true*	*finished*	

Figure 9.19: Integrating black-box activities

9.2.1 Defining a black-box activity

For each object type, an overview table is automatically generated. As illustrated in Fig. 9.20, in order to adequately integrate any black-box activity, it needs to be defined in respect to a particular object type, but may also process instances of related object types at run-time. Therefore, the identifier of the processed object instance is passed out to the activity as input. It can then be used to determine related object instances when implementing the business function of the black-box activity.

Example 9.16 (Processing related object instances derived by the OID):
Consider an activity comparing the provided `skills` of an `applicant` with the `skills` required by the offered `job`. This activity is defined in respect to an `application`. However, based on the OID of the `application`, information from the corresponding `skill` object instances are fetched.

Black-box activities are defined for a particular object type at build-time and are executable for each object instance of this type at run-time. Regarding the `review` object instances (cf. Fig.

9.19), for example, activity `send mail to reviewer` becomes executable for each `review` object instance when activating state `finished`.

Note that the required relationship between black-box activities and particular object types is not really a restriction. If several input parameters are required (e.g., OIDs of several object instances not related to each other), a user interface must be implemented that allow for required data inputs; i.e., black-box activities may also comprise user forms. However, these must be provided by the implementation of the black-box activity.

Example 9.17 (Processing related object instances requiring additional user input):
When comparing two `applications`, the activity making this comparison is assigned to the `application` object type. At run-time, it may then be invoked in respect to one `application` object instance. For selecting the other one, a corresponding user interface must be (manually) implemented.

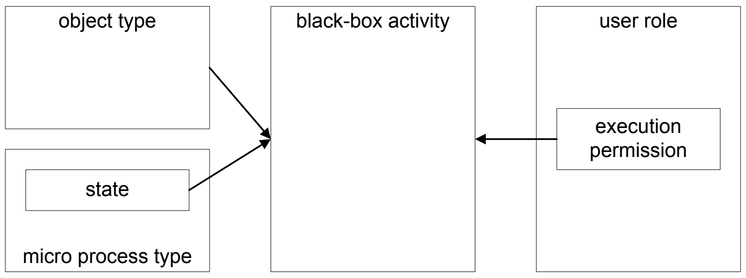


Figure 9.20: Relationship between black-box activities and object types

To foster the reuse of a black-box activity (e.g., to enable its use at different points during process execution), it must be encapsulated by an *activity template*. Such a template comprises the name of the activity, the object type the activity refers to, and the required service implementation[2] (cf. Def. 36). When generating overview tables, the name of the activity is used as label (cf. black-box activity `send mail to reviewer` in Fig. 9.19).

Definition 36 (Black-box activity template):
A **black-box activity template** is a tuple **actTempl = (name, oType, service)** where

- name $\in$ Identifiers is an identifier.
- oType $\in$ OTypes is the object type the activity refers to.
- service corresponds to the implementation of the business function encapsulated by the activity template.

ActTemplates denotes the set of all activity templates.

Whether or not a black-box activity may be invoked for a particular object instance may also depend on the currently activated state of this instance. Therefore, any black-box activity template is directly assigned to a particular state of the micro process type that corresponds to the respective object type (cf. Def. 37). Consequently, at a certain point in time a black-box activity may be executable for certain object instances, while this is not the case for others.

[2]How to implement respective services and connect them with the processed object instance is out of the scope of this thesis

Definition 37 (Black-box activity):
Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet, BackTransTypeSet) $\in$ MicProcTypes be a micro process type. Then:

A **black-box activity** is a tuple **act = (actTempl, stateType)** where

- actTempl $\in$ ActTemplates is an activity template.
- stateType $\in$ StateTypeSet with micProcType.oType = actTempl.oType
  is a state type of the micro process type that corresponds to the object type the activity template refers to.

ActivitySet$_{\textbf{micProcType}}$ corresponds to the finite set of black-box activities defined for micProcType. Further, **ActivitySet$_{\textbf{state}}$** denotes the set of all black-box activities assigned to a particular micro process state (of all micro process types).
In the following, a micro process type represents a tuple **micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet, BackTransType, ActivitySet)**; i.e., parameter ActivitiySet is assigned.

9.2.2 User Assignment and Authorization

Similar to execution responsibilities of form-based activities, permissions for executing black-box activities must be granted to selected user roles (cf. Def. 38).

Example 9.18 (Execution permissions):
Consider Fig. 9.19. It must be defined which user roles may execute activity `send mail to reviewer` when state `finished` becomes activated.

Definition 38 (Execution permission for black-box activities):
Let UserRoles be the set of all defined user roles. Then:

An **execution permission** is a tuple **execPerm = (act, role)** where

- act $\in$ ActivitySet$_{\text{state}}$ is a black-box activity.
- role $\in$ UserRoles is a user role.

ExecPermissions denotes the set of all definable execution permissions.

At a particular point in time, a black-box activity may be mandatory for a certain user, while being optional for another one. More precisely, a mandatory black-box activity must be executed by a particular user. Simultaneously, another user may execute the activity optionally. As discussed in Chapt. 8, PHILharmonicFlows additionally allows for data-driven process execution. Process execution may even proceed if required attribute values become available after executing a black-box activity (instead of manually editing any user form). In turn, process execution may proceed without executing a black-box activity if required attribute values are already available. Opposed to this, in certain situations, a more activity-driven execution behavior might be required when compared with a strictly data-driven one; i.e., it should be possible to enforce the execution of black-box activities independent from already available attribute values. Like for generic activities, this requires to differentiate between mandatory black-box activities and optional ones. For this purpose, execution permissions are definable as mandatory in PHILharmonicFlows (cf. Def. 39a).

Definition 39 (Properties of execution permissions):

a) **mandatory: ExecPermissions $\mapsto$ BOOLEAN** defines whether a particular black-box activity must be executed by a certain user role; i.e., the state to which the activity refers may only be left after executing the activity.

b) **repeatable: ExecPermissions $\mapsto$ BOOLEAN** defines whether the execution of a particular black-box activity may be repeated by authorized users; i.e., the responsible user role may execute the black-box activity several times as long as the respective state remains activated.

Assume that for a black-box activity, a corresponding mandatory execution permission is available. The state this activity is assigned to then may only be left after having executed the activity. To enforce a strict execution sequence for black-box activities, different states must be specified for them. In addition, a corresponding mandatory execution permissions must be specified. Finally, to allow for the re-execution of a black-box activity, the corresponding execution permission must be defined as repeatable (cf. Def. 39b).

When a state becomes activated, black-box activities are automatically assigned to the worklists of the users owning respective execution permissions (cf. Fig. 9.19). When generating the overview table listing the object instances for which mandatory black-box activities must be executed, the respective activity is not displayed in the popup menu containing the optional activities, but directly in the table. For mandatory black-box activities their label is displayed as tooltip.

9.2.3 Execution of Black-box Activities

Since a micro process execution now may also depend on the execution of black-box activities, the operational semantics introduced in Chapt. 8 must be extended. Like for the other components, a number of run-time markings is required for black-box activities. Based on these markings, it can be expressed when respective activities are executable, running, or not executable any longer. More precisely, each black-box activity is either marked as WAITING, READY, ACTIVATED, UNCONFIRMED, CONFIRMED, or SKIPPED (cf. Def. 40).

Definition 40 (Markings for black-box activities):
Let micProcInstance be a micro process instance of type micProc; i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

M_{Act}: ActivitySet $\mapsto$ ActivityMarkings assigns to an activity its current marking M_{Act}(activity) $\in$ ActivityMarkings. Thereby, **ActivityMarkings = {WAITING, READY, ACTIVATED, UNCONFIRMED, CONFIRMED, SKIPPED}**. The semantics of these markings is described in Tab. 9.2.

In the following, a micro process instance represents a tuple **micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data}, M_{Act})**.

These markings have to following meanings:

Marking	Label	Description
○	WAITING	It is not possible to execute the activity since the state it is assigned to is not activated. Note that the activity may be executed later on when its state becomes activated (i.e., a previous state is currently activated).
●	READY	The black-box activity is assigned to the currently activated state. Hence it may be executed.
▶	ACTIVATED	After invoking the black-box activity, it is running (i.e., its state is marked as ACTIVATED).
□	UNCONFIRMED	The execution of the black-box activity is completed and the state the activity belongs to is still activated.
■	CONFIRMED	The execution of the black-box activity is completed and the state the activity belongs to has been left (i.e., marked as CONFIRMED).
☒	SKIPPED	The black-box activity has not been executed and it will not be executed anymore (unless a backward jump will be triggered). Either the state the black-box activity refers to was not activated or left without having executed the black-box activity (i.e., the latter is optional).

Table 9.2: Activity markings

Fig. 9.21 illustrates the markings and their possible transitions:

1. A black-box activity changes its marking from WAITING to READY when the state the activity belongs to becomes marked as ACTIVATED.

2. A black-box activity changes its marking from READY to ACTIVATED when being invoked (either by a user or automatically).

3. A black-box activity changes its marking from ACTIVATED to UNCONFIRMED when its execution terminates.

4. An already executed black-box activity changes its marking from UNCONFIRMED to CONFIRMED when a subsequent state becomes activated; i.e., the state the black-box activity belongs to becomes marked as CONFIRMED.

5. A not yet executed black-box activity changes its marking from READY to SKIPPED when a subsequent state becomes activated; i.e., the state the black-box activity belongs to becomes marked as CONFIRMED.

6. A black-box activity changes its marking from WAITING to SKIPPED when the state the activity belongs to becomes marked as SKIPPED.

7. An already executed black-box activity changes its marking from UNCONFIRMED to ACTIVATED when its execution is repeated.

When a micro process instance becomes initialized, black-box activities must be initialized as well. Consequently, all black-box activities assigned to the start state of the micro process instance are initially marked as READY. In turn, all other black-box activities are initially marked as WAITING (cf. Reaction Rule RR1"').

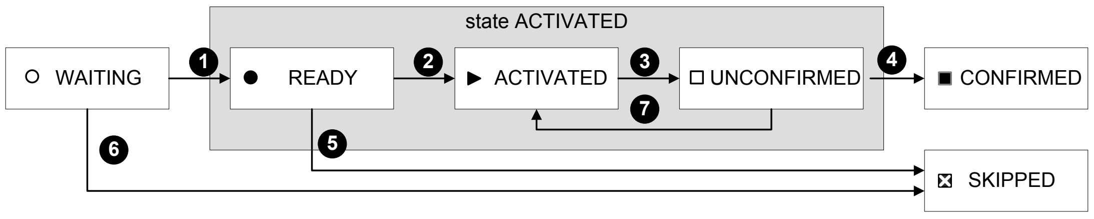


Figure 9.21: Black-box activity markings and their transitions

Reaction Rule (RR1”’: Initializing black-box activities):
Let dm = (name, OTypeSet, RelTypeSet) $\in$ DM be a data model and ds = (dm, OSet, RelSet, M_{Cc}) be a corresponding data structure. Further, let micProcInstance be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet, ActivitySet); i.e., micProcInstance $\in$ micprocinstances(micProc). Finally, let startState = (name, sMicStepSet) $\in$ StateSet be the start state of micProcInstance and startMicStep $\in$ MicStepSet the start micro step.

When creating a new object instance: OSet = OSet $\cup$ {o} with o.oid = micProcInstance.oid, the initial marking of a black-box activity is set as follows:

a)-j) see RR1 in Sect. 8.3

k) see RR1’ in Sect. 8.6

l) see RR1” in Sect. 9.1

m) $\forall$ act $\in$ $\text{ActivitySet}_{\text{startState}}$: M_{Act}(act) := READY;
i.e., all black-box activities assigned to the start state are marked as READY.

n) $\forall$ act $\in$ $\text{ActivitySet}_{\text{micProcType}}$ - $\text{ActivitySet}_{\text{startState}}$: M_{Act}(act) := WAITING;
i.e., all black-box activities not belonging to the start state are marked as WAITING.

Black-box activities currently marked as READY may be executed by any authorized user. This is expressed by *Execution Rule ER8*. Accordingly, these activities are added to the overview tables. Furthermore, mandatory black-box activities are considered in generated worklists.

Execution Rule (ER8: Executing black-box activities):
Let micProcInstance = (micProc, oid, M_{State}, M_{MicStep}, M_{MicTrans}, $M_{\text{BackTrans}}$, M_{Data}, M_{Act}) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet, ActivitySet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

a) $\forall$ act $\in$ $\text{ActivitySet}_{\text{State}}$ with M_{Act}(act) = READY $\wedge$ mandatory(act) = TRUE:
The black-box activity must be executed when it becomes marked as READY.

b) $\forall$ act $\in$ $\text{ActivitySet}_{\text{State}}$ with M_{Act}(act) = READY $\wedge$ mandatory(act) = FALSE:
The black-box activity may be optionally executed when it becomes marked as READY.

c) $\forall$ act $\in$ $\text{ActivitySet}_{\text{State}}$ with M_{Act}(act) = UNCONFIRMED $\wedge$ repeatable(act) = TRUE:
The execution of the black-box activity may be repeated when the activity becomes marked as UNCONFIRMED.

To indicate whether a black-box activity was executed, we introduce function *executed*:

Definition 41 (Executed activities):
Let micProcInstance be a micro process instance of type micProc; i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

executed: ActivitySet $\mapsto$ BOOLEAN expresses whether or not the black-box activity has been executed yet.

When terminating the execution of a black-box activity, its marking changes from ACTIVATED to UNCONFIRMED (cf. *Reaction Rule RR10*). Only repeatable black-box activities may then be re-executed. The latter is possible as long as the state they are assigned to is marked as ACTIVATED (cf. Execution Rule ER8C).

Reaction Rule (RR10: Black-box activity execution):
Let micProcInstance be a micro process instance of type micProc; i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

$\forall$ act $\in$ ActivitySet with M_{Act}(act) = ACTIVATED $\wedge$ executed(act) = TRUE: M_{Act}(act) := UNCONFIRMED;

i.e., a black-box activity changes its marking from ACTIVATED to UNCONFIRMED if the execution terminates.

Black-box activities must be taken into account when activating subsequent states. In such a case, all executed black-box activities currently marked as UNCONFIRMED of the currently activated state are re-marked as CONFIRMED (cf. Marking Rule MR9'g). In turn, all non-executed black-box activities currently marked as READY of this state are re-marked as SKIPPED (cf. Marking Rule MR9'h). Finally, all black-box activities of the subsequent state become marked as READY; i.e., they may then be executed (cf. Marking Rule MR9'i).

Marking Rule (MR9': State change considering black-box activities):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data}, M_{Act}) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet, ActivitySet); i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

micTrans=(source, target, priority) $\in$ MicTransSet with $M_{MicTrans}$(micTrans) = READY $\wedge$ isexternal(micTrans) = TRUE:

a-f) see MR9 in Sect. 8.5

g) $\forall$ act $\in$ $\text{Activities}_{source}$ with M_{Act}(act) = UNCONFIRMED: M_{Act}(act) := CONFIRMED;
i.e., all black-box activities of the newly CONFIRMED state which are currently marked as UNCONFIRMED, are re-marked as CONFIRMED.

h) $\forall$ act $\in$ $\text{Activities}_{source}$ with M_{Act}(act) = READY: M_{Act}(act) := SKIPPED;
i.e., all black-box activities of the newly CONFIRMED state which are currently marked as READY, are re-marked as SKIPPED.

i) $\forall$ act $\in$ $\text{Activities}_{target}$ with M_{Act}(act) = WAITING: M_{Act}(act) := READY;
i.e., all black-box activities of the newly ACTIVATED state which are currently marked as WAITING, are re-marked as READY.

Note that it is not possible to proceed with the execution of a micro process instance (i.e., to leave the currently activated state) if there exist mandatory activities of this state that have not been executed yet. Consequently, an activity for committing a state change is not executable as long as not all mandatory activities will be executed. Consider as example the deactivated icon as illustrated in Fig. 9.19. In order to block external micro transitions in such a case, we introduce an additional marking BLOCKED$_{act}$ for micro transitions (cf. Def. 42).

Definition 42 (Micro transition marking considering black-box activities):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data}, M_{Act}) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet, ActivitySet); i.e., micProcInstance $\in$ micprocinstances(micProc). Thereby, $M_{MicTrans}$: MicTransSet $\mapsto$ MicroTransitionMarkings assigns to each micTrans $\in$ MicTransSet its current marking $M_{MicTrans}$(micTrans) with MicroTransitionMarkings = {WAITING, CONFIRMABLE, READY, ENABLED, ACTIVATED, UNCONFIRMED, CONFIRMED, BYPASSED, SKIPPED}. Then we extend the set of possible markings for micro transitions as follows:

MicroTransitionMarkings := MicroTransitionMarkings $\cup$ {BLOCKED$_{act}$}.
Marking BLOCKED$_{act}$ indicates that it is not possible to activate the micro transition since not all mandatory activities of the source state of this micro transition have been executed yet.

As shown in Sect. 8.5, a state transition will be enabled when a corresponding external micro transition becomes marked as READY. In turn, this marking triggers the activation of the subsequent state. For explicit micro transitions requiring a user commitment, marking CONFIRMABLE is assigned before. To consider black-box activities in this context, external micro transitions may only be marked as READY or CONFIRMABLE if all mandatory black-box activities of the currently activated state are marked as UNCONFIRMED (cf. Marking Rule MR1”’).

Marking Rule (MR1”’: Marking external micro transitions as BLOCKED$_{act}$):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data}, M_{Act}) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet, ActivitySet); i.e., micProcInstance $\in$ micprocinstances(micProc). Further, let state = (name, sMicStepSet) $\in$ StateSet be a state. Then:

a) see MR1 in Sect. 8.4

b) see MR1’ in Sect. 8.4

c) see MR1” in Sect. 8.5

d) $\forall$ micStep=(ref, ValueSteps) $\in$ sMicStepSet with $M_{MicStep}$(micStep) = UNCONFIRMED:
   $\forall$ micTrans $\in$ outtrans(micStep) with isexternal(micTrans) = TRUE:

$$M_{MicTrans}(\text{micTrans}) := \begin{cases} \textbf{CONFIRMABLE}, \text{ if explicit(micTrans) = TRUE} \wedge \\ \qquad \forall \text{ act} \in \text{Activities}_{state}: M_{Act}(\text{act}) = \text{UNCONFIRMED} \\ \textbf{READY}, \text{ if explicit(micTrans) = FALSE} \wedge \\ \qquad \forall \text{ act} \in \text{Activities}_{state}: M_{Act}(\text{act}) = \text{UNCONFIRMED} \\ \textbf{BLOCKED}_{\textbf{act}}, \text{ else} \end{cases}$$

   i.e., as long as not all black-box activities of a state have been executed, outgoing external micro transitions are marked as BLOCKED$_{act}$.

e) $\forall$ valueStep $\in$ ValueSteps with $M_{MicStep}$(valueStep) = UNCONFIRMED:
   $\forall$ micTrans $\in$ outtrans(valueStep) with explicit(micTrans) = TRUE $\wedge$ isexternal(micTrans) = TRUE:
      $M_{MicTrans}$(micTrans) as defined in d)
   i.e., as long as not all black-box activities of a state have been executed, all outgoing external micro transitions are marked as BLOCKED$_{act}$.

After having executed all mandatory black-box activities, the blocking is undone; i.e., respective micro transitions are either marked as READY or CONFIRMABLE (depending on whether they are implicit or explicit ones). For this purpose, we introduce *Marking Rule MR23*:

Marking Rule (MR23: Unblocking external micro transitions):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data}, M_{Act}) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet, ActivitySet); i.e., micProcInstance $\in$ micprocinstances(micProc). Further, let state = (name, sMicStepSet) $\in$ StateSet be a state. Then:

$\forall$ micTrans = (source, target, priority) $\in$ MicTransSet with source $\in$ sMicStepSet $\wedge$ $M_{MicTrans}$(micTrans) = BLOCKED_{act}:
$\forall$ act $\in$ $\text{Activities}_{state}$: M_{Act}(act) = UNCONFIRMED:

$$M_{MicTrans}(\text{micTrans}) := \begin{cases} \textbf{CONFIRMABLE}, & \text{if explicit(micTrans) = TRUE} \\ \textbf{READY}, & \text{if explicit(micTrans) = FALSE} \end{cases}$$

i.e., if the execution of all mandatory black-box activities is completed, the state transition is enabled by marking implicit micro transitions as READY and explicit ones as CONFIRMABLE.

Fig. 9.22 additionally contains Execution Rule ER8, Reaction Rule RR10, and Marking Rule MR23 as required for the execution of black-box activities.

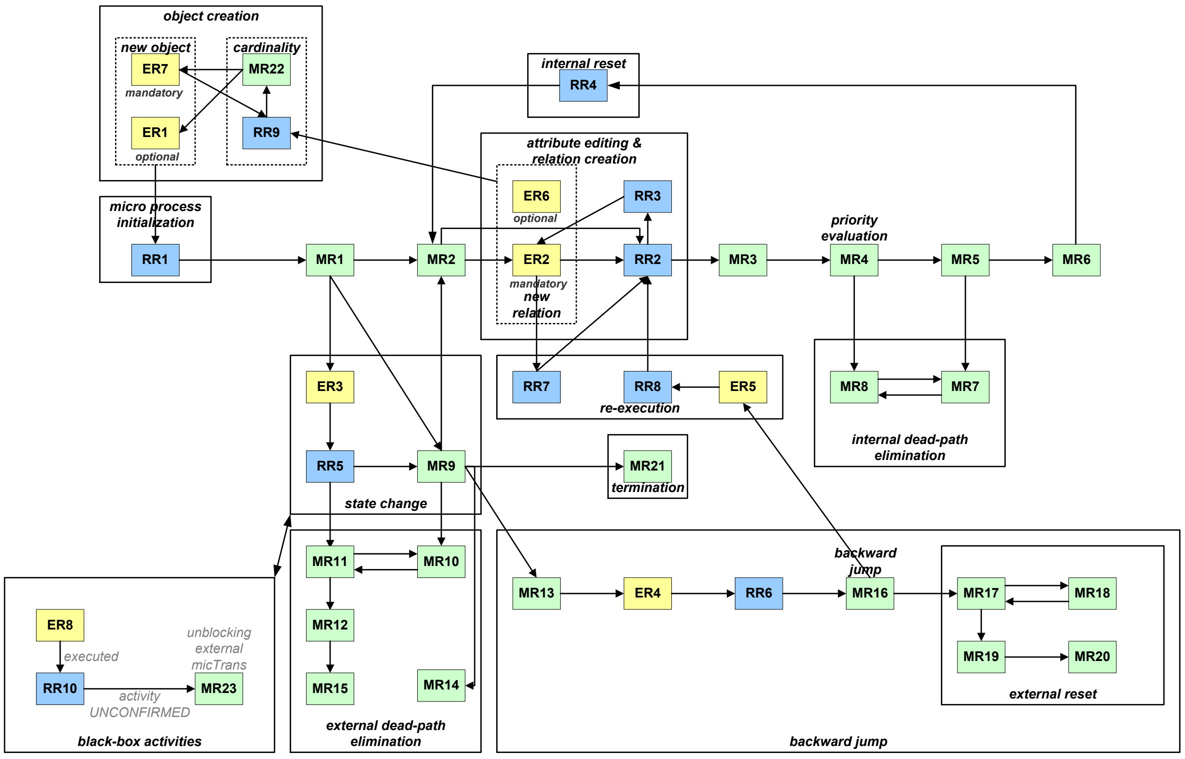


Figure 9.22: Rules for black-box activities: ER8, RR10, and MR23

When performing an external dead-path elimination (cf. Sect. 8.5), black-box activities must be marked as SKIPPED when the state they belong to becomes marked as SKIPPED as well. This is expressed by *Marking Rule MR24* (cf. Fig. 9.23).

Marking Rule (MR24: Marking black-box activities as SKIPPED):
Let micProcInstance = (micProc, oid, M_{State}, $M_{MicStep}$, $M_{MicTrans}$, $M_{BackTrans}$, M_{Data}, M_{Act}) be a micro process instance of type micProc = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet, ActivitySet); i.e., micProcInstance $\in$ micprocinstances(micProc). Further, let state $\in$ StateSet be a state. Then:

$\forall$ act = (actTempl, state) $\in$ $\text{Activities}_{state}$ with M_{State}(state) = SKIPPED: M_{act}(act) := SKIPPED;

i.e., when a state becomes marked as SKIPPED, corresponding black-box activities are marked as SKIPPED as well.

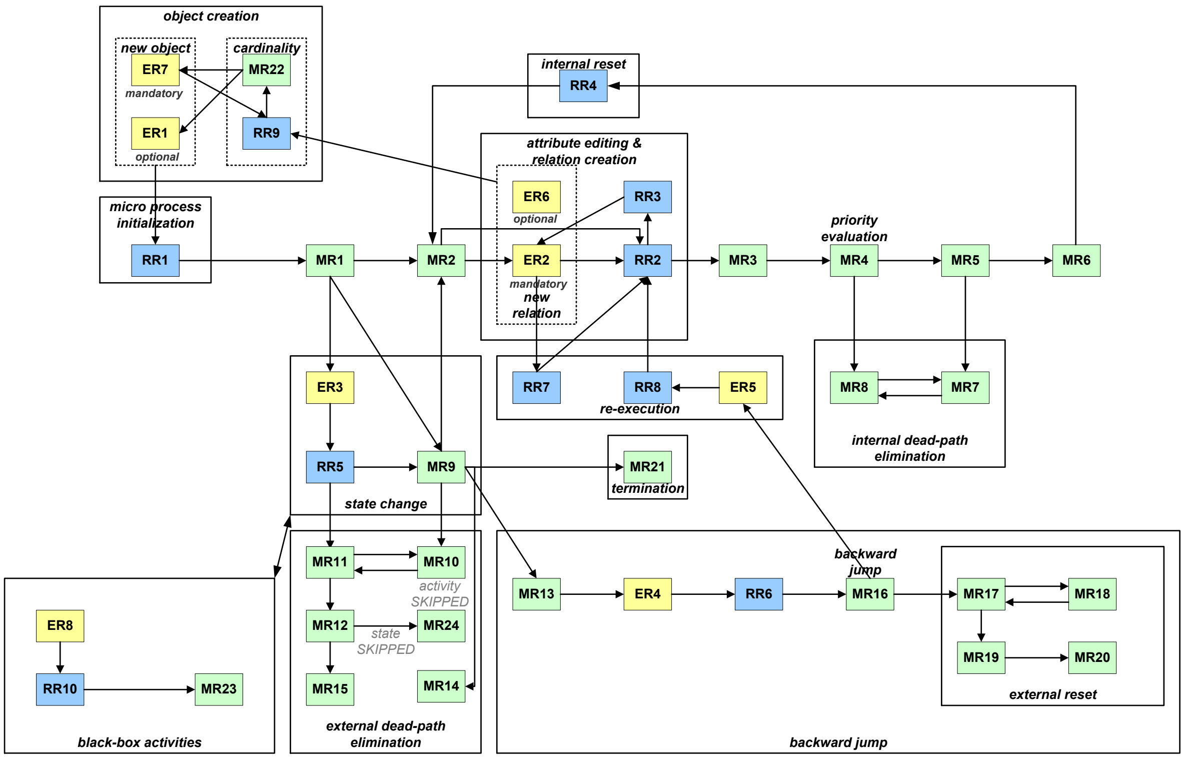


Figure 9.23: Rules for black-box activities during the external dead-path elimination

9.3 Batch Execution

As discussed, both form-based and black-box activities are executed in respect to a particular object instance. In addition, when executing form-based activities, object instances of related object types may be accessed simultaneously. To support this, actions corresponding to different object instances may be combined in one form (cf. Fig. 9.24).

Batch execution allows applying the same activity to a collection of object instances having the same object type in one go. Such aggregated execution is possible for both generic activities (i.e., editing, creating, and deleting of object instances, user commitments, backward jumps) and black-box activities. A form-based activity comprises input fields for each attribute type of the corresponding object type. Edited attribute values are then assigned to all invoked object instances; i.e., there exists one input field per attribute for all object instances (cf. Fig. 9.25).

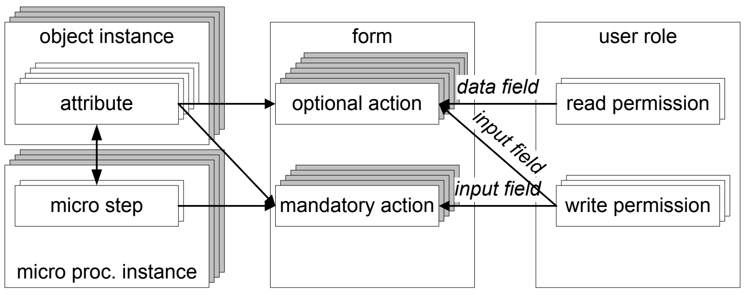


Figure 9.24: Context-sensitive activity

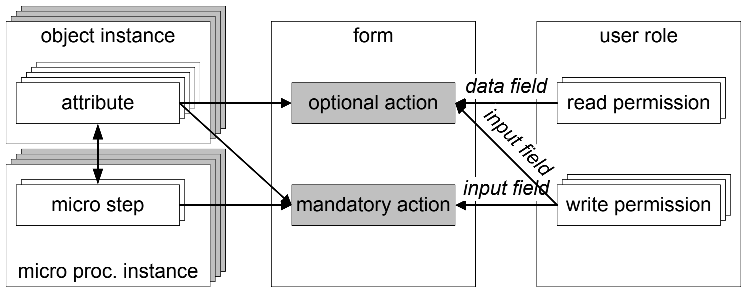


Figure 9.25: Batch execution of user forms

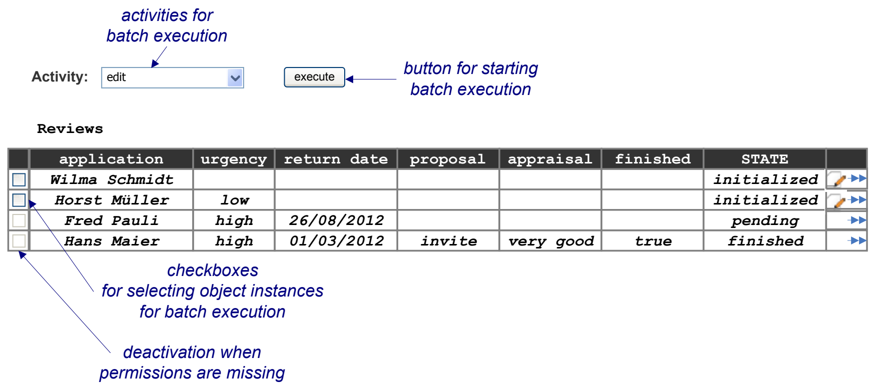
As a particular challenge it must be ensured that any user triggering such a batch execution owns the required permissions for all processed object instances. This necessitates an appropriate selection mechanism for object instances, which considers the currently activated object states as well. Regarding a batch execution based on form-based activities, the provided attribute values are assigned to all selected object instances. Therefore, a form should only comprise input fields for those attributes the user wants to change. All other ones must retain their individual values.

To ensure this, PHILharmonicFlows uses the following procedure for the batch execution of activities:

1. selection of the desired activity by the user,
2. selection of the desired attributes by the user (in case of form-based activities),
3. automatic de-selection of object instances (with missing permissions) by the system,
4. selection of the desired object instances by the user, and
5. execution of the activity by the user.

First, the activity for which batch execution shall be applied must be selected by the user. For this purpose, as illustrated in Fig. 9.26, a combo box listing all possible activities (defined in the context of the respective object type) is displayed at the top of the overview table. The left column of this table provides a checkbox for selecting desired object instances (cf. Fig. 9.26). If the user invoking the selected activity does not own corresponding permissions for the currently activated state of an object instance, the corresponding checkbox is automatically disabled;

e.g., consider the deactivated checkboxes in Fig. 9.26. This way, PHILharmonicFlows ensures that batch execution is only activated for object instances for which the respective user owns the permissions required. Moreover, when selecting the activity for editing object instances, a separate user dialog is invoked enabling the selection of particular attributes. If all information is available (i.e., desired action, desired object instances and optionally the desired attributes), the corresponding execution can be started by using the execute-button (as displayed besides the combo box for selecting the activity).


activities for batch execution
Activity:
edit
execute
button for starting batch execution
checkboxes for selecting object instances for batch execution
deactivation when permissions are missing

Reviews

	application	urgency	return date	proposal	appraisal	finished	STATE	
☐	*Wilma Schmidt*						*initialized*	
☐	*Horst Müller*	*low*					*initialized*	
☐	*Fred Pauli*	*high*	*26/08/2012*				*pending*	
☐	*Hans Maier*	*high*	*01/03/2012*	*invite*	*very good*	*true*	*finished*	

Figure 9.26: Enabling batch execution

9.4 Further Issues

In addition to the creation and processing of object instances, their deletion constitutes another relevant use case for application systems. In this context, like for relational databases, *referential integrity reference* has to be taken into account; i.e., a relation must not target at an object instance that does not exist anymore. Regarding object-aware process support, in addition, corresponding macro process instances need to be handled; i.e., coordination componentes must be updated after deleting a particular micro process instance. Both requires comprehensive concepts.

For deleting object instances, generic activities should be provided as well. Further, corresponding *delete permissions* need to be assigned using authorization tables. Opposed to *create permissions*, however, delete permissions (cf. Def. 43) depend on the currently activated state of the respective micro process instance.

Definition 43 (Delete permissions):
Let dm = (name, OTypeSet, RelTypeSet) be a data model and oType = (name, AttrTypeSet) $\in$ OTypeSet an object type with related micro process type micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet, BackTransTypeSet). Further, let UserRoles be the set of all defined user roles. Then:

A **delete permission delPerm = (oType, stateType, role)** is a tuple where

- oType $\in$ OTypeSet is an object type.
- stateType $\in$ StateTypeSet is a state type.
- role $\in$ UserRoles is a user role.

DeletePermissions corresponds to the set of all definable delete permissions.

If a user owns the permission to delete an object instance in a particular state, a corresponding activity for deleting this object instance will be displayed.

Example 9.19 (Delete permissions):
Consider Fig. 9.14. The `personnel officer` owns the delete permission for `review` object instances in state `initialized`. Hence, a corresponding optional activity is added to the respective overview table (cf. Fig. 9.19).

Actually, PHILharmonicFlows does not physically delete object instances. Instead, corresponding micro process instances are marked as SKIPPED indicating that their execution is no longer necessary. Otherwise, it would not be possible to trace the processing of the deleted object instance later on. When a micro process instance is marked as SKIPPED, the current processing state is frozen (i.e., the markings of states, micro steps, micro transitions, and backward transitions retain their marking) and all attribute values are further available. Opposed to this, however, relations to other object instances (and micro process instances respectively) need to be adequately handled. In this context, different options exist. Either all relations to other instances are deleted[3] or all instances which reference the object instance deleted are deleted as well[4]. Which approach shall be used depends on the respective use case; i.e., object-aware processes are driven by user decisions. For this reason, when deleting an object instance responsible users must be aware of affected lower-level object instances and decide about the concrete procedure. Both cases imply further adaptations of the resulting process and data structure (cf. Chapt. 12). When deleting dependent object instances, additional references to other higher-level object instances need to be considered. In turn, when only deleting the relations to other object instances, certain parts of the process structure become decoupled. As example consider `applications` not referring to any `job offer`. These `applications` are not assignable to any macro process instance any longer. Thus, for adequately handling the deletion of object instances additional features must be developed.

9.5 Summary

PHILharmonicFlows differentiates between generic and black-box activities. Generic activities comprise form-based activities for creating, editing, reading, and deleting object instances. Based on the defined operational semantics, these activities can be automatically generated at run-time. Opposed to this, a black-box activity requires an implementation or encapsulates any legacy application. Application data is then managed by the respective software component; i.e., outside the control of PHILharmonicFlows. Finally, PHILharmonicFlows allows for batch execution; i.e., to process multiple object instances in one go.

[3]Note that this is similar to "on delete set null" when using SQL[DD97]
[4]Note that this is similar to "on delete cascade" when using SQL[DD97]

10 Macro Process Modeling

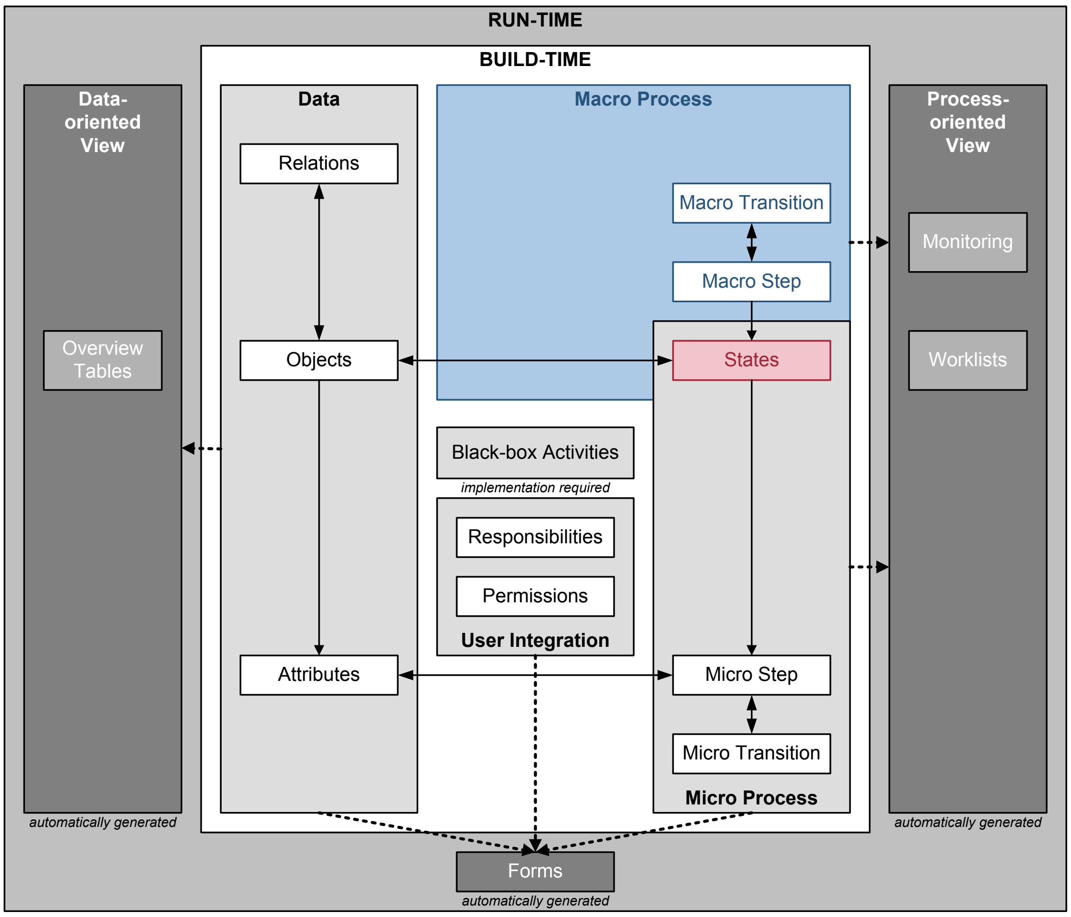


Figure 10.1: Macro process modeling in PHILharmonicFlows

Generally, a business process involves multiple object instances with same and different type. Moreover, object instances are inter-related and each of them is coupled with a corresponding micro process instance. Consequently, a *complex process structure* comprising multiple object instances and their interactions emerges at run-time. PHILharmonicFlows allows modeling such multi-object processes in terms of *macro processes*. As opposed to a *micro process*, which defines the *behavior* of a particular object type, a *macro process* covers *object interactions*; i.e., the execution of micro process instances needs to be coordinated taking the given data structure into account. More precisely, a macro process refers to parts of the data structure and consists of *macro steps* and *macro transitions* linking them (cf. Fig. 10.1). Opposed to a micro step that refers to a single attribute of a particular object type, a macro step refers to an object type by its own. Taking the semantical relationships between object types into account, various *coordination components* are required for coordinating the different kinds of interactions among the micro process instances of a process structure(i.e., coordination components *process contexts*, *aggregations*, and *transverse* as illustrated in Fig. 10.1). As will be shown in the following, macro processes allow hiding the complexity of large process structures from users.

10.1 State-based View

As described, PHILharmonicFlows allows modeling object behavior in terms of states and state transitions. Opposed to existing approaches (e.g., [BHS09, MRH07]), however, our framework enables a mapping between the values of object attributes and object states, and hence ensures compliance between them. In this context, data authorizations (i.e., permissions) are granted at the level of individual object attributes (and relations). In turn, process authorization (i.e., user assignment) is based on object states. As will be shown, states may be also used to define process synchronization constraints (i.e., object interactions). For each defined micro process, PHILharmonicFlows automatically generates *state-based views*. As illustrated in Fig. 10.2, a state-based view of a micro process instance abstracts from particular micro steps; i.e., only the states of the micro process instance are displayed. Accordingly, external micro transitions are logically mapped to the states their source and target micro steps belong to; i.e., the source (target) of a micro transition is displayed in the state-based view according to the state the source (target) micro step belongs to.

Example 10.1 (State-based view of the review micro process type):
Consider Fig. 10.2. The state-based view of the `review` micro process type comprises states `initialized`, `pending`, `reject proposed`, `invitation proposed`, and `finished`. These states are inter-connected through external micro transition types.

For state-based views, PHILharmonicFlows defines function *precedingStateTypes*. For a particular state `s`, it determines all predecessors (including `s` itself cf. Def. 44). In the following, this function is used for structural analyses of both micro and macro process types.

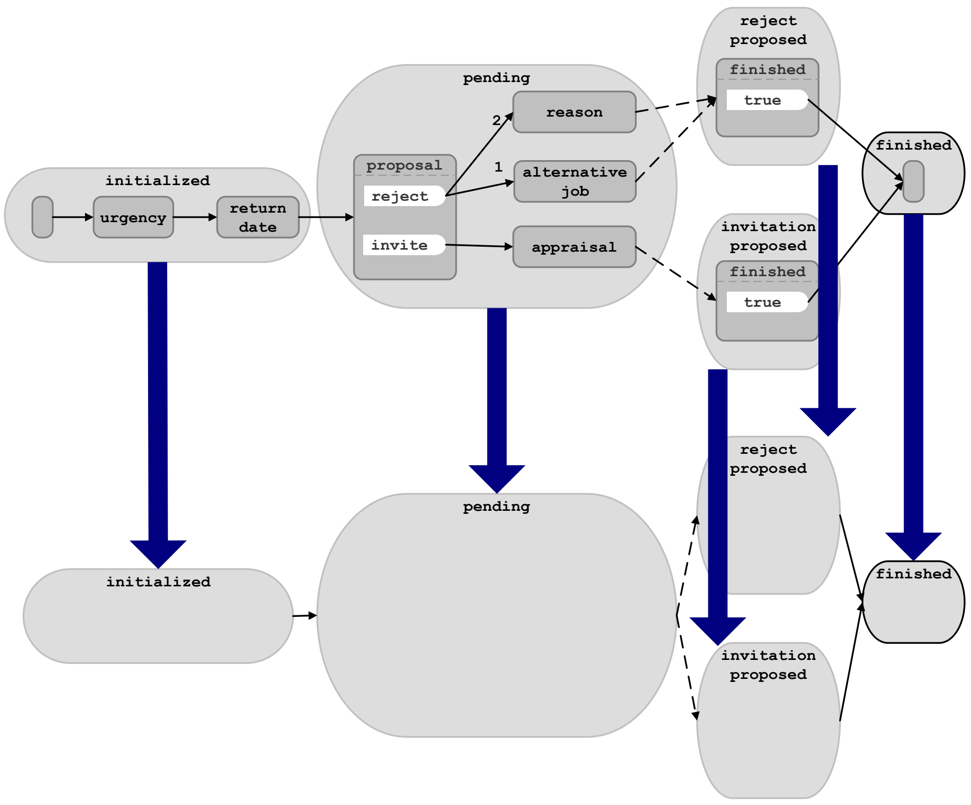


Figure 10.2: Generating the state-based view of the review micro process type

Definition 44 (Predecessor state types):
Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet, ActivitySet) ∈ MicProcTypes be a micro process type. Then:

For each state type s, **precedingStateTypes: StateTypeSet** $\mapsto$ **$2^{\text{StateTypeSet}}$** determines its directly or indirectly preceding state types (including s).

Example 10.2 (Predecessor state types):
Consider Fig. 10.2. Preceding state types of state `reject proposed` are the states `initialized`, `pending`, and state `reject proposed`.

10.2 Complex Process Structure

Object instances may be created by any user owning a corresponding create permission (cf. Def. 32). For each newly created object instance, a corresponding micro process instance is automatically initialized (cf. Marking Rule MR1) and started (cf. Reaction Rule RR1), resulting in the activation of the start state of the respective micro process instance. When creating an object instance, the respective user may assign attribute values and relations based on an automatically generated form (cf. Sect. 9.1). Which input and data fields are displayed in this context, depends on the read and write permissions granted for this user in the start state of the micro process instance.

As discussed, object instances are inter-related and coupled with corresponding micro process instances. Consequently, at run-time a *complex process structure* emerges comprising multiple object instances and their inter-dependencies (cf. Fig. 10.3).

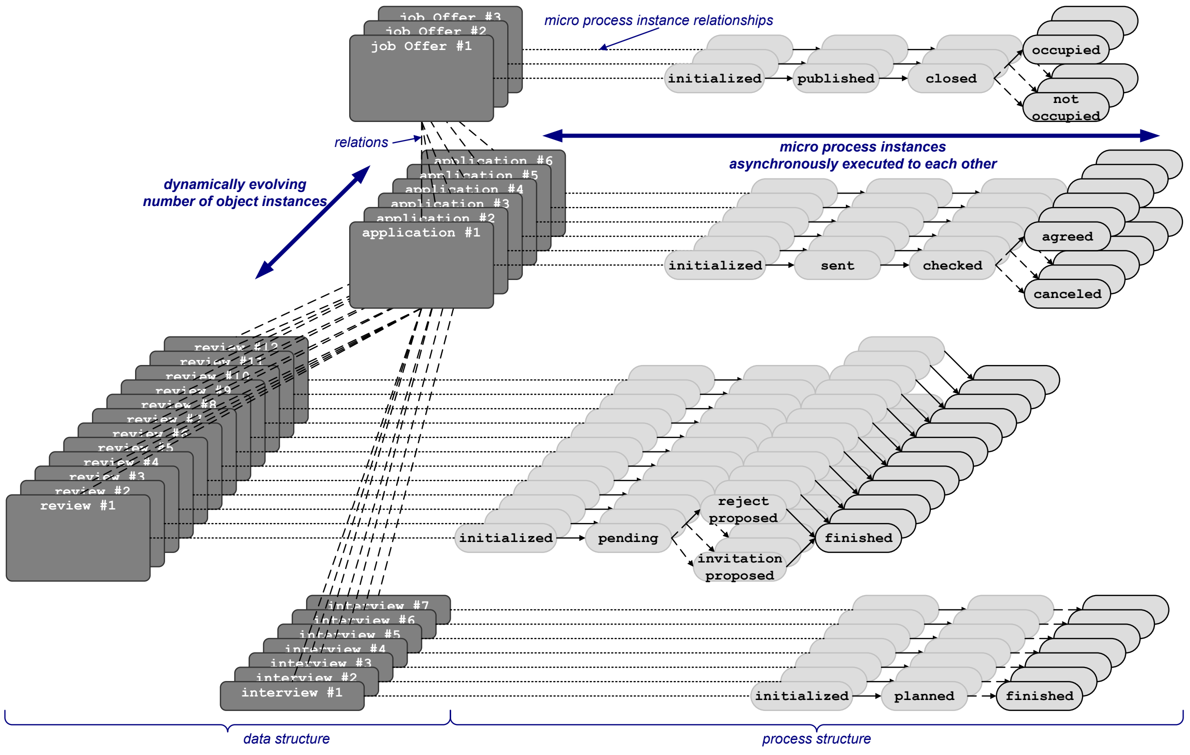


Figure 10.3: Complex process structure

Note that a process structure may comprise dozens or hundreds of micro process instances [MHHR06, MRH08a]. Thereby, a dynamically evolving number of object instances has to be supported. Although the corresponding micro process instances are executed asynchronously to each other, they need to be coordinated at certain points during their execution (cf. Prop. 5).

10.3 Macro Process Types

Despite the *asynchronous processing* of the micro process instances corresponding to a *complex process structure*, their execution must be synchronized at certain points in time; i.e., there exist *execution dependencies* between the micro process instances (cf. Fig. 10.4). In this context, the *potentially large number of micro process instances* that dynamically evolves during run-time constitutes a challenge. In particular, an appropriate abstraction for large process structures is required. On one hand, such abstraction must be comprehensible for end users, on the other it should serve as basis for synchronizing micro process execution.
When coordinating micro process instances, in addition, the semantic *relationships* that exist

between corresponding object instances must be taken into account; e.g., for each `application` object instance, several `review` object instances may exist. In particular, micro process instances may also have to be coordinated, even if no direct relation between the corresponding object instances exists; e.g., micro process instances related to `interview` object instances must be coordinated with the ones of `review` object instances. Generally, the *relationships* existing between object instances should be considered when synchronizing the execution of micro process instances. Thereby, the cardinalities of the semantic relationships as defined in the data model must be considered. As another challenge, a flexible coordination of micro process instances is required, while their execution should obey a *data-driven paradigm*; i.e., micro processes as well as macro processes should be defined and executed based on data; i.e., object types, object attributes, and object states.

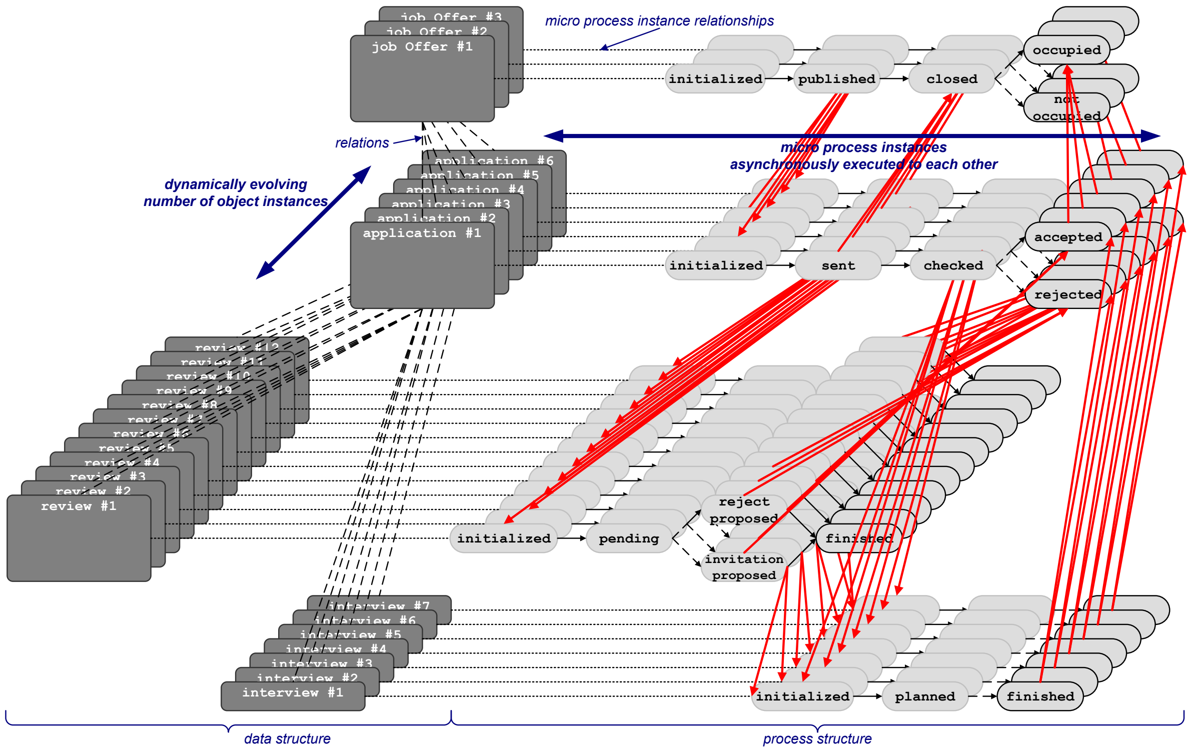


Figure 10.4: Execution dependencies in a complex process structure

To hide the complexity of large process structures from process designers and end-users, PHILharmonicFlows allows defining *macro process types* in a "flat" and compact way. For this purpose, at the type level, a macro process consists of a number of *macro step types* and *macro transition types* (cf. Fig. 10.5). As opposed to imperative process modeling, where process steps are defined in terms of activities, a particular macro step type `ot(s)` refers to an object type `ot` and a corresponding state type `s`. While micro process types are based on attribute types, the definition of a macro process type is based on object types. In this context, the state types of the micro process definitions play a crucial role. In particular, they do not only provide the basis for coordinating the execution of any micro process instance among different user roles, but also for synchronizing the execution of inter-related micro process instances. To express the dependencies existing between the states of micro process instances corresponding to different types, macro step types may be inter-connected using *macro transition types*.

Whether or not the state of the target macro step may be activated then depends on the state of the source macro step.

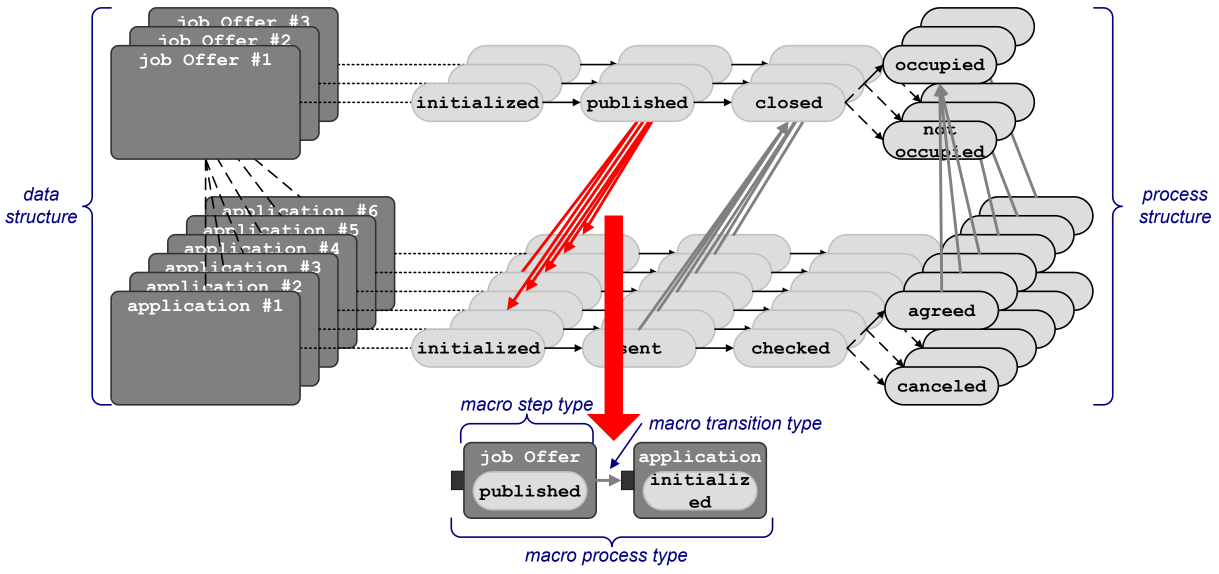


Figure 10.5: Mapping dependencies between macro step and macro transition types

Example 10.3 (Macro step and macro transition type):
Fig. 10.5 shows the macro step type referring to object type `job offer` in state `published`. It further depicts a macro transition type describing a dependency between a `job offer` in state `published` and corresponding `applications` in state `initialized`.

A macro process type may include both *parallel* and *alternative execution paths* (cf. Fig. 10.6). To express this, each macro step type comprises a number of *port types*. In turn, each macro transition type connects a macro step type with a port type of a subsequent macro step type. Generally, more than one macro transition type may refer to a particular port type. Whether the state of the macro step type can be activated during run-time then depends on the activation of its ports. More precisely, a state may only be activated if at least one of its ports belonging to a macro step referencing this state becomes activated, i.e., all incoming macro transitions of this port are fired. Based on this, *alternative execution* paths can be defined using multiple ports (i.e., *OR-semantics* is enabled). In turn, to enable *parallel execution* several macro transitions must target to the same port (i.e., *AND-semantics* is enabled).

Example 10.4 (AND semantics):
Consider the macro step type referring to object type `job offer` in state `not occupied` (cf. Fig. 10.6). At run-time, this macro step type may only be activated if the `job offer` object instance reaches state `closed` and for all `application` object instances, which refer to the respective `job offer` object instance, state `rejected` is activated.

Example 10.5 (OR semantics):
Consider the macro step type referring to object type `application` in state `rejected` (cf. Fig. 10.6). At run-time, it may only be activated if state `reject proposed` is activated for related all `review` or all related `interview` objects instances.

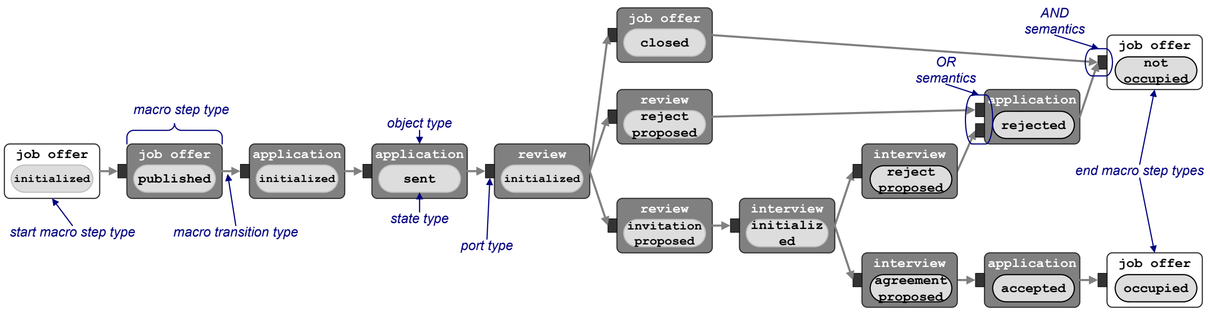


Figure 10.6: Recruitment macro process type

When executing a particular macro process instance at run-time, we must consider that not all micro process instances of the emerging process structure belong to one and the same *macro process instance*. For example, consider Fig. 10.7. All micro process instances related to the same macro process instance are coloured red. Consequently, a process structure may be mapped to several macro process instances. To handle this at run-rime, each macro process type refers to a *primary object type*. If an object instance of the primary object type is created at run-time, a corresponding macro process instance is initialized as well. This macro process instance then comprises all micro process instances which directly or indirectly reference the primary object instance. Finally, a particular object instance may be involved in several macro process instances (cf. Ex. 10.6).

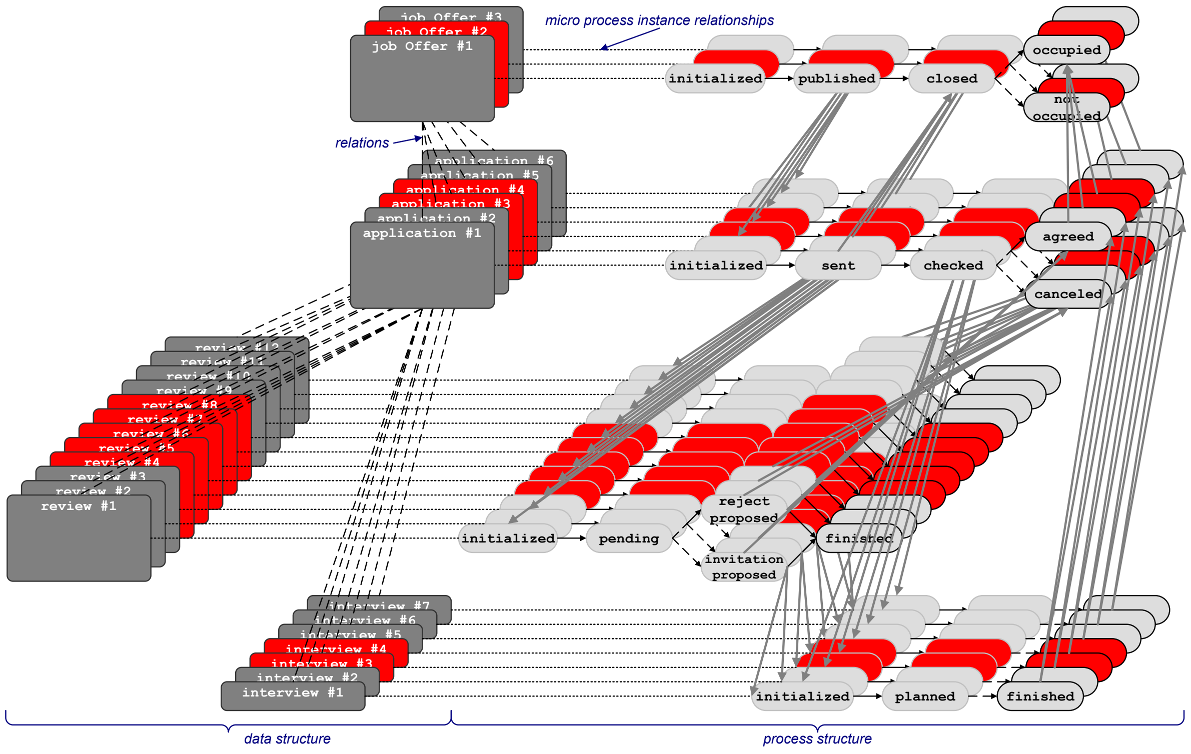


Figure 10.7: Recruitment macro process instance (coloured in red)

Example 10.6 (Macro process instances):
The macro process instance illustrated in Fig. 10.7 refers to a particular `job offer` object instance. For this purpose, the macro process instance comprises a `job offer` micro process instance, all `application` micro process instances referring to the latter , and all `review` as well as `interview` micro process instances referring to the respective `application` object instances. If another macro process type is defined including the `application` object type as primary object type, a new macro process instance is created for each of the two coloured `application` object instances. In this case, several macro process instances refer to the same micro process instances.

Definition 45 (Macro process types):
Let dm = (name, OTypeSet, RelTypeSet) be a data model. Then:

A **macro process type** is a tuple **macProcType = (name, oType, MacStepTypeSet, MacTransTypeSet)** where

- name $\in$ Identifiers is an identifier.
- oType = (name, AttrTypeSet) $\in$ OTypeSet is the primary object type.
- MacStepTypeSet is a finite set of macro step types with **macStepType = (oType, stateType, PortTypeSet)** $\in$ **MacStepTypeSet** having the following meaning:
    - oType $\in$ OTypeSet is an object type with related micro process type micProcType.
    - stateType $\in$ micProcType.StateTypeSet is a state type of the referenced object type.
    - PortTypeSet is a finite set of port types. Each port has a unique id in respect to all port types of macProcType.
- MacTransTypeSet $\subset$ MacStepTypeSet $\times$ MacStepTypeSet $\times$ PortTypeSet is a finite set of macro transition types. For **macTransType = (source, target, port)** $\in$ **MacTransTypeSet**, source (target) $\in$ MacStepTypeSet corresponds to the source (target) macro step and port $\in$ PortTypeSet is a port type of the target macro step; i.e., port $\in$ target.PortTypeSet.

MacProcTypes denotes the set of all macro process types and **PortTypes** denotes the set of all port types belonging to any macro step type;
i.e., $\forall$ p $\in$ PortTypes: $\exists$ macStepType $\in$ MacStepTypeSet with p $\in$ macStepType.PortTypeSet.
Finally, $\mathbf{sPortTypeSet_{stateType}}$ $\subseteq$ PortTypeSet is the finite set of port types corresponding to a particular stateType.

In the following, a state type represents a tuple **stateType = (name, sMicStepTypeSet, sPortTypeSet)**.

To ensure the correct execution of macro process instances at run-time, like for micro process instances, PHILharmonicFlows prescribes several *structural properties*. First of all, Def. 46 introduces functions for structurally analyzing a macro process type[1].

Definition 46 (Functions for structurally analyzing macro process types):
Let macProcType = (name, oType, MacStepTypeSet, MacTransTypeSet) $\in$ MacProcTypes be a macro process type. Then:

- For each macro step type, **intransCount: MacStepTypeSet** $\mapsto \mathbb{N}_0$ determines the number of incoming macro transition types.
- For each macro step type, **outtransCount: MacStepTypeSet** $\mapsto \mathbb{N}_0$ determines the number of outgoing macro transition types.
- For each macro step type m, **precedingMacroStepTypes: MacStepTypeSet** $\mapsto \mathbf{2^{MacroStepTypes}}$ determines its directly or indirectly preceding macro step types.

[1]We omit a formal definition here due to the intuitive semantics of these functions.

A macro step type without an incoming macro transition type is denoted as *start macro step type*. In turn, macro step types without outgoing macro transition types are called *end macro step types* (cf. Def. 47). Thereby, we require that all start and end macro step types refer to the primary object type of the macro process type. Further, the start macro step type must refer to the start state type of the micro process type corresponding to this primary object type. In turn, all end macro step types must refer to an end state type of this micro process type.

Definition 47 (Start and end macro step types):
Let macProcType = (name, refType, MacStepTypeSet, MacTransTypeSet) $\in$ MacProcTypes be a macro process type. Further, let macStepType = (oType, stateType, PortTypeSet) $\in$ MacStepTypeSet be a corresponding macro step type and micProcType = (oType, MicStepTypeSet, MicTransTypeSet, StateTypeSet, ActivitySet) be the micro process type corresponding to the primary object type of macStepType; i.e., macStepType.oType = micProcType.oType. Finally, startStateType$_{\text{micProcType}}$ $\in$ StateTypeSet corresponds to the start state and EndStateTypes$_{\text{micProcType}}$ $\subset$ StateTypeSet to the set of end state types of micProcType. Then:

- The **startMacStepType** of macProcType is the only macro step type
  macStepType $\in$ MacStepTypeSet with intransCount(macStepType) = 0 $\wedge$
  macStepType.oType = refType $\wedge$ macStepType.stateType = startStateType$_{\text{micProcType}}$.
- An **endMacStepType** of macProcType corresponds to a macro step type
  macStepType $\in$ MacStepTypeSet with outtransCount(macStepType) = 0 $\wedge$
  macStepType.oType = refType $\wedge$ macStepType.stateType $\in$ EndStateTypes$_{\text{micProcType}}$.
  EndMacStepTypes$_{\text{macProcType}}$:= {mSType $\in$ MacStepTypeSet | mSType is an end macro step type} comprises all end macro step types. Thereby, EndMacStepTypes $\neq$ 0.

When defining macro transition types, certain macro step types must not be linked with each other; i.e., the relationship between the object type of the source macro step type and the one of the target macro step must be taken into account. In this context, PHILharmonicFlows utilizes the semantic relationships that exist between the respective object types according to the defined data structure. However, when defining the execution dependencies between micro process types, it is not always sufficient to only consider direct relations between object instances; e.g., relations indicating that a `review` object corresponds to a particular `application` object. To allow for a more sophisticated process coordination, in addition, *indirect* (i.e., *transitive*) as well as *transverse* relationships have to be also considered; e.g., to access all `reviews` related to a particular `job offer` or all `reviews` related to an `interview` referencing the same `application`. Regarding the `job offer` macro process type (cf. Fig. 10.6), for example, there exists a dependency between `reviews` and `interviews` although these two object types do no directly reference each other. Overall, taking the semantic relationships between object instances into account is a key to allow for the concurrent, but well synchronized processing of object instances and their corresponding micro process instances respectively.

To support this, PHILharmonicFlows structures a data model into *data levels* (cf. Def. 48). All object types not referring to any other object type are placed on Level #1. As illustrated in Fig. 10.8A, any other object type is then assigned to a lower data level as the object types it references.

Definition 48 (Data levels):
Let dm = (name, OTypeSet, RelTypeSet) $\in$ DM be an **acyclic** data model. Then:

Function **level** assigns to each object type ot $\in$ OTypeSet its level(ot):
level: OTypeSet $\mapsto \mathbb{N}$ with

$$\text{level(ot)} := \begin{cases} 1, & \nexists \text{rel} \in \text{RelTypeSet: rel.relType.source = ot} \\ 1 + \max\{\text{level}(\text{ot}') \mid & \text{else} \\ (\text{name}, \text{ot}, \text{ot}', \text{min}, \text{max}) \in \text{RelTypeSet}\} & \end{cases}$$

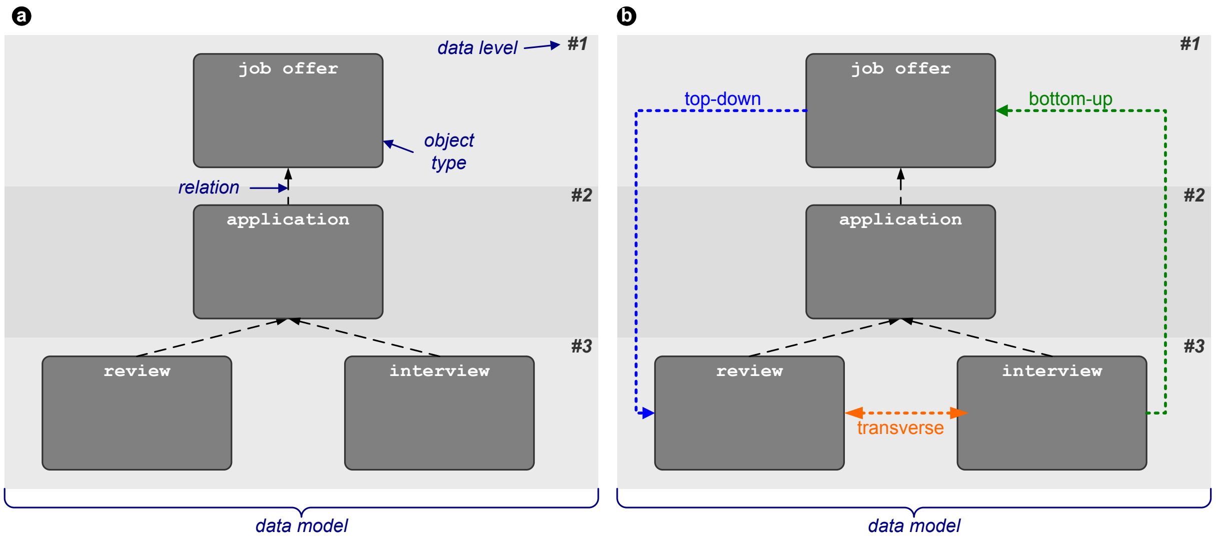


Figure 10.8: Data levels and relationships

Example 10.7 (Organizing the data model in data levels):
Consider Fig. 10.8. Since the `job offer` object type does not reference any other object type, it is placed on the top data level (i.e., data level #1). Further, the `application` object type references the `job offer` object type; hence the `application` object type is assigned to data level #2. In turn, the `application` object type is referenced by the `review` and the `interview` object types. These object types are assigned to data level #3.

As a prerequisite for such an organization of the data model into data levels, the data model must be *acyclic*. To be more precise, cycles must be detected and resolved before applying such a layering to the data model. In this context, we provide function *formsCycle* (cf. Def. 49). For each relation type, it evaluates whether this relation is contained in any cycle of the data model. Relation types for which this applies are visualized for users (cf. Fig. 10.9) who then must manually dissolve them. Therefore, at least one relation type involved in a cycle must be flagged using function *resolvesCycle* (cf. Def. 49) to interrupt the cycle. Selected relation types (i.e., relation types for which function resolvesCycle evaluates to true) are not considered when re-arranging the data structure into different data levels. Altogether, either the data model is acyclic or cycles are resolved using function *resolvesCycle* (cf. Def. 50); i.e., PHILharmonicFlows hierarchically organizes object types.

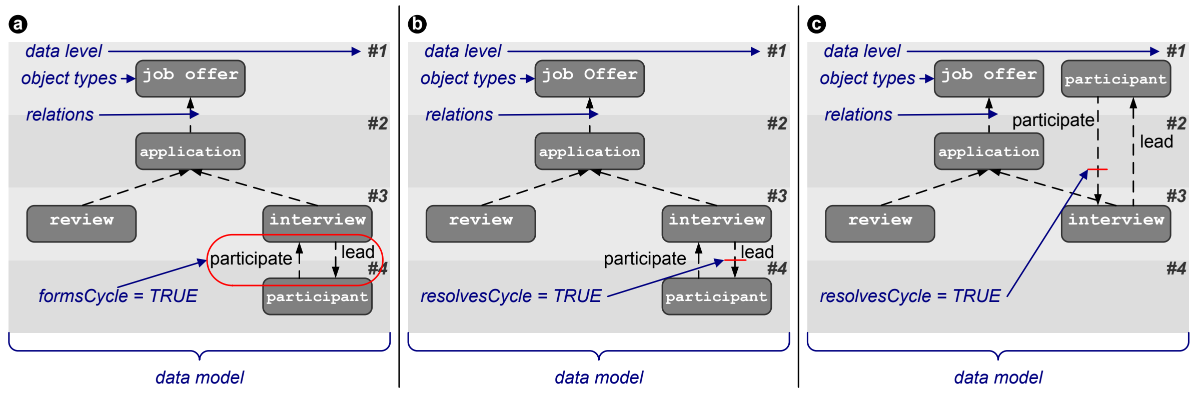


Figure 10.9: Handling cyclic relations

Definition 49 (Functions for structurally analyzing the data model):
Let dm = (name, OTypeSet, RelTypeSet) ∈ DM be a data model. Then:

- **formsCycle: RelTypeSet ↦ BOOLEAN** indicates whether or not a particular relation type is contained in any cycle in dm.
- **resolvesCycle: RelTypeSet ↦ BOOLEAN** optionally marks particular relation types to resolve potential cycles of dm.

Example 10.8 (Handling cyclic relations):
Consider Fig. 10.9. Relation types `participate` and `lead` form a cycle. For them, function formsCycle evaluates to true. Hence, one of these relation types must be flagged, which is then indicated by function resolvesCycle. Regarding Fig. 10.9b, relation type `lead` is flagged. Therefore, object type `participant` is placed at a lower data level as object type `interview`. By contrast, in Fig. 10.9c relation type `participate` is flagged. In this case, the `participant` object type is placed at a higher data level as object type `interview`. Since the `participant` object type does not reference another object type it is placed at data level #1.

Definition 50 (Structural properties of the data model):
Let dm = (name, OTypeSet, RelTypeSet) ∈ DM be a data model. Then:

Either dm is acyclic; i.e., ∀ relType ∈ RelTypeSet: formsCycle(relType) = FALSE,

or cycles have been resolved; i.e.,
$\quad$ RelTypeSet* := {relType ∈ RelTypeSet | resolvesCycle(relType) = FALSE}, ⇒
$\quad$ dm = (name, OTypeSet, RelTypeSet*) is acyclic.

Further, it is important to determine the various relationships existing between two object types. For a particular object type, function *path* specifies a path of relation types through which another object type is transitively referenced.

Definition 51 (Path between two object types):
Let dm = (name, OTypeSet, RelTypeSet) $\in$ DM be a data model. Then:

path: OTypeSet $\times$ $\mathbf{2}^{\mathbf{RelTypeSet}}$ $\mapsto$ **OTypeSet** with path(oType,pathRelTypeSet) = oType' $\wedge$
oType' $\in$ OTypeSet $\wedge$ $\exists$ rt_i = (name, source, target, min, max) $\in$ RelTypeSet: i=1..n with
rt_1.source = oType $\wedge$ rt_1.target = $oType_1$ $\wedge$ rt_2.source = $oType_1$ $\wedge$ rt_2.target = $oType_2$ $\wedge$ ...
$\wedge$ rt_n.source = $oType_{n-1}$ $\wedge$ rt_n.target = $oType_n$ $\wedge$ $oType_n$ = oType'}.

Example 10.9 (Paths between two object types):
Consider Fig. 10.10. Object type `A` is indirectly referenced by object type `D`. Thereby, two paths of relation types exist. By contrast, object type `F` is not referenced by object type `D`.

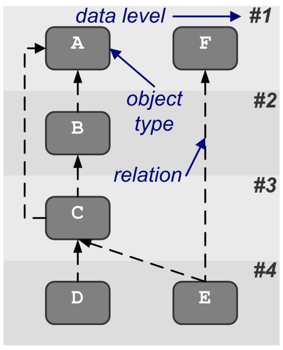


Figure 10.10: Paths between object types

We denote an object type `A` directly or indirectly referencing an object type `B` as *lower-level object type* of `B` (cf. Def. 52). Accordingly, an object type directly or indirectly referenced by other object types is denoted as *higher-level object type* (cf. Def. 52).

Definition 52 (Higher- and lower-level object types):
Let dm = (name, OTypeSet, RelTypeSet) $\in$ DM be an **acyclic** data model (i.e., dm may contain cycles resolved by using function resolvesCycle). Then:

- **higherlevel: OTypeSet** $\mapsto$ $\mathbf{2}^{\mathbf{OTypeSet}}$ with
  higherlevel(ot) := {ot' $\in$ OTypeSet | level(ot) < level(ot') $\wedge$
  $\exists$ RelTypeSet' $\subseteq$ RelTypeSet with path(ot,RelTypeSet') = ot'}.
- **lowerlevel: OTypeSet** $\mapsto$ $\mathbf{2}^{\mathbf{OTypeSet}}$ with
  lowerlevel(ot) := { ot' $\in$ OTypeSet | ot $\in$ higherlevel(ot') }

As illustrated in Fig. 10.8b, a relationship between object types `A` and `B` is categorized as *top-down (bottom-up)*, if `B` is a lower-level (higher-level) object type of `A`. Furthermore, we categorize a relationship between object types `A` and `B` as *transverse* if there exists another object type `C` of which both `A` and `B` are lower-level object types (i.e., `A` and `B` have a higher-level object type in common). These three relationships are defined by Def. 53.

Definition 53 (Relationships):
Let dm = (name, OTypeSet, RelTypeSet) $\in$ DM be an **acyclic** data model.
Further, let $ot_i \in$ OTypeSet, i=1,...,3 be different object types. Then:

- **top-down: OTypeSet $\times$ OTypeSet $\mapsto$ BOOLEAN** with

$$\textbf{top-down}(ot_1,ot_2) := \begin{cases} \textbf{true}, & ot_1 \in \text{higherlevel}(ot_2) \\ \textbf{false}, & \text{else} \end{cases}$$

- **bottom-up: OTypeSet $\times$ OTypeSet $\mapsto$ BOOLEAN** with

$$\textbf{bottom-up}(ot_1,ot_2) := \begin{cases} \textbf{true}, & ot_1 \in \text{lowerlevel}(ot_2) \\ \textbf{false}, & \text{else} \end{cases}$$

- **transverse: OTypeSet $\times$ OTypeSet $\mapsto$ BOOLEAN** with

$$\textbf{transverse}(ot_1,ot_2) := \begin{cases} \textbf{true}, & \exists\, ot_3\colon ot_i \in \text{lowerlevel}(ot_3)\colon i=1,2 \\ \textbf{false}, & \text{else} \end{cases}$$

Example 10.10 (Relationships):
Consider Fig. 10.11.

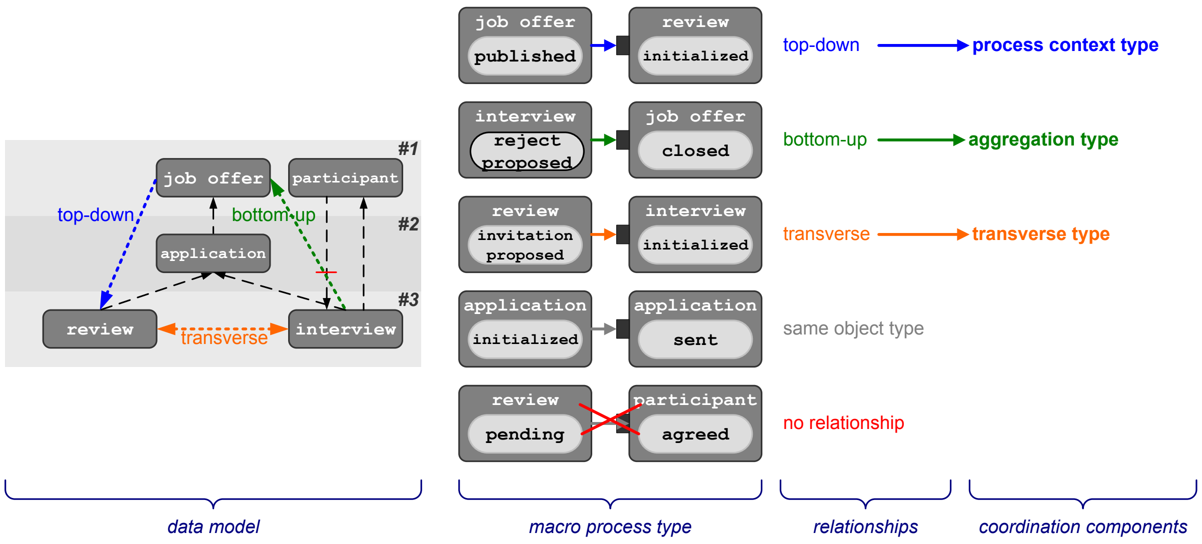
a) **Top-down:** State `initialized` of a `review` micro process instance may only be activated if state `published` of the corresponding `job offer` micro process instances is activated. This constitutes a top-down relationship since several lower-level `review` micro process instances depend on the execution of a higher-level `job offer` micro process instance. Note that `job offer` (belonging to the source macro step type) is a higher-level object type of object type `review` (belonging to the target macro step type).

b) **Bottom-up:** The activation of state `closed` of a `job offer` micro process instance depends on the activation of state `reject proposed` for all related `interview` micro process instances. This constitutes a bottom-up relationship since one higher-level `job offer` micro process instance depends on the execution of all lower-level `interview` micro process instances. Note that `interview` (belonging to the source macro step type) is a lower-level object type of object type `job offer` (belonging to the target macro step type).

c) **Transverse:** The activation of state `initialized` of an `interview` micro process instance depends on the activation of state `invitation proposed` of the `review` micro process instances belonging to the same `application`. This constitutes a transverse relationship since several lower-level `interview` micro process instances depend on the execution of all lower-level `review` micro process instance referencing the same higher-level `application`. Note that the `review` object type (belonging to the source macro step type) and `interview` object type (belonging to the target macro step type) have a common higher-level object type (i.e., `application`).

As illustrated in Fig. 10.11, macro transition types may only connect macro step types whose object types have a top-down, bottom-up, or transverse relationship to each other, or which refer to the same object type (cf. Def. 54f). Otherwise, it would not be possible to asynchronously execute and synchronize the corresponding micro process instances (cf. Sect. 11).

Taking solely macro step types and their incoming and outgoing macro transition types into account, a macro process type must be *acyclic* (cf. Def. 54a). Further, according to Def. 54b, each macro process type contains exactly *one start macro step type* and *at least one end macro step type* (cf. Def. 54c). All other macro steps must have at least one incoming as well as one outgoing macro transition type (cf. Def. 54d + e). Note that properties a-e ensure *reachability of each macro step type*. In particular, each macro step can be reached starting from the start macro step. Finally, from any macro step, at least one end macro step can be reached.



#1
#2
#3
job offer
participant
application
review
interview
top-down
bottom-up
transverse
job offer published → review initialized
interview reject proposed → job offer closed
review invitation proposed → interview initialized
application initialized → application sent
review pending → participant agreed
top-down → process context type
bottom-up → aggregation type
transverse → transverse type
same object type
no relationship
data model
macro process type
relationships
coordination components

Figure 10.11: Macro transition types and corresponding relationships

To ensure correct process execution at run-time, two critical situations must be taken into account. First, regarding deterministic micro process types, we must ensure that the execution dependencies specified by the macro transition types do not lead to a *mutual waiting* (cf. Fig. 10.12).

Example 10.11 (Mutual waiting):
Consider Fig. 10.12. State `published` of a `job offer` micro process instance may only be activated if state `planned` of a related `interview` micro process instance becomes activated. To initialize an `interview`, however, the corresponding `job offer` must be in state `closed`. Since state `closed` of the `job offer` micro process instance succeeds state `published`, a deadlock will occur; i.e., both micro process instances will be waiting for each other.

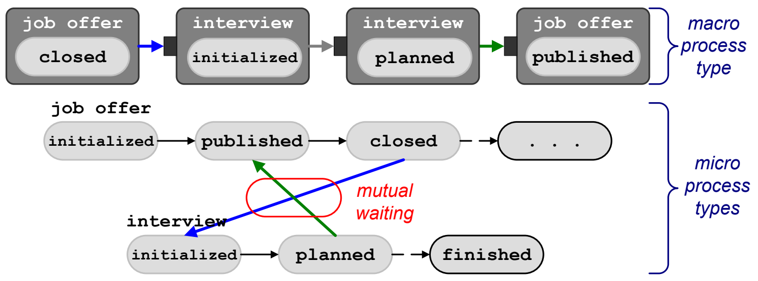


Figure 10.12: Deadlock situation using deterministic micro process types

When using non-deterministic micro process types, corresponding execution dependencies must consider that at most one execution path is selectable at run-time. For this reason, we must avoid that any execution sequence of the macro process type comprises state types belonging to different alternative execution paths; i.e., *mutual exclusions* as illustrated in Fig. 10.13 must be prevented.

Example 10.12 (Mutual exclusion):
Consider Fig. 10.13. Regarding the macro process type, state `reject proposed` of a `review` micro process instance is followed by state `invitation proposed`; i.e., after activating state `reject proposed`, state `invitation proposed` must be activated. Since these states belong to different alternative execution paths in the corresponding `review` micro process instance, these two states exclude each other.

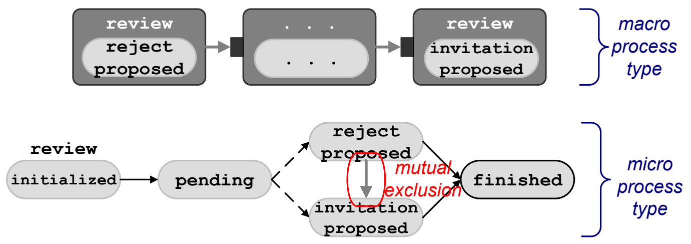


Figure 10.13: Deadlock situations using non-deterministic micro process types

To prevent *mutual waitings* as well as *mutual exclusions*, each macro step type referring to the same object type as one of its predecessors must refer to a state type that succeeding the one of the preceding macro step type (cf. Def. 54g). Nevertheless, non-deterministic micro process types might lead to deadlocks at run-time if the state type required by a macro step type is not reached. Since the number of object instances created during run-time is unknown at build-time, such situations cannot be prevented in the forefront. Instead, they must be properly handled at run-time (see Sect. 13.2 for details).

Finally, we must consider that lower-level object instances do not exist before the higher-level ones they refer to are created. Hence, any macro transition type categorized as bottom-up must not target at a macro step type referring to a start state type (cf. Def. 54h).

Definition 54 (Structural properties of macro process types):
Let dm = (name, OTypeSet, RelTypeSet) $\in$ DM be an acyclic data model. Let further macProcType = (name, oType, MacStepTypeSet, MacTransTypeSet) $\in$ MacProcTypes be a macro process type. Then:

a) macProcType is acyclic.

b) $\exists!$ s = startMacStepType $\in$ MacStepTypeSet;
i.e., there exists exactly one start macro step type.

c) EndMacStepTypes $\neq \emptyset$; i.e., there exists at least one end macro step type.

d) $\forall$ m $\in$ MacStepTypeSet - {startMacStepType}: intransCount(m) $\geq$ 1;
i.e., all macro step types except the start macro step type have at least one incoming macro transition type.

e) $\forall$ m $\in$ MacStepTypeSet - EndMacStepTypes: outtransCount(m) $\geq$ 1;
i.e., all macro step types except end macro step types have at least one outgoing macro transition type.

f) $\forall$ m = (source, target, port) $\in$ MacTransTypeSet:
top-down(source.oType, target.oType) = true $\vee$
bottom-up(source.oType, target.oType) = true $\vee$
transverse(source.oType, target.oType) = true $\vee$
source.oType = target.oType;
i.e., all macro transition types may only connect macro step types whose object types constitute a top-down, bottom-up, or transverse relationship to each other, or the macro step types refer to the same object type.

g) $\forall$ $m_1 \in$ MacStepTypeSet:
  $\forall$ $m_2 \in$ precedingMacStepTypes(m_1) with m_1.oType = m_2.oType:
    m_2.stateType $\in$ precedingStateTypes(m_1.stateType);
i.e., if a successor macro step type refers to the same object type as one of its predecessors, its state type must be a successor of the predecessor state type.

h) $\forall$ macTransType = (s, t, port) $\in$ MacTransTypeSet with bottom-up(s.oType, t.oType) = true:
  t.stateType != startStateType;
i.e., macro transition types categorized as bottom-up must not target macro step types referring to the start state type.

10.4 Summary

To capture and model object interactions, a macro process type may be defined. As opposed to traditional process modeling approaches, where process steps are defined in terms of activities, a macro step type always refers to an object type together with a corresponding state type. To take the dynamically evolving number of object instances as well as their asynchronous execution into account, for each macro transition type, a corresponding coordination component type needs to be defined. For this purpose, PHILharmonicFlows takes the relationship between the object types of the source and the target macro steps into account. To cover this, we automatically structure the data model into different data levels. All object types not referring to any other object type are placed on Level #1. Any other object type is always assigned to a lower data level as the object types it references. This way, we can automatically categorize macro transition types. The required coordination component then depends on the type of the respective macro transition type.

11 Coordination Components

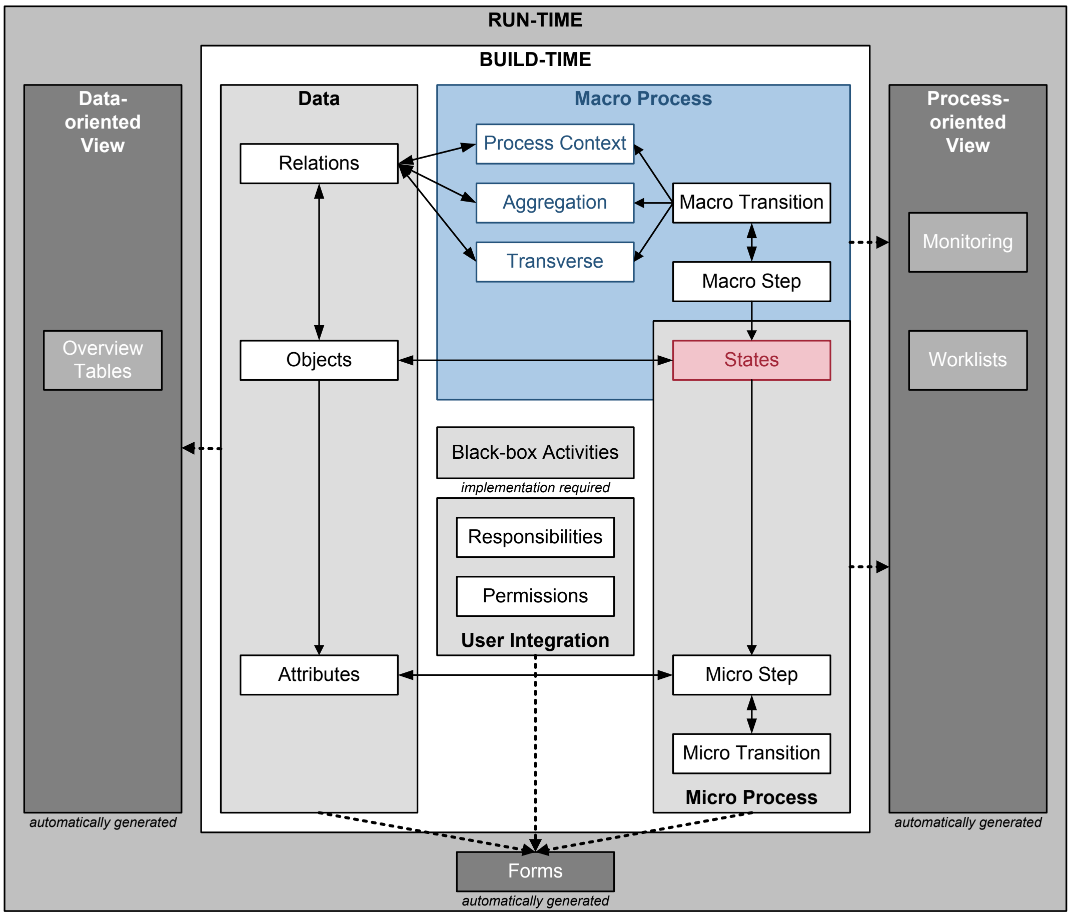


Figure 11.1: Coordination components in PHILharmonicFlows

At run-time, a macro process instance coordinates the concurrent processing of inter-related object instances. Thereby, for each object type a set of object instances exist. Furthermore, an object instance of a particular type may be referred by related collections of object instances corresponding to lower-level object types. Overall, this results in large and complex process structures. Accordingly, the execution of a set of lower-level micro process instances may depend on a higher-level micro process instance and vice versa.
By contrast, macro process types, as shown in Chapt. 10, are modeled in a flat and compact way abstracting from the underlying process structure and its complexity. Note that in this context, the micro process instances may be assigned to different states of a macro process instance. Thus, the execution of a macro process instance is not that strict sequential as in traditional approaches. More precisely, at a certain point in time, several macro steps are reached by a subset of their related micro process instances. Hence, the dependencies between micro process instances, as defined by the macro process type, must obey a well-defined operational semantics to properly coordinate these micro process instances. More precisely, for each macro transition type, a corresponding *coordination component type* must be defined (cf. Fig. 11.1). In turn, this coordination component should consider the type of relationship existing between the object types referenced by the source and target macro step type of this transition. While for each macro transition type representing a top-down relationship a *process context type* must be specified, bottom-up relationships require an *aggregation type*. In turn, for relationships of type transverse a *transverse type* should be used.

As a prerequisite for coordinating micro process instances at run-time, therefore, the semantic relations of the corresponding object instances (i.e., data structure) must be structurally analyzed. In this context, for a particular object type, *referencedOI* determines the object instance referenced by a path of relation types (cf. Def. 55).

Definition 55 (Referenced object instance):
Let dm = (name, OTypeSet, RelTypeSet) $\in$ DM be a data model and ds = (dm, OSet, RelSet, M_{Cc}) be a corresponding data structure. Then:

referencedOI: OSet $\times$ $2^{\text{RelTypeSet}}$ $\mapsto$ OSet with referencedOI(oid, pathRelTypeSet) := oid' where oid' $\in$ OSet:
$\exists$ r_i = (relType, soid, toid) $\in$ RelSet: i=1..n $\wedge$ $\forall$ r_i: r_i.relType $\in$ pathRelTypeSet:
r_1.soid = oid $\wedge$ r_1.toid = r_2.soid $\wedge$ r_2.soid = r_3.toid $\wedge$ ...
$\wedge$ r_{n-1}.toid = r_n.soid $\wedge$ r_n.toid = oid'.

Example 11.1 (Referenced object instance):
Consider Fig. 11.2. `Review #7` indirectly references `job offer #1` through a relation type. More precisely, this path comprises the relation type connecting the `review` and the `application` object types as well as the relation type connecting the `application` and `job offer` object types.

Further, we denote all object instances directly or indirectly referenced by a particular object instance `o` (i.e., reached on any path starting from `o`) as *higher-level object instances* of `o`. Accordingly, we denote the object instances directly or indirectly referencing an object instance `o'` as a *lower-level object instance* of `o'` (cf. Def. 56).[1]

[1]Remember that the higher an object instance in the hierarchy is the lower its level # will be.

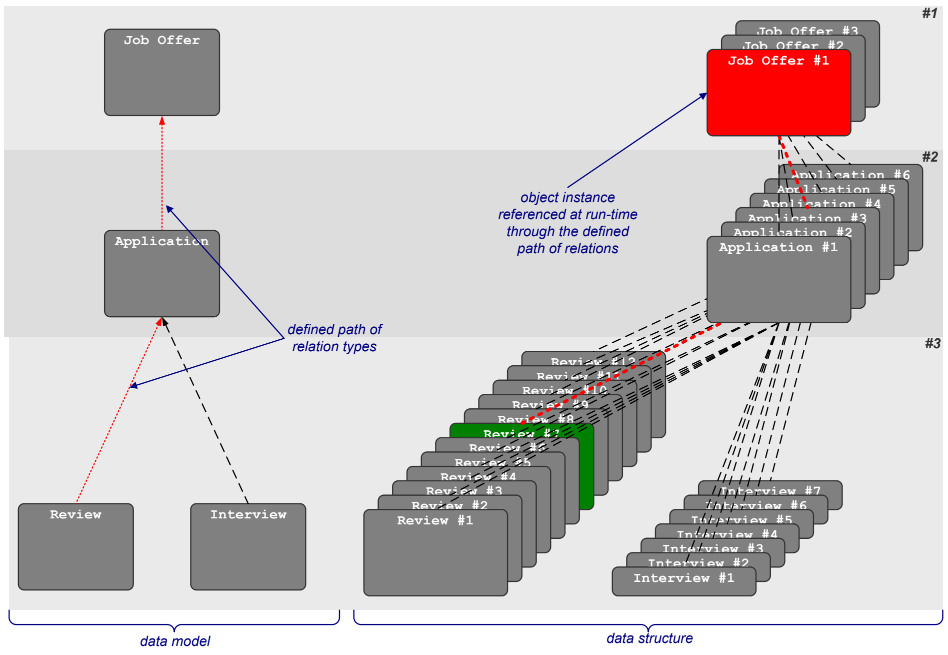


Figure 11.2: Determining referenced object instances

Definition 56 (Higher- and lower-level object instances):
Let dm = (name, OTypeSet, RelTypeSet) $\in$ DM be a data model and ds = (dm, OSet, RelSet, M_{Cc}) be a corresponding data structure. Then:

- **higherlevelOIs: OSet $\mapsto$ 2^{OSet}** with
  higherlevel(o) := {o' $\in$ OSet | level(o'.oType) $\leq$ level(o.oType) $\wedge$
  $\exists$ RelSet' $\subseteq$ RelSet with referencedOI(o,RelSet') = o'.
- **lowerlevelOIs: OSet $\mapsto$ 2^{OSet}** with
  lowerlevel(o) := {o' $\in$ OSet | o $\in$ higherlevelOIs(o')}

Example 11.2 (Higher- and lower-level object instances):
Consider Fig. 11.3. `Application #3` and `job offer #1` are the higher-level object instances of object instance `review #7`. In turn, the lower-level object instances of `job offer #1` comprise `application` object instances `#1 - #3`, `review` object instances `#1 - #8`, and `interview` object instances `#1 - #6`.

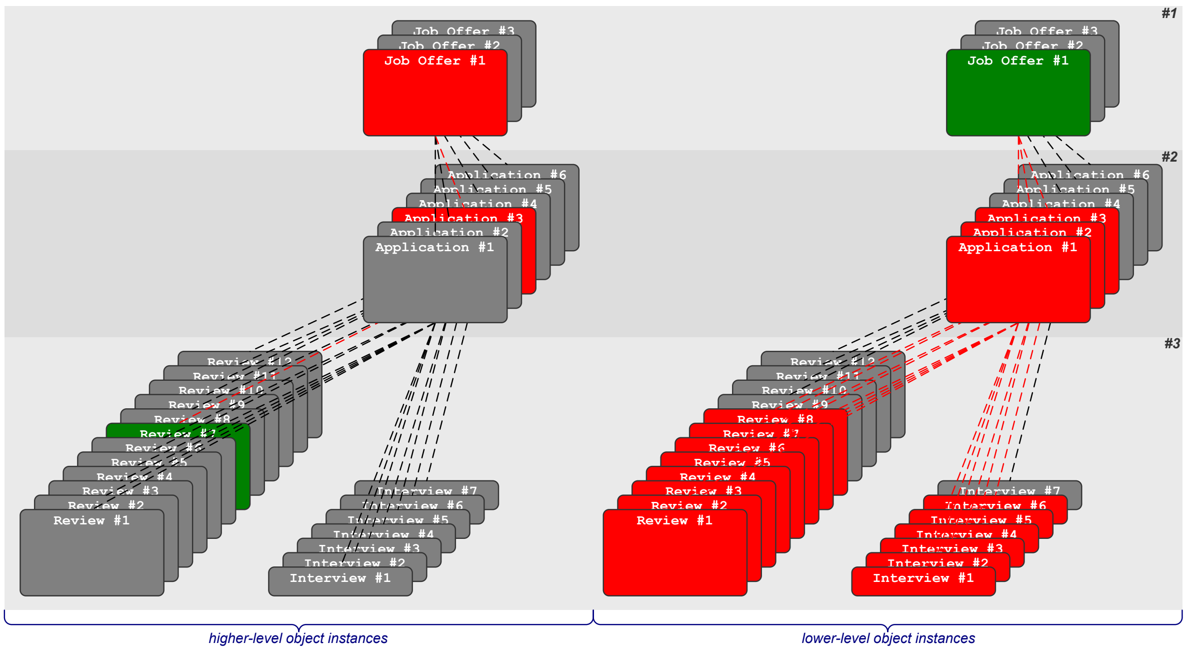


Figure 11.3: Higher- and lower-level object instances

11.1 Process Contexts

We first consider *top-down relationships* within a data structure. For them, the execution of a set of lower-level micro process instances depends on the one of a higher-level micro process instance. Regarding Fig. 11.4, for example, the execution of a set of `application` micro process instances depends on the execution of the corresponding `job offer` micro process instance. To cover this behavior in a macro process type, for each macro transition type categorized as top-down, a corresponding *process context type* must be defined. The latter specifies whether or not the respective state of a lower-level micro process instance, as specified by the target macro step, may be activated; i.e., the activation of this state depends on the state of the referenced higher-level micro process instance (as specified by the source macro step type).

To differentiate between AND- and OR-semantics during micro process instance coordination, port types manage the activation of states (cf. Exs. 10.4 and 10.5). Thereby, each port type belongs to a particular state type of a macro step type (cf. Sect. 10.3). As illustrated in Fig. 11.5, a process context type comprises the port type that belongs to the state type referenced by the target macro step type (cf. portType in Def. 57) as well as the micro process type corresponding to the source macro step type (cf. sourceMicProcType in Def. 57).

To allow for the asynchronous execution of the higher-level micro process instance, a process context type additionally stores subsequent states of the higher-level micro process type, which also enable the lower-level instances to activate the state specified by the target macro step type. As example consider the processing of `job offers` as illustrated in Fig. 11.5. Here, a `job offer` micro process instance may activate state `closed` even if not all corresponding `interviews` have been `planned`; i.e., it is possible to process `job offers` while `interviews` may further activate

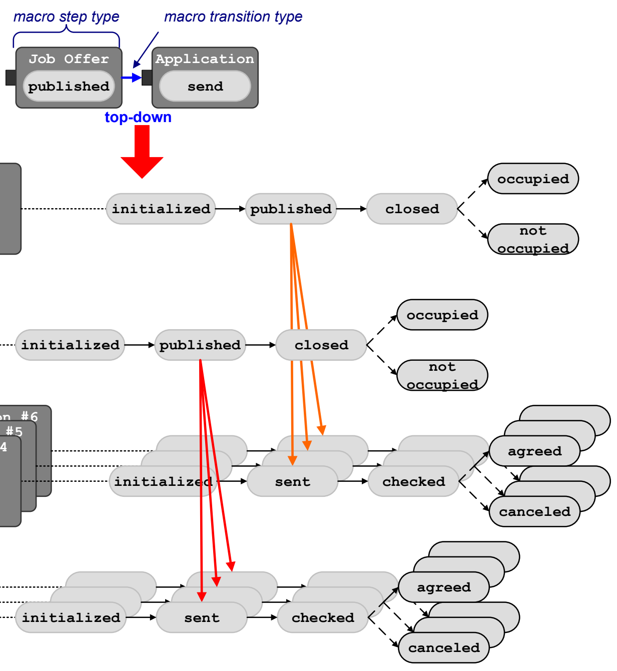


Figure 11.4: Top-down relationship

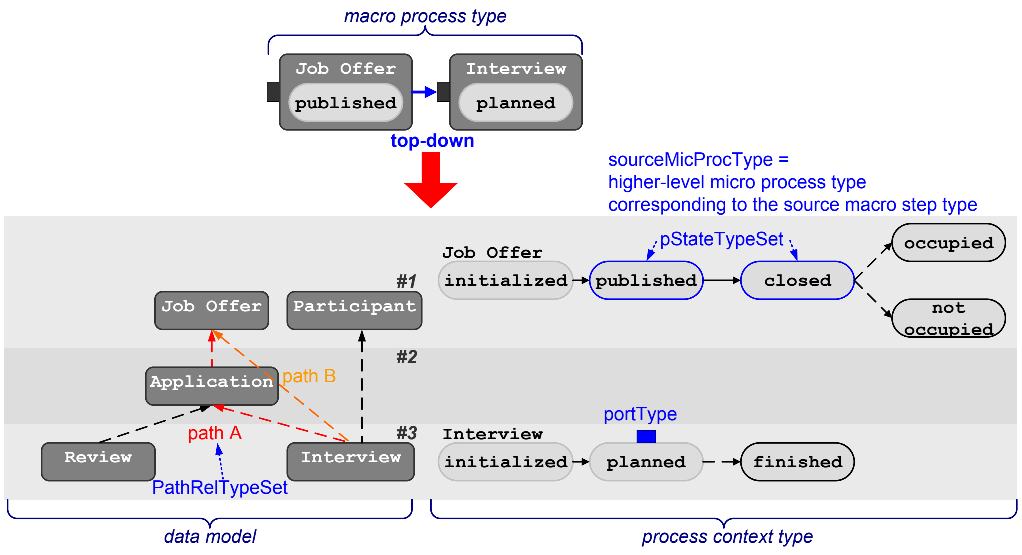


Figure 11.5: Process context type

state `planned`. This way, at run-time, the execution of the higher-level micro process instance must not be blocked until all lower-level micro process instances have activated the respective state. For this purpose, each process context type comprises a set of states belonging to its source micro process type (cf. pStateTypeSet in Def. 57). In particular, each process context type contains at least the state relating to the source macro step type, but subsequent states may be optionally added if required. This way, an asynchronous execution of different micro process instances is possible. However, only states succeeding the state defined by the source macro step type may be chosen. If one of these states becomes activated at run-time, all lower-level micro process instances referring to the higher-level one may then activate the state specified by the target macro step type.

A top-down relationship must not necessarily coincide with a direct relation of the data model, i.e., it may represent a transitive (i.e., indirect) relationship between two object types as well. Further, note that there may be several paths within a data structure based on which a particular higher-level object instance may be reached from a lower-level one (cf. Fig. 11.5). To avoid ambiguities, therefore, the path between the two object types involved (i.e., the object types referenced by the source and target macro step types) are specified as well.

Definition 57 (Process context types):
Let dm = (name, OTypeSet, RelTypeSet) be a data model and macProcType = (name, oType, MacStepTypeSet, MacTransTypeSet) ∈ MacProcTypes be a macro process type. Further, let macTransType = (s, t, pType) ∈ MacTransTypeSet be a top-down macro transition type; i.e., top-down(s.oType, t.oType) = true. Then:

A **process context type** is defined as tuple **pcType = (pType, sourceMicProcType, pStateTypeSet, PathRelTypeSet)** where:

- pType is the port type that belongs to the state type of the target macro step type of macTransType.
- sourceMicProcType is the micro process type corresponding to the source macro step type;
  i.e., sourceMicProcType.oType = s.oType.
- pStateTypeSet ⊆ sourceMicProcType.StateTypeSet is a set of state types corresponding to sourceMicProcType with:
  - s.stateType ∈ pStateTypeSet;
    i.e., pStateTypeSet comprises at least the state type of the source macro step type.
  - ∀ stateType ∈ pStateTypeSet: s.stateType ∈ precedingStateTypes(stateType);
    i.e., all other state types in pStateTypeSet succeed the state type corresponding to the source macro step type.
- PathRelTypeSet ⊆ RelTypeSet is a finite set of relation types describing a path from the lower-level object type of the target macro step type to the higher-level object type of the source macro step type;
  i.e., path(t.oType,PathRelTypeSet) = s.oType.

PCTypes corresponds to the set of all definable process context types. Further, **$\textbf{PCTypes}_{\textbf{pType}}$** denotes the set of all process context types belonging to the same port type; i.e., $\text{PCTypes}_{\text{pType}}$:= {p ∈ PCTypes | p.portType = pType}.

Example 11.3 (Process context type):
Consider Fig. 11.5. Since the macro transition type between the macro step type referring to the `job offer` object type in state `published` and the macro step type referring to the `interview` object type in state `planned` is categorized as top-down, a corresponding process context type must be defined. The latter comprises at least state `published` which is referenced by its source macro step type. Further, state `closed` is added to this process context type, which allows activating state `planned` of the `interview` micro process instance even if state `closed` of the corresponding `job offer` micro process instance has been already activated; i.e., the `job offer` micro process instance may asynchronously continue its execution. To determine the concrete `job offer` micro process instances, a particular `interview` micro process instance depends on, at run-time, the set of relations referring to the `application` object instances is evaluated; i.e., `pathA` (cf. Fig. 11.5) is chosen.

At run-time, a process context is initialized for each lower-level micro process instance (cf. Fig. 11.6). Depending on the currently activated state of the higher-level micro process instance, it controls whether or not the state of the lower-level micro process instance may be activated. For this purpose, each *process context instance* (cf. Def. 58) comprises the port corresponding to the lower-level micro process instance and the referenced higher-level micro process instance (cf. port and sourceMicProcInstance in Def. 58). Only if one state defined by the corresponding process context type (cf. pStateTypeSet in Def. 57) is activated for the higher-level micro process instance, the lower-level one may activate the respective state (cf. Chapt. 12).

Definition 58 (Process context instance):
Let micProcInstance be an instance of a micro process type micProc $\in$ MicProcTypes; i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

A **process context instance** is a tuple **pcInstance = (pcType, port, sourceMicProcInstance)** where

- pcType is a process context type.
- port is the instance of pcType.portType which belongs to a state of micProcInstance:
  $\exists$ state $\in$ micProc.StateSet $\wedge$ port $\in$ state.sPortSet.
- sourceMicProcInstance is an instance of pcType.sourceMicProcType referenced by micProcInstance; i.e.,
  pcType.sourceMicProcInstance $\in$ micprocinstances(pcType.sourceMicProcType) $\wedge$
  sourceMicProcInstance.oid = referencedOI(micProcInstance.oid, pcType.PathRelTypeSet).

PCInstances denotes the set of all process context instances corresponding to any process context type pcType $\in$ PCTypes. Further, **PCInstances**$_{\textbf{port}}$ corresponds to the set of all process context instances referring to the same port. Finallly, **pcinstances: PCTypes** $\mapsto$ $\mathbf{2^{PCInstances}}$ assigns to each process context type pcType $\in$ PCTypes its corresponding instances pcinstances(pcType) $\subseteq$ PCInstances.

Example 11.4 (Process context instances):
Consider a process context type that enables an `application` micro process instances to activate state `send`. As illustrated in Fig. 11.6, at run-time, for `job offer #1` three `application` object instances exist. Hence, three instances of the process context type are initialized. Each of them corresponds to an `application` micro process instance controlling whether or not state `send` can be activated.

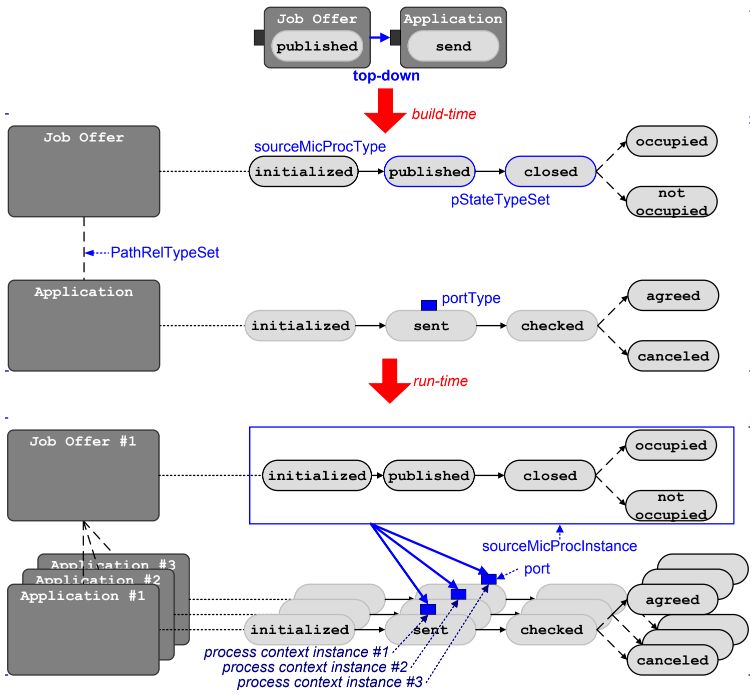


Figure 11.6: Process context instance

11.2 Aggregations

Aggregation types provide another fundamental concept for coordinating micro process instances. An aggregation type must be defined for each macro transition type categorized as *bottom-up* (cf. Fig. 11.7). Compared to process context types, aggregation types work the other way around; i.e., the execution of a particular higher-level micro process instance is coordinated with the one of a number of related lower-level micro process instances. Consider thereto Prop. 5 (see Sect. 3.1) according to which an `application` may only be rejected if all corresponding `reviews` propose the rejection. More precisely, whether the higher-level micro process instance may activate a certain state depends on the activated states of the lower-level instances. In this context, the state of the higher-level micro process instance is defined by the target macro step type. In turn, the source macro step type specifies the state of the lower-level instances.

At run-time, an aggregation instance is initiated for each higher-level micro process instance aggregating corresponding lower-level micro process instances. For this purpose, Def. 59 provides functions for counting lower-level micro process instances.

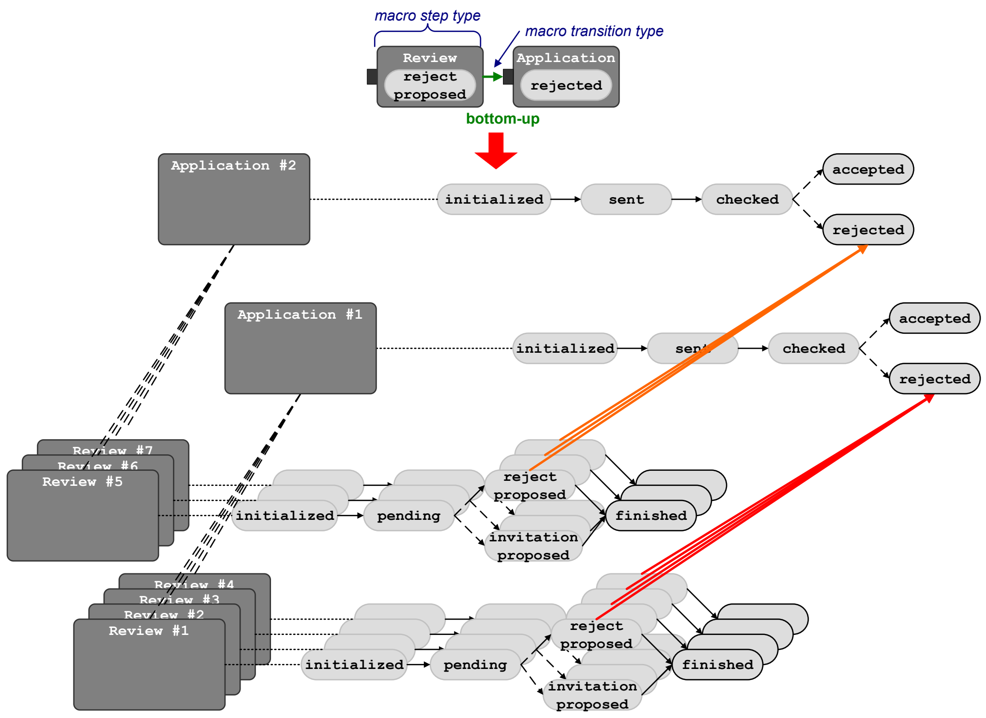


Figure 11.7: Bottom-up relationship

Definition 59 (Micro process instance counter):
Let micProcType = (oType, MicStepSet, MicTransSet, StateSet, BackTransSet, ActivitySet) ∈ MicProcTypes be a micro process type and MicProcSet := micprocinstances(micProcType). Further, let stateType ∈ StateSet be a state of micProcType. Then:

- **#ALL: $2^{\text{MicProcInstances}}$ $\mapsto$ INTEGER** with #ALL(MicProcSet) = |MicProcSet|;
  #ALL determines the total number of micro process instances from a given instance set.
- **#IN: $2^{\text{MicProcInstances}}$ $\times$ stateType $\mapsto$ INTEGER** with
  #IN(MicProcSet, stateType) = | {micProc ∈ MicProcSet with M_{State}(stateType) = ACTIVATED} |;
  #IN determines the number of micro process instances from a given instance set for which the state is currently marked as ACTIVATED.
- **#BEFORE: $2^{\text{MicProcInstances}}$ $\times$ stateType $\mapsto$ INTEGER** with
  #BEFORE(MicProcSet, stateType) = | {micProc ∈ MicProcSet with M_{State}(stateType) = WAITING} |;
  #BEFORE determines the number of micro process instances from a given instance set for which the state is currently marked as WAITING.
- **#AFTER: $2^{\text{MicProcInstances}}$ $\times$ stateType $\mapsto$ INTEGER** with
  #AFTER(MicProcSet, stateType) = | {micProc ∈ MicProcSet with M_{State}(stateType) = CONFIRMED} |;
  #AFTER determines the number of micro process instances from a given instance set for which the state is currently marked as CONFIRMED.
- **#SKIPPED: $2^{\text{MicProcInstances}}$ $\times$ stateType $\mapsto$ INTEGER** with
  #SKIPPED(MicProcSet, stateType) = | {micProc ∈ MicProcSet with M_{State}(stateType) = SKIPPED} |;
  #SKIPPED determines the number of micro process instances from a given instance set for which the state is currently marked as SKIPPED.

Each aggregation type comprises the port type of the higher-level micro process type as well as the lower-level micro process type itself (cf. portType and sourceMicProcType in Fig. 11.8). When defining an aggregation type, the counters may be used for specifying a predicate. The default predicate of an aggregation type is `#IN = #ALL`; i.e., for all lower-level micro process instances the defined state must be activated. However, this predicate may be overwritten introducing more complex conditions; i.e., additional counters may be used referring to the number of micro process instances for which the respective state is currently not reached (i.e., marked as WAITING), was already reached (i.e., marked as CONFIRMED), or belongs to an alternative path skipped during execution (i.e., marked as SKIPPED). These counters allow continuing the execution of the lower-level micro process instances even if the higher-level micro process instance has not reached the respective state yet. In addition, simple arithmetic expression may be used; e.g., `#IN = #ALL - 3`. At run-time, the defined predicate is then evaluated based on the states of corresponding lower-level micro process instances.

Transitive relationships may be also considered when defining macro transition types representing bottom-up relationships. Like for process context types, the desired path of relation types must be specified for an aggregation type as well (cf. PathRelTypeSet in Fig. 11.8).

Definition 60 (Aggregation types):
Let dm = (name, OTypeSet, RelTypeSet) be a data model and macProcType = (name, oType, MacStepTypeSet, MacTransTypeSet) $\in$ MacProcTypes be a macro process type. Further, let macTransType = (s, t, pType) $\in$ MacTransTypeSet be a bottom-up macro transition type; i.e., bottom-up(s.oType, t.oType) = true. Then:

An **aggregation type** is defined as tuple **aggType = (pType, sourceMicProcType, sourceStateType, aggConstraint, PathRelTypeSet)** with:

- pType is the port type that belongs to the state type of the target macro step type of macTransType.
- sourceMicProcType is the micro process type of the source macro step type; i.e.;
  sourceMicProcType.oType = s.oType.
- sourceStateType = s.stateType is the state type based on which the micro process instances are aggregated at run-time.
- aggConstraint: MicProcInstances $\times$ sourceStateType $\mapsto$ BOOLEAN is a predicate based on the counters introduced in Def. 59 and simple artithmetic expressions.
- PathRelTypeSet $\subseteq$ RelTypeSet is a finite set of relation types describing a path from the lower-level object type of the source macro step type to the higher-level object type of the target macro step type; i.e.,
  path(s.oType,PathRelTypeSet) = t.oType.

AGGTypes corresponds to the set of all definable aggregation types. Further, **AGGTypes$_{\text{pType}}$** denotes the set of all aggregation types belonging to the same port type; i.e.; AGGTypes$_{\text{pType}}$:= {a $\in$ AGGTypes | a.portType = pType}.

Example 11.5 (Aggregation type):
Consider Fig. 11.8. Since the macro transition type between the macro step type referring to the `review` object type in state `reject proposed` and the macro step type referring to the `application` object type in state `rejected` is categorized as bottom-up, a corresponding aggregation type must be defined. In the given case, the predicate is defined as `"#IN + #AFTER = #ALL"`; i.e., an `application` may only be `rejected` if all corresponding `reviews` propose rejection. Since the aggregation predicate comprises counter #AFTER as well, on the one hand, a `review` may be finished (state `finished` can be activated) even if the `application` has not reached state `rejected`. On the other hand, state `rejected` of the `application` may also be activated when some `reviews` are already `finished`.

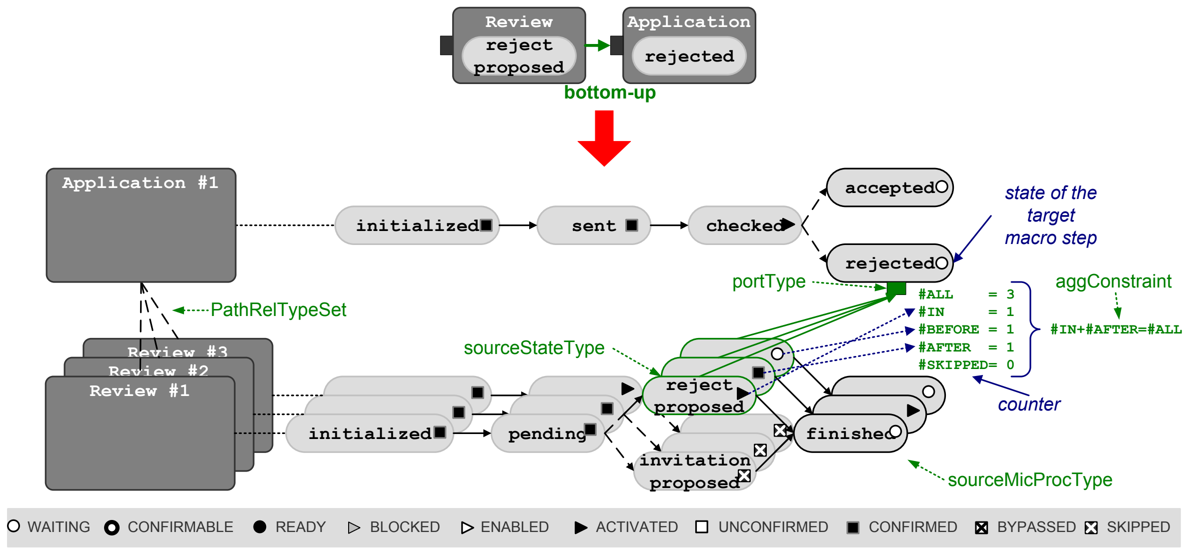


Figure 11.8: Aggregation type

Opposed to process context types, which are initiated for each lower-level micro process instance, an aggregation is initiated for each higher-level one (cf. Fig. 11.9). Each aggregation instance manages the complete set of corresponding lower-level micro process instances (cf. Def. 61). If a new relation is assigned at run-time, the affected micro process instance will be added.

Definition 61 (Aggregation instance):
Let micProcInstance be an instance of a micro process type micProc $\in$ MicProcTypes; i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

An **aggregation instance** is a tuple **aggInstance = (aggType, port, SourceMicProcSet, eval)** where

- aggType is an aggregation type.
- port is the instance of the portType that belongs to the state of micProcInstance as referred by the target macro step: $\exists$ state $\in$ micProc.StateSet $\wedge$ port $\in$ state.sPortSet.
- SourceMicProcSet is a set of micro process instances of aggType.sourceMicProcType that reference micProcInstance:
  $\forall$ sMicProcInstance $\in$ SourceMicProcSet:
  sMicProcInstance $\in$ micprocinstances(aggType.sourceMicProcType) $\wedge$
  micProcInstance.oid = referencedOI(sMicProcInstance.oid, aggType.PathRelTypeSet).
- eval: aggConstraint $\times$ SourceMicProcSet $\mapsto$ BOOLEAN is a function evaluating the predicate defined by aggConstraint in respect to instance set SourceMicProcSet; i.e., functions #ALL, #IN, #AFTER, #BEFORE, and #SKIPPED are evaluated based on SourceMicProcSet.

AGGInstances denotes the set of aggregation instances of any aggregation type aggType $\in$ AGGTypes. Further, **AGGInstances$_{\text{port}}$** is the set of all aggregation instances targeting at same port. Finallly, **agginstances: AGGTypes $\mapsto$ 2$^{\text{AGGInstances}}$** assigns to each aggregation type aggType $\in$ AGGTypes its corresponding instances agginstances(aggType) $\subseteq$ AGGInstances.

Example 11.6 (Aggregation instances):
Consider Fig. 11.9. For `application #1`, four `review` object instances exist. Since an aggregation instance is initiated for each higher-level micro process instance, only one aggregation instance exists. The latter comprises all four `review` micro process instances referring to `application #1`.

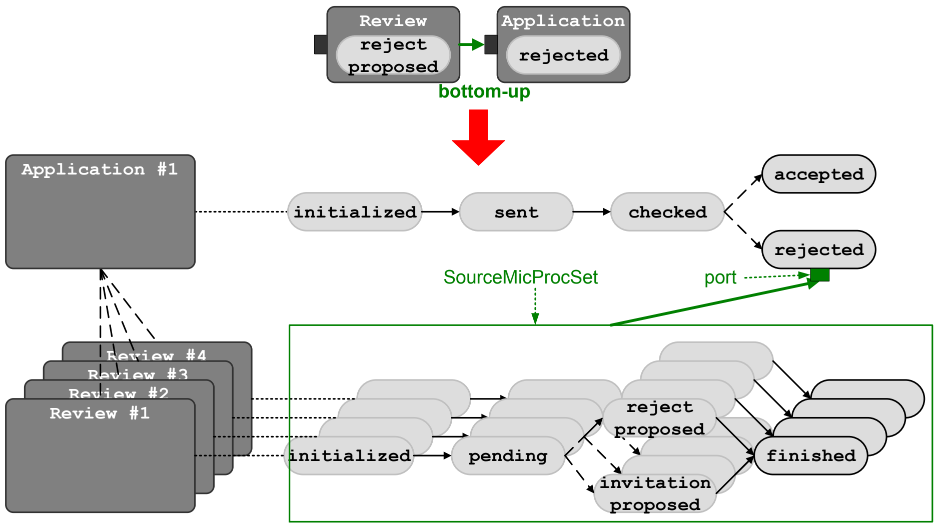


Figure 11.9: Aggregation instance

11.3 Transverse

When using *transverse relationships*, two different sets of micro process instances must be coordinated in the context of a higher-level one. For example, the execution of an `interview` micro process instance may depend on the one of all `review` micro process instances referencing the same `application` micro process instance. This means, coordination is not based on direct or indirect relationships but rather on a relationship to a common higher-level instance. For specifying such a coordination, a corresponding *transverse type* can be used (cf. Fig. 11.10).

When defining a transverse type, for two object types (i.e., the one of the source and the one of the target macro step type), a *common higher-level object type* must be specified. The lower-level instances of the two object types are then coordinated in respect to the common higher-level one (cf. Def. 62). The latter may be referenced indirectly (i.e., transitively) by both object types. In this context, two paths of relation types must be specified – one originating from the object type of the source macro step type and one from the object type of the target macro step type to the common higher-level object type.

When using a transverse instance at run-time, all micro process instances corresponding to the respective source macro step type (cf. Fig. 11.11) are aggregated in respect to the referenced higher-level object instance. The same counters are used for specifying an aggregation predicate as in the context of aggregation types (cf. Def. 59).

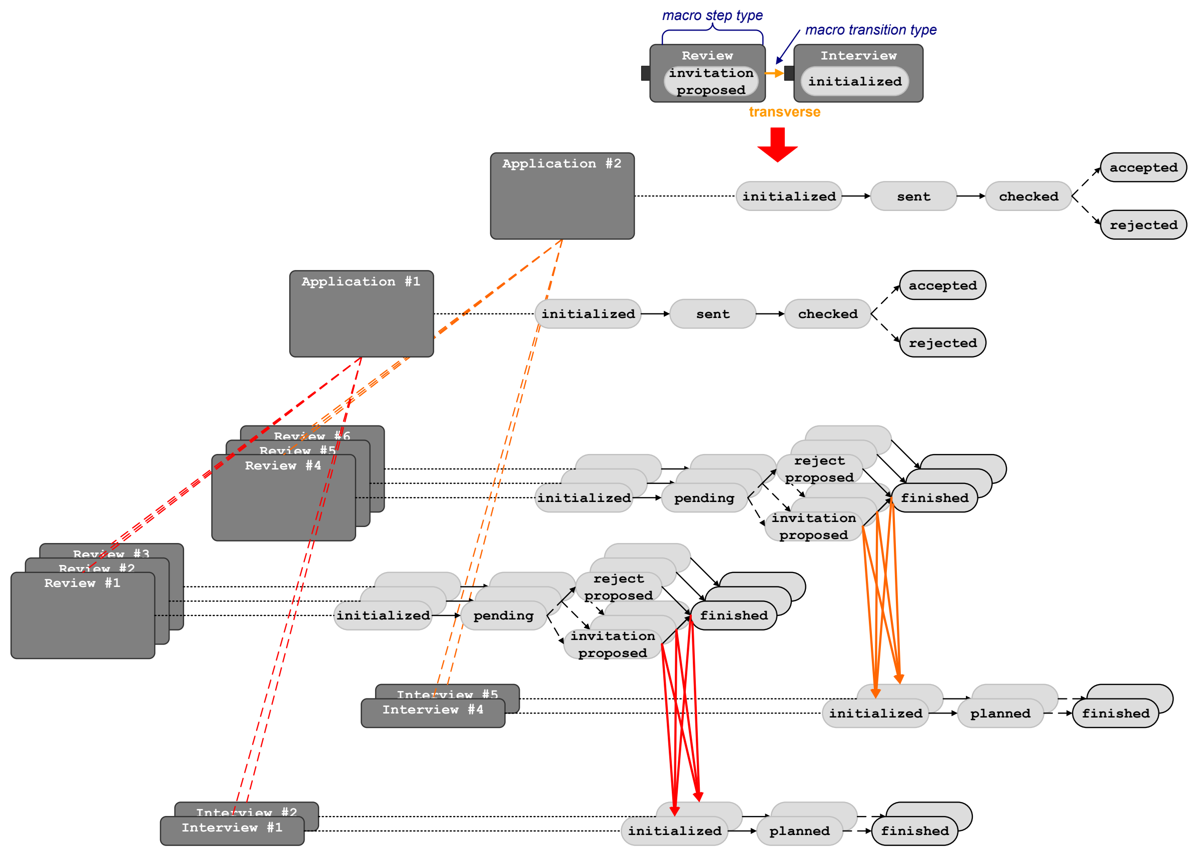


Figure 11.10: Transverse relationship

Example 11.7 (Common higher-level object type):
Consider the data model from Fig. 10.8. Common higher-level object types of `review` and `interview` are `application` and `job offer`. One of these object types must be chosen when defining a transverse type that coordinates the execution of `interview` micro process instances with the one of certain `review` micro process instances. Obviously, there is a difference in coordinating `interviews` and `reviews` belonging to the same `application` or `job offer`. In the latter case, the set of related instances is significantly higher since it contains all `reviews` and `interviews` corresponding to any `application` for a particular `job offer`.

Definition 62 (Transverse type):
Let dm = (name, OTypeSet, RelTypeSet) be a data model and macProcType = (name, oType, MacStepTypeSet, MacTransTypeSet) $\in$ MacProcTypes be a macro process type. Further, let macTransType = (s, t, pType) $\in$ MacTransTypeSet be a transverse macro transition type; i.e., transverse(s.oType, t.oType) = true. Then:

A **transverse type** is defined as tuple **transType = (pType, sourceMicProcType, sourceStateType, higherOT, transConstraint, SourcePathRelTypeSet, TargetPathRelTypeSet)** with the following properties:

- pType corresponds to the port type belonging to the state type of the target macro step type of macTransType.
- sourceMicProcType is the micro process type of the source macro step type; i.e.,
  sourceMicProcType.oType = s.oType.

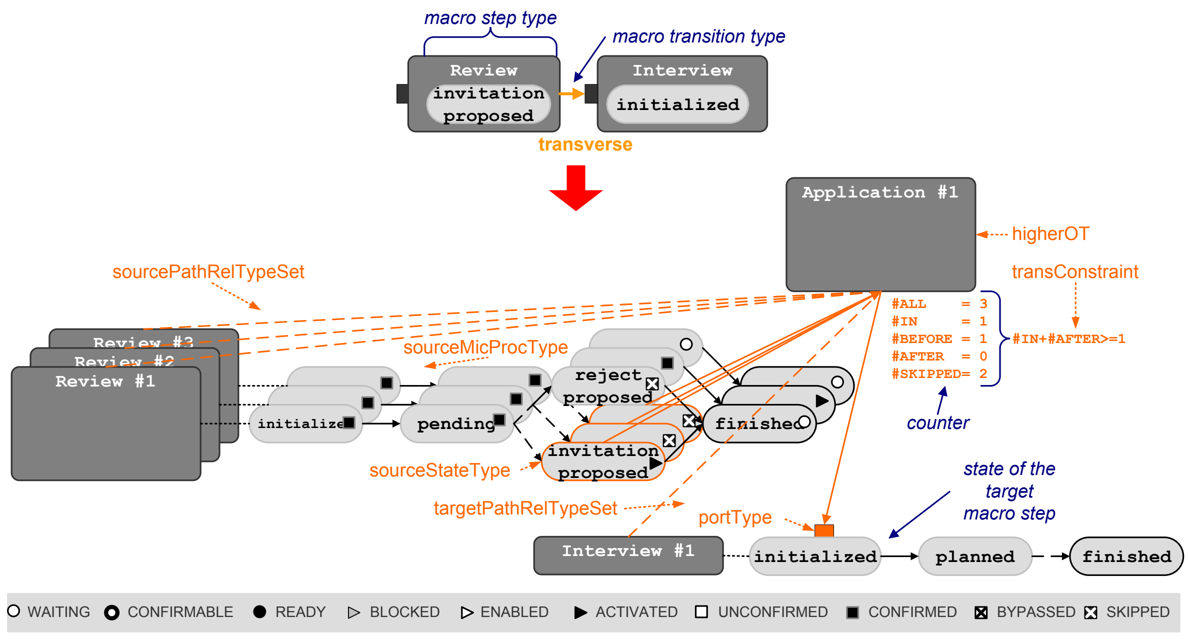


Figure 11.11: Transverse type

- sourceStateType = s.stateType is the state type based on which the micro process instances are aggregated at run-time.
- higherOT ∈ OTypes corresponds to an object type transitively referenced by the two object types referrenced by the source as well as target macro step type; i.e., higherOT ∈ higherlevel(s.oType) ∩ higherlevel(t.oType).
- transConstraint: MicProcInstances × sourceStateType ↦ BOOLEAN is a predicate based on the counters introduced in Def. 59 and simple artithmetic expressions.
- SourcePathRelTypeSet ⊆ RelTypeSet is a finite set of relation types describing a path from the lower-level object type of the source macro step type to the higher-level object type of the target macro step type; i.e.,
  path(s.oType,SourcePathRelTypeSet) = higherOT.
- TargetPathRelTypeSet ⊆ RelTypeSet is a finite set of relation types describing a path from the lower-level object type of the target macro step type to the common higher-level object type higherOT; i.e.:
  path(t.oType,TargetPathRelTypeSet) = higherOT.

TRANSTypes corresponds to the set of all definable transverse types. Further, **TRANSTypes$_{\text{portType}}$** denotes the set of all transverse types targeting at a particular port type.

Example 11.8 (Transverse type):
Consider Fig. 11.11. Since the macro transition type between the macro step type referring to the `review` object type in state `invitation proposed` and the one referring to the `interview` object type in state `initialized` is categorized as transverse, a transverse type must be defined. For this, as predicate `"#IN + #AFTER >= 1"` is used; i.e., an `interview` may be `initialized` if at least one corresponding `review` proposes to invite the candidate. Since counter #AFTER is considered, a `review` may be finished (state `finished` can be activated) even if the `interview` has not been `initialized` yet.

At run-time, for each micro process instance belonging to the target macro step type a transverse instance (cf. Def. 63) is created. It manages all micro process instances corresponding to the source macro step type and referring to same higher-level object instance (cf. Fig. 11.12).

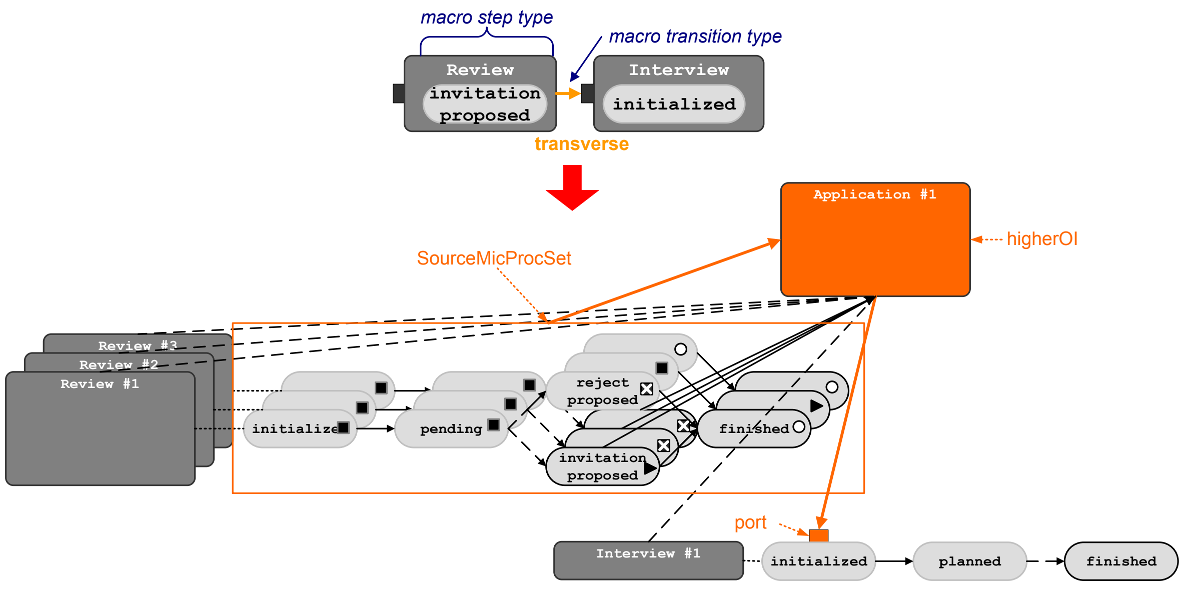


Figure 11.12: Transverse instance

Definition 63 (Transverse instance):
Let sourceMicProcInstance and targetMicProcInstance be two instances micro process types sourceMicProc and targetMicProc respectively. Then:

A **transverse instance** corresponds to a tuple **transInstance = (transType, port, higherOI, SourceMicProcSet)** where

- transType a transverse type.
- port is the instance of the portType corresponding to a state of micProcInstance:
  $\exists$ state $\in$ micProc.StateSet $\wedge$ port $\in$ state.sPortSet.
- higherOI is a common higher-level object instance with type transType.higherOT; i.e.,
  higherOI = referencedOI(targetMicProcInstance.oid, transType.TargetPathRelTypeSet).
- SourceMicProcSet comprises all micro process instances of sourceMicProcType referencing higherOI:
  $\forall$ sMicProcInstance $\in$ SourceMicProcSet:
  sMicProcInstance $\in$ micprocinstances(sourceMicProcType) $\wedge$
  higherOI = referencedOI(sMicProcInstance.oid, transType.SourcePathRelTypeSet).

TRANSInstances denotes the set of all transverse instances of any transverse type transType $\in$ TRANSTypes. Further, **TRANSInstances$_{\text{port}}$** corresponds to the set of all transverse instances targeting at the same port. Finallly, **transinstances: TRANSTypes** $\mapsto$ **2$^{\text{TRANSInstances}}$** assigns to each transverse type transType $\in$ TRANSTypes the set of corresponding transverse instances with transinstances(transType) $\subseteq$ TRANSInstances.

11.4 Summary

In summary, process context, aggregation, and transverse coordination components enable an advanced synchronization of interdependent micro process instances. In particular, these components allow for an asynchronous micro process execution and take the dynamically evolving set of micro process instances of a particular macro process type into account. This way, the emerging process structure is abstracted and hence becomes manageable at run-time. Since the definition of coordination components is based on a macro process type and modeled in a flat and comprehensible way, process designers are guided in deciding how many and which coordination components are required.

12

Macro Process Execution

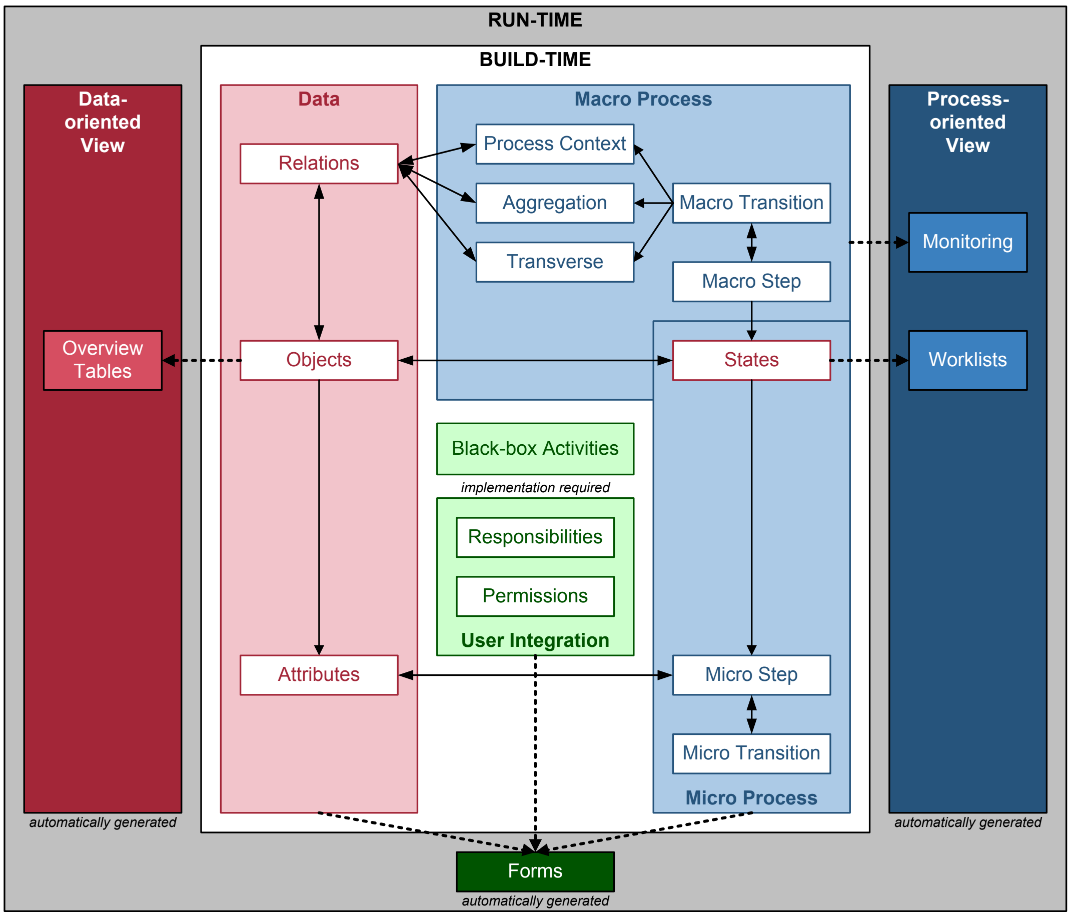


Figure 12.1: Macro process execution in PHILharmonicFlows

As discussed in the previous chapters, a macro process type provides an *abstract view* on a complex process structure. In particular, it provides transparency of the coordination components that are required to describe the various execution dependencies between the micro process types involved. More precisely, a macro process type reflects required process context, aggregation, and transverse types at a high level of abstraction. In turn, these coordination components define synchronization constraints between micro process instances based on their corresponding states (cf. Chapt. 11). Chapter 12 introduces a precise *operational semantics* for enacting macro process instances, which is based on concrete instances of process context, aggregation, and transverse types. For this purpose, we extend the operational semantics we introduced for micro process execution (cf. Chapt. 8). Thereby, we must be able to control whether a particular state of any micro process instance concerned in a process structure may be activated taking both the current progress of this micro process instance and its dependencies to other micro process instances into account. Note that this requires dynamic adaptions of worklists, overview tables, and user forms; i.e., which activities are mandatorily or optionally executable at a certain point in time not only depends on the state of the micro process instance itself, but also on the instances of the various coordination components and hence on the states of other micro process instances (cf. Chap. 11). The correct activation of these coordination components is a fundamental system feature which requires a well-defined operational semantics as well.

12.1 States as Interface between Micro and Macro Processes

For a particular macro process type, at run-time, the instances of its process context, aggregation, and transverse types control the coordination of the micro process instances involved. In particular, these coordination components are defined in terms of dependencies between different states of micro process instances (cf. Chapt. 11). Hence, *states act as interface between macro and micro process instances*. For this purpose, the port types associated with the steps of a macro process type are mapped to the particular micro process types (cf. Def. 64). More precisely, for each state type which is referred by at least one macro step type, one or more port types exist. Note that several macro transition types may target at the same port type. This way, it becomes possible to differentiate between AND- and OR-semantics (cf. Sect. 10.3). Further, for each incoming macro transition type of a port type, a respective coordination component is defined. Depending on the type of the incoming macro transition (i.e., top-down, bottom-up, or transverse), each port type refers to one or more coordination components; i.e., process context, aggregation, or transverse types.[1]

Definition 64 (Using state types as interface between macro and micro process types):
Let macProcType = (name, oType, MacStepTypeSet, MacTransTypeSet) be a macro process type. Further, let stateType = (name, sMicStepTypeSet) be a state type. Then:

A **port type** is a tuple **sPortType = (stateType, pPCTypeSet, pAGGTypeSet, pTRANSTypeSet)** with

- stateType $\in$ StateTypes corresponds to the state type the port type belongs to;
  i.e., $\exists$ macStepType $\in$ MacStepTypeSet with sPortType $\in$ macStepType.PortTypeSet.
- pPCTypeSet $\subseteq$ PCTypes corresponds to the finite set of process context types targeting at sPortType:
  $\forall$ pcType $\in$ pPCTypeSet: pcType.portType = sPortType.

[1] except the macro transition connects two macro step types referring to the same object type

- pAGGTypeSet $\subseteq$ AGGTypes corresponds to the finite set of aggregation types targeting at sPortType: $\forall$ aggType $\in$ pAGGTypeSet: aggType.portType = sPortType.
- pTRANSTypeSet $\subseteq$ TRANSTypes corresponds to the finite set of transverse types targeting at sPortType: $\forall$ transType $\in$ pTRANSTypeSet: transType.portType = sPortType.

Further, **sPortTypeSet$_{\textbf{stateType}}$** comprises all port types defined for stateType. Accordingly, the definition of a state type is extended to **stateType = (name, sMicStepTypeSet, sPortTypeSet)**.
Finally, **PortInstances** denotes the set of all port instances corresponding to any port type portType $\in$ PortTypes.

12.2 Macro Process Instances

Each macro process type is associated with a primary object type (cf. Def. 45). This way, it becomes possible to divide an emerging process structure into different subsets of micro process instances.[2] For example, for macro process type `recruitment`, several instances exist of which each belongs to a particular `job offer`; in Fig. 10.7, the micro process instances corresponding to a particular `job offer` are red-colored. Generally, each macro process instance refers to an instance of its primary object type. Further, it comprises all micro process instances directly or indirectly referencing this object instance (cf. Ex. 10.6).

Remember that for proper micro process execution, we introduced reaction, execution, and marking rules (cf. Chapt. 8). These rules have been defined based on well-defined markings. To properly coordinate the micro process instances of an emerging process structure, we introduce additional markings for process context, aggregation, and transverse instances as well as for the corresponding port instances. Based on these markings, we introduce additional reaction, execution and marking rules and extend existing ones. Like the markings we introduced for micro process instances, the ones for coordination components and ports are not only used to describe which components are currently activated[3], but also to identify the actual execution path of a macro process instance (i.e., the coordination components and ports actually used to activate certain states). Finally, the additional markings are applied to detect deadlock situations.

Regarding a process context (cf. Sect. 11.1), it can be derived whether or not one of its states can still be marked as ACTIVATED later on (i.e., at least one state of the process context is still marked as WAITING). In turn, if all states have been already marked as CONFIRMED or SKIPPED, the process context cannot be activated anymore. For this reason, PHILharmonicFlows distinguishes between *real deadlocks* and *waiting situations*. Regarding aggregation and transverse instances, however, this differentiation is not possible. Here, lower-level micro process instances provide information needed for continuing with the execution of a higher-level micro process instance. In this context, the concrete number of lower-level micro process instances may depend on user decisions as long as cardinality constraints are met. As a consequence, it is not possible to determine whether aggregations or transverse predicates can

[2]Note that these subset must not necessarily be disjoint.

[3]A process context instance becomes marked as ACTIVATED if one of its states belonging to the source micro process instances is ACTIVATED. In turn, aggregation and transverse instances become marked as ACTIVATED if there corresponding aggregation predicate evaluates to true.

evaluate to true in a later state. However, both situations (i.e., real deadlocks and waiting situations), must be detected and concepts to adequately handle them be provided (cf. Sect. 13.2).

At each point during macro process execution, all coordination components and ports are associated with a marking; i.e., not only the process context, aggregation, transverse and port instances currently activated. In the following, we describe their markings in order to allow for correct coordination of the micro process instances involved.

Example 12.1 (Markings of macro process instances):
To give a first impression of the rules introduced in the following, consider the `application` micro process instance from Fig. 12.2. State `accepted` comprises a port currently marked as ACTIVATED. In turn, this activation is due to the fact that all coordination components targeting at this port are marked as ACTIVATED as well. Hence, state `accepted` may be reached; i.e., the respective external explicit micro transition is marked as CONFIRMABLE. By contrast, state `rejected` has a port marked as WAITING, since all coordination components are marked as WAITING as well. Therefore, it is not possible to mark the incoming external micro transition as CONFIRMABLE at this stage.

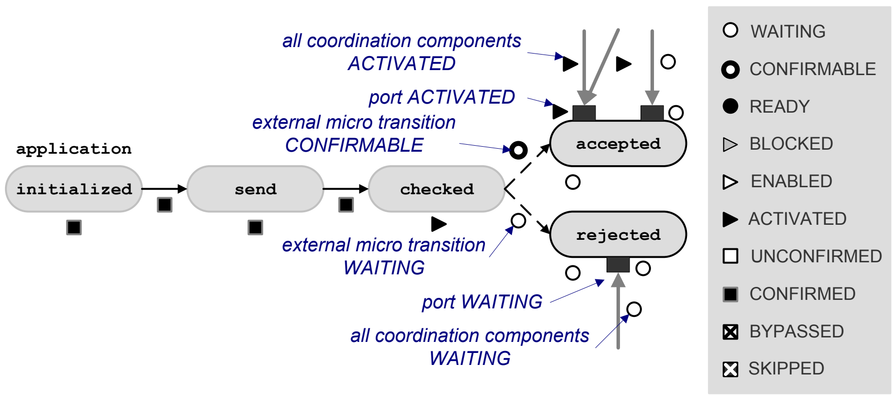


Figure 12.2: Markings for macro process execution

Definition 65 (Macro Process Instance):
A **macro process instance** is a tuple **macProcInstance = (macProc, oid, M_{Pc}, M_{Agg}, M_{Trans}, M_{Port})** with

- macProc is a macro process.
- oid is the identifier of the primary object instance macProcInstance belongs to; i.e., oid $\in$ oinstances(macProc.oType).
- **M_{Pc}: PCInstances $\mapsto$ PcMarkings** assigns to a process context instance pc its current marking M_{Pc}(pc) $\in$ PCMarkings. Thereby, we define **PcMarkings = {WAITING, ENABLED, ACTIVATED, CONFIRMED, SKIPPED, BLOCKED}**. The semantics of these markings is described in Tab. 12.1.
- **M_{Agg}: AGGInstances $\mapsto$ AggMarkings** assigns to an aggregation agg its current marking M_{Agg}(agg) $\in$ AGGMarkings. Thereby, we define **AggMarkings = {WAITING, ENABLED, ACTIVATED, CONFIRMED, SKIPPED}**. The semantics of these markings is described in Tab. 12.2.
- **M_{Trans}: TRANSInstances $\mapsto$ TransMarkings** assigns to a transverse tv its current marking M_{Trans}(tv) $\in$ TransMarkings. Thereby, we define **TransMarkings = {WAITING, ENABLED, ACTIVATED, CONFIRMED, SKIPPED}**. The semantics of these markings is described in Tab. 12.2.
- **M_{Port}: PortInstances $\mapsto$ PortMarkings** assigns to a port its current marking M_{Port}(port) $\in$ PortMarkings. Thereby, we define **PortMarkings = {WAITING, ENABLED, ACTIVATED, CONFIRMED, SKIPPED, BLOCKED}**. The semantics of these markings is described in Tab. 12.3.

Fig. 12.3 summarizes the markings of port, process context, aggregation, and transverse instances and their possible transitions[4].

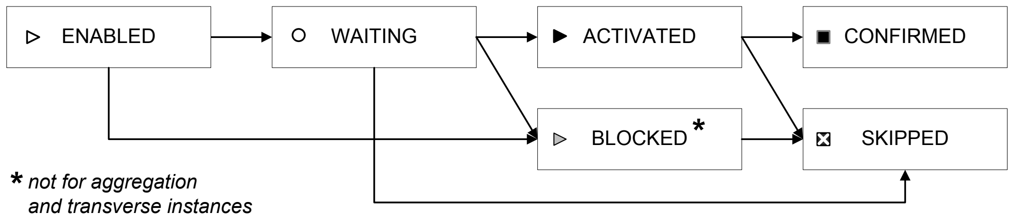


Figure 12.3: Markings of port, process context, aggregation, and transverse instances and marking transitions

12.2.1 Markings for Process Context Instances

A process context instance describes a dependency between two micro process instances of different type (cf. Def. 58); i.e., the activation of a state of a lower-level micro process instance depends on the currently activated state of a higher-level micro process instance. For this purpose, a process context refers to a number of states of the higher-level micro process. If one of these states is activated at run-time, for the lower-level micro process instance the respective state may be activated. At run-time, the following issues are relevant:

- Does the higher-level micro process instance, on which the execution of the lower-level one depends, already exist?
- Is one of the states of the higher-level micro process instance belonging to the process context that is currently activated?
- Is it still possible to activate one of the states defined by a process context later on?
- Was a given process context used to activate the respective state of the referred lower-level micro process instance?

Each process context instance has one of the following markings: WAITING, ENABLED, ACTIVATED, CONFIRMED, or BLOCKED (cf. Def. 65). These markings are illustrated in Fig. 12.3 and have the following meanings (cf. Tab. 12.1):

Marking	Label	Description
▷	ENABLED	The port (and corresponding state respectively) to which the process context refers *has not been activated* yet. In particular, no higher-level micro process instance is referenced when this marking is assigned.
○	WAITING	The port (and corresponding state respectively) to which the process context refers *has not been activated* yet. However, a higher-level micro process instance exists, but none of the states referenced by the process context is currently marked as ACTIVATED. Finally, it is still possible to activate one of these states later on.
▷	BLOCKED	The port (and corresponding state respectively) to which the process context refers *has not been activated* yet. No state of the higher-level micro process instance corresponding to the process context is currently ACTIVATED. Further, it is no longer possible to activate one of these states later on.

[4]Note that a backward jump or the deletion of a relation may lead to re-markings of involved instances. These transitions are not illustrated in Fig. 12.3 and not discussed in detail.

▶	ACTIVATED	The port (and corresponding state respectively) to which the process context refers *has not been activated* yet. In turn, the currently activated state of the higher-level micro process instance corresponds to one of the states of the process context.
■	CONFIRMED	The port (and corresponding state respectively) to which the process context refers *has been activated*; i.e., the process context (and corresponding port respectively) to activate the respective state of the lower-level micro process instance.
☒	SKIPPED	The process context (and corresponding port respectively) was not used to activate the respective state (i.e., another port belonging to the same state was used or the state was skipped due to a dead-path elimination). Finally, the process context does not become activated later on.

Table 12.1: Markings of process context instances

12.2.2 Markings for Aggregation and Transverse Instances

Opposed to process context instances which describe a dependency between exactly two micro process instances, aggregation (transverse) instances represent a dependency between a higher-level micro process instance and several lower-level (transverse) micro process instances. More precisely, the activation of a state of a higher-level transverse micro process instance depends on a predicate, which refers to the activated states of related lower-level (transverse) micro process instances. These predicates may use functions counting the corresponding lower-level (transverse) micro processes (cf. Sect. 11.2). If the respective predicate evaluates to true at run-time, for the corresponding higher-level (transverse) micro process instance the respective state may be activated.
Since the mentioned predicates depend on the actual number of currently created object instances and micro process instances respectively, it cannot be automatically decided whether or not a predicate will evaluate to true later on. Consequently, marking WAITING may be assigned even if it is not possible to ACTIVATE the aggregation later on. To detect deadlock situations in this context, marking WAITING is visualized when monitoring macro process instances (cf. Sect. 13.1). This way, users can detect micro process instances which cannot proceed since their execution depends on the execution of other micro process instances.

In summary, each aggregation (transverse) instance has one of the following markings (cf. Tab. 12.2 and Fig. 12.3): WAITING, ENABLED, ACTIVATED, CONFIRMED, or SKIPPED.

Marking	**Label**	**Description**
▷	ENABLED	The port (and corresponding state respectively) to which the aggregation (transverse) instance refers has not been activated yet. There are currently no lower-level micro process instances available.
○	WAITING	The port (and corresponding state respectively) to which the aggregation (transverse) refers has not been activated yet. There exists at least one lower-level micro process instance, but the predicate corresponding to the aggregation (transverse) instance currently *evaluates to false*.
▶	ACTIVATED	The port (and corresponding state respectively) to which the aggregation (transverse) refers has not been activated yet. The predicate corresponding to the aggregation (transverse) instance currently *evaluates to true*.
■	CONFIRMED	The port (and corresponding state respectively) to which the aggregation (transverse) instance refers was activated (or is currently activated); i.e., the aggregation (transverse) instance (and corresponding port respectively) was used to activate the respective state of the higher-level micro process instance.

☒	SKIPPED	The aggregation (transverse) instance (and its corresponding port respectively) was not used to activate the respective state (or the state was skipped due to a dead-path elimination). The aggregation (transverse) instance will not be activated later on even if the predicate evaluates to true (i.e., process execution has progressed).

Table 12.2: Markings of aggregation and transverse instances

12.2.3 Markings for Ports

To each state of a micro process instance, a number of ports may be assigned. In turn, each port comprises one or more coordination components; i.e., process context, aggregation, or transverse instances. A certain state may then only be ACTIVATED if at least one of its ports becomes ACTIVATED. In turn, to activate a port, all corresponding coordination components must be marked as ACTIVATED. In summary, depending on its incoming coordination components, each port has one of the following markings (cf. Tab. 12.3 and Fig. 12.3):WAITING, ENABLED, ACTIVATED, CONFIRMED, SKIPPED, or BLOCKED (cf. Def. 65).

Marking	**Label**	**Description**
▷	ENABLED	The port (and corresponding state respectively) has not been activated yet. Marking ENABLED indicates that the marking of the port must be re-evaluated due to a changed marking of a corresponding coordination component.
○	WAITING	The port (and corresponding state respectively) has not been activated yet. In particular, the port cannot be activated at this moment since at least one of its coordination component is currently marked as WAITING. However, it is still possible to activate the port later on; i.e., no coordination component referring to this port is currently marked as SKIPPED or BLOCKED.
▶	ACTIVATED	All coordination components are marked as ACTIVATED.
▷	BLOCKED	The port (and corresponding state respectively) has not been activated yet and at least one coordination component (i.e., process context) is currently marked as BLOCKED.
■	CONFIRMED	The port was previously ACTIVATED; i.e., it was used to activate the state it belongs to.
☒	SKIPPED	The port has not been activated and will also not become activated anymore; i.e., either another port of the state was used for activating the port or the state itself was SKIPPED.

Table 12.3: Markings of port instances

12.2.4 Additional Markings for Micro Process Instances

The current processing state of a macro process instance can be derived from the processing states of its corresponding micro process instances (cf. Fig. 12.4). More precisely, each macro process type refers to a primary object type (cf. Def. 45). Hence, each macro process instance refers to a particular object instance at run-time. In this context, note that not for each object instance a corresponding macro process instance exists (i.e., only for those object instances whose object type is primarily referred by a particular macro process type). This way, macro process instances can be uniquely identified (and all object instances and micro process instances respectively) belonging to the same macro process instance can be determined (cf. Chapt. 10.3); i.e., a macro process instance comprises all micro process instances whose object instance directly or indirectly references the primary object instance of the macro process instance. Further, note that a macro process type may also comprise macro step types referring to an object type that is assigned to a higher level than the primary object type.

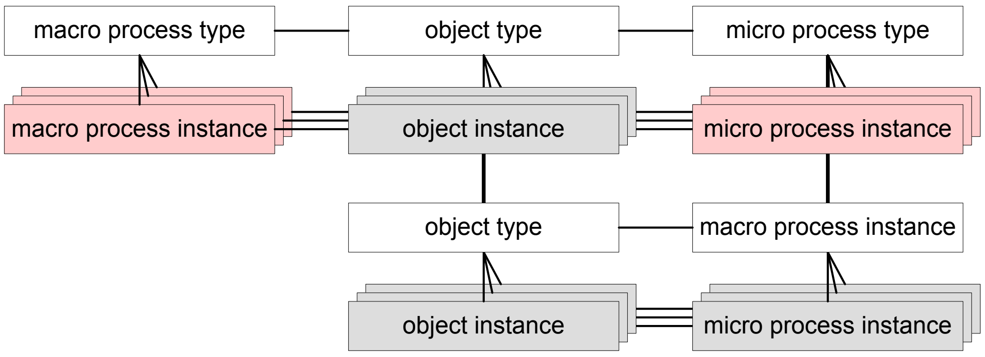


Figure 12.4: Correlation between macro and micro process instances

When executing a particular micro process instance, the activation of a subsequent state depends on its ports activated (if such ports exist)[5]. In particular, an external micro transition may only be marked as READY (implicit micro transition) or CONFIRMABLE (explicit micro transition) if at least one port of the state the target micro step belongs to is currently marked as ACTIVATED (or no port exists for this state). As long as it is not possible to activate the subsequent state, an external micro transition needs to be blocked. For this purpose, we introduce markings $\text{BLOCKED}_{\text{waiting}}$ and $\text{BLOCKED}_{\text{deadlocked}}$ for micro transitions (cf. Def. 66). While the former indicates that subsequent states may be activated later on, the latter expresses that process execution is blocked and user intervention becomes necessary.

Definition 66 (Micro transition markings extended for detecting deadlock situations):
Let micProcInstance be a micro process instance of type micProc; i.e., micProcInstance ∈ micprocinstances(micProc). Further, let micProcInstance.M_{MicTrans}: micProc.MicTransSet ↦ MicroTransMarkings be the function that assigns to each micTrans ∈ MicTransSet its current marking M_{MicTrans}(micTrans). Thereby, MicroTransMarkings = {WAITING, CONFIRMABLE, READY, ENABLED, ACTIVATED, UNCONFIRMED, CONFIRMED, BYPASSED, SKIPPED, $\text{BLOCKED}_{\text{act}}$}. Then we extend the set of micro transition markings as follows:

MicroTransMarkings := MicroTransMarkings ∪ {$\textbf{BLOCKED}_{\textbf{waiting}}$, $\textbf{BLOCKED}_{\textbf{deadlocked}}$}.
The semantics of these two additional markings is described in Tab. 12.4.

Marking	Label	Description
▷	$\text{BLOCKED}_{\text{waiting}}$	Since no port of the state the target micro step belongs to is currently marked as ACTIVATED, this state cannot be ACTIVATED; i.e., to mark the external micro transition either as READY (implicit micro transition) or CONFIRMABLE (explicit micro transition). However, at least one port of the state may be marked as ACTIVATED later on; i.e., not all ports are currently marked as BLOCKED.

[5]Note that ports do not exist for each state. Instead, a port only exist for the states referenced by a macro step type.

▷	BLOCKED$_{\text{deadlocked}}$	Since no port of the state the target micro step belongs to is currently marked as ACTIVATED, this state cannot be ACTIVATED; i.e., to mark the external micro transition either as READY (implicit micro transition) or CONFIRMABLE (explicit micro transition). Moreover, it is no longer possible to activate any port of the target state later on; i.e., all ports are currently marked as BLOCKED.

Table 12.4: External micro transition markings

For detecting situations in which the processing of a particular micro process instance is blocked, markings BLOCKED$_{\text{waiting}}$ and BLOCKED$_{\text{deadlocked}}$ are forwarded to the corresponding micro process instance. For this purpose, we introduce WAITING and BLOCKED as additional markings for micro process instances as a whole (cf. Def. 67).

Definition 67 (Micro process markings for detecting deadlock situations):
Let M_{MicProc}: MicProcInstances $\mapsto$ MicroProcessMarkings be the function assigning to a micro process instance micProcInstance its current marking M_{MicProc}(micProcInstance) $\in$ MicroProcessMarkings := {RUNNING, FINISHED} (cf. Def. 24). Then we extend this set of markings as follows:

MicroProcessMarkings := MicroProcessMarkings $\cup$ **{WAITING, BLOCKED}** (cf. Tab. 12.5).

Marking	Label	Description
○	WAITING	A micro process instance is marked as WAITING if there exists at least one external micro transition currently marked as BLOCKED$_{\text{waiting}}$ and no other external micro transition is currently marked as BLOCKED$_{\text{deadlock}}$ (i.e., deadlocks are of higher priority).
▷	BLOCKED	A micro process instance is marked as BLOCKED if there exists at least one external micro transition currently marked as BLOCKED$_{\text{deadlocked}}$.

Table 12.5: Micro process markings in the context of deadlocks

A particular micro process instance currently marked as BLOCKED$_{\text{waiting}}$ or BLOCKED$_{\text{deadlocked}}$, can be recognized based on the markings of the corresponding macro process instance. This is very important for monitoring issues (cf. Sect. 13.1). On one hand, deadlocks situations can be detected earlier. On the other, it becomes possible to skip all micro process instances of a particular macro process instance in one go.

12.3 Initializing of Coordination Components

The operational semantics for executing macro process instances can now be defined based on the markings introduced for process context, aggregation, transverse, and port instances. More precisely, we define a number of rules that control the interactions between the states of the different micro process instances corresponding to a macro process instance. Where required, we overwrite or extend existing rules, which we introduced when defining the operational semantics of micro process execution (cf. Chapt. 8). To illustrate this, the rule overview figures from Chapt. 8 are extended (e.g., see Fig. 12.5). All rules required for macro process execution are coloured, while already defined rules for micro process execution are grey.

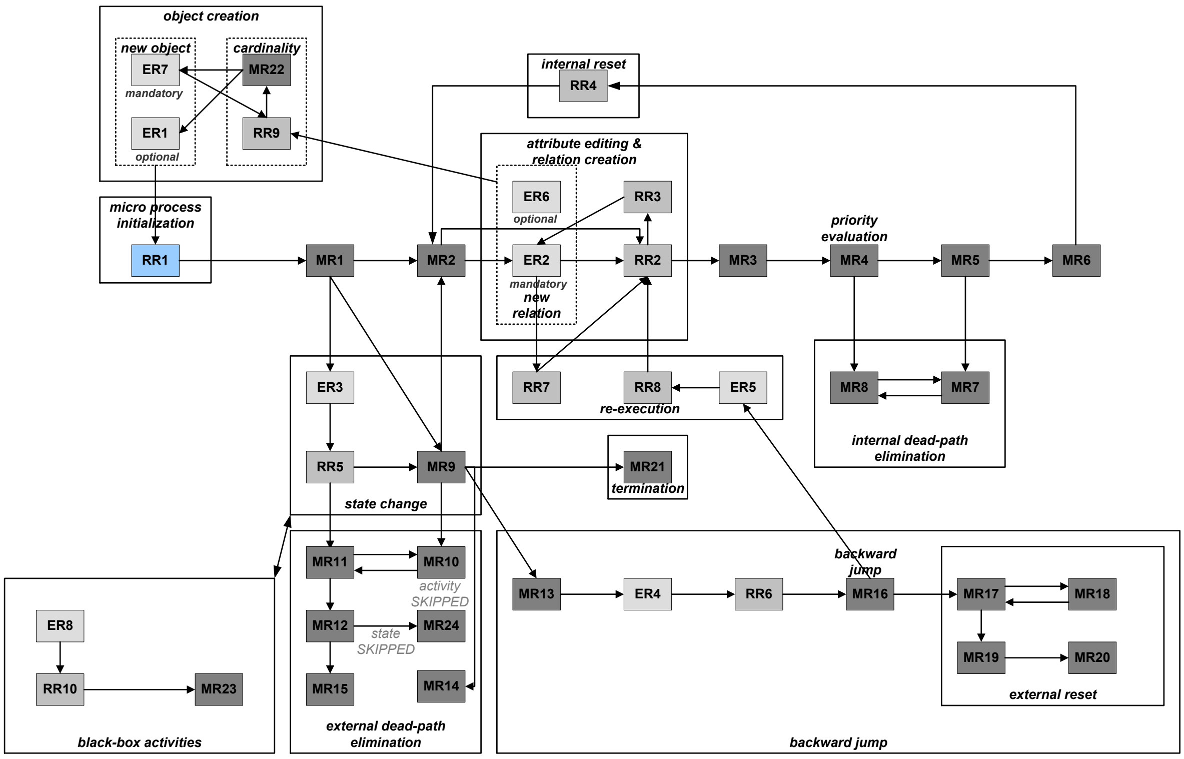


Figure 12.5: Rule for initializing coordination components

When creating an object instance, the corresponding micro process instance is automatically created (cf. Reaction Rule RR1). In this context, all ports (and coordination components targeting at these ports respectively) must be instantiated as well. Note that for a newly created object instance, no relations to other object instances exist. In this case, the object instance is created independently from other ones (cf. Sect. 9.1). For this reason, the particular coordination component instances are created, but without referencing the respective higher- or lower-level source micro process instances (cf. Def. 12.6a). More precisely, coordination component instances are created in respect to the micro process instance their port belongs to. The source micro process instance (in case of process context instances) or the set of source micro process instances (in case of aggregation and transverse instances) is assigned afterwards, when respective relations become available (cf. Def. 12.6b). For this reason, all ports and all corresponding coordination components are first marked as WAITING indicating that it is not possible to activate the respective states to which the ports belong to. To consider this, Reaction Rule RR1, which was introduced for initializing micro process instances, is extended (cf. Reaction Rule RR1””o-r).

When lower-level object instances are required in the context of particular higher-level ones (i.e., a minimal cardinality is defined), it is possible to create new object instances directly in the context of a higher-level object instance (cf. Sect. 9.1.2). In this case, the relation to the higher-level object instance is automatically assigned and all corresponding coordination components are updated; i.e., source micro process instances are assigned (cf. Sect. 12.4).[6]

[6]Note that the creation of an object instance also depends on the state currently activated for a particular higher-

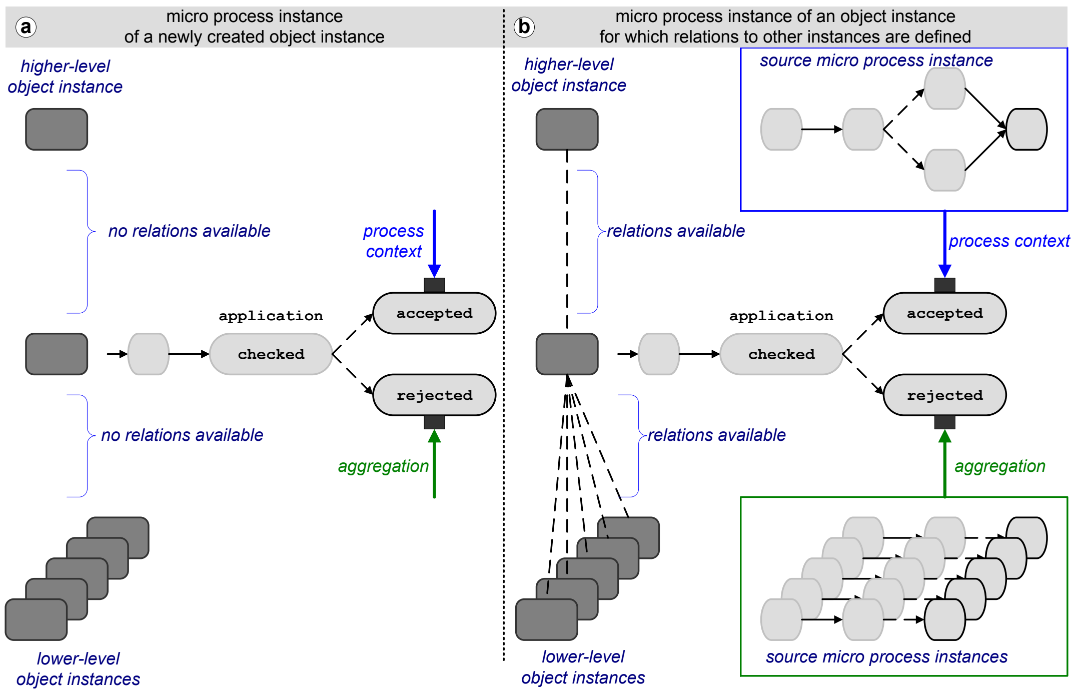

a
micro process instance of a newly created object instance
higher-level object instance
no relations available
process context
application
checked
accepted
rejected
no relations available
aggregation
lower-level object instances
b
micro process instance of an object instance for which relations to other instances are defined
higher-level object instance
source micro process instance
relations available
process context
application
checked
accepted
rejected
relations available
aggregation
lower-level object instances
source micro process instances

Figure 12.6: Initializating ports and coordination components

Reaction Rule (RR1””: Initializing coordination components and ports):
Let micProcInstance be an instance of micro process type micProc; i.e., micProcInstance ∈ micprocinstances(micProc). Further, let macProcInstance be an instance of a macro process type macProc; i.e., macProcInstance ∈ macprocinstances(macProc). Finally, let state ∈ micProc.StateSet be a state and port ∈ state.sPortSet be a port of this state. Then:

If a new object instance is created; i.e., OSet = OSet ∪ {o} o.oid = micProcInstance.oid, the initial markings of port, process context, aggregation, and transverse instances are defined as follows:

a)-j) see RR1 in Sect. 8.3

k) see RR1’ in Sect. 8.6

l) see RR1” in Sect. 9.1

m)-n) see RR1”’ in Sect. 9.2

o) ∀ pc ∈ port.pPCSet: macProcInstance.M_{Pc}(pc) := ENABLED;
i.e., all process context instances are initially marked as ENABLED.

p) ∀ agg ∈ port.pAGGSet: macProcInstance.M_{Agg}(agg) := ENABLED;
i.e., all aggregation instances are initially marked as ENABLED.

q) ∀ tv ∈ port.pTRANSSet: macProcInstance.M_{Trans}(tv) := ENABLED;
i.e., all tranverse instances are initially marked as ENABLED.

r) ∀ port ∈ sPortSet: macProcInstance.M_{Port}(port) := WAITING;
i.e., all port instances are initially marked as WAITING.

level instance; i.e., the creation of an object instance is not only restricted by the cardinalities defined, but also by process context types referring to start states. Details are discussed in Sect. 12.10

12.4 Composing Micro Process Instances

When a relation between two object instances is created, the micro process instance corresponding to the source object instance (as well as its her corresponding lower-level micro process instances) must be integrated in the process structure. Therefore, the respective coordination components must be adjusted; i.e., source micro process instances must be assigned to process context, aggregation, and transverse instances (cf. Sect. 12.3). For this purpose, we introduce *Reaction Rule RR11*, which is triggered when a new relation between two object instances is established (cf. Fig. 12.7). After assigning the source micro process instance(s) to a coordination component, their marking must be adapted (cf. Sect. 12.5). Therefore, all changed coordination components are first marked as ENABLED. The latter are then evaluated to check whether their marking must be changed (i.e., Marking Rules MR25, MR26, and MR27).

Transitive dependencies between micro process instances must be considered as well. For this reason, we differentiate between *direct* and *indirect* coordination components. Regarding the latter, coordination component instances referring to lower- and higher-level micro process instances must be updated accordingly. This is done by applying *Reaction Rule RR11*. Since the identification of these coordination component instances is far from being trivial, we discuss the different parts of this rule in detail.

As the focus of this thesis is on process modeling and execution, initially, we only consider coordination components not applied for any state activation so far; i.e., the state to which the port of the coordination component belongs has not been activated so far. Respective coordination components are either marked as CONFIRMED or SKIPPED. Otherwise, inconsistent process states might result necessitating additional exception handling mechanisms (cf. Sect. 12.10.2).

Reaction Rule (RR11: Updating coordination components):
Let dm $\in$ DM be a data model and ds = (dm, OSet, RelSet, M_{Cc}) be a corresponding data structure. Further, let $micProcInst_i$ be an instance of micro process type $micProc_i$, i=1,2; i.e., $micProcInst_i \in micprocinstances(micProc_i)$. Then:

When creating a relation instance rel = (relType, soid, toid), i.e., RelSet = RelSet $\cup$ {rel}, the following marking changes are made:

a) $\forall$ pcType $\in$ PCTypes with rel.relType $\in$ pcType.PathRelTypeSet:
   $\forall$ pcInst $\in$ pcinstances(pcType) $\wedge$ pcInst.port.state $\in$ $micProcInst_1$.StateSet with
   $micProcInst_1$.oid $\in$ lowerlevelOIs(rel.soid) $\cup$ {rel.soid} $\wedge$ M_{Pc}(pcInst) $\notin$ {CONFIRMED, SKIPPED}:
   $\exists$ $micProcInst_2$ with $micProcInst_2$.oid = referencedOI($micProcInst_1$.oid, pcType.PathRelTypeSet):

   - pcInst.sourceMicProcInstance := $micProcInst_2$
   - M_{Pc}(pcInst) := ENABLED

b) $\forall$ aggType $\in$ AGGTypes with rel.relType $\in$ aggType.PathRelTypeSet:
   $\forall$ aggInst $\in$ agginstances(aggType) $\wedge$ aggInst.port.state $\in$ $micProcInst_1$.StateSet with
   $micProcInst_1$.oid $\in$ higherlevelOIs(rel.toid) $\cup$ {rel.toid} $\wedge$ M_{Agg}(aggInst) $\notin$ {CONFIRMED, SKIPPED}:

   - $\forall$ s $\in$ aggInst.SourceMicProcInstancesSet: s := $micProcInst_2$
     with $micProcInst_1$.oid = referencedOI($micProcInst_2$.oid, aggType.PathRelTypeSet)
   - M_{Agg}(aggInst) := ENABLED

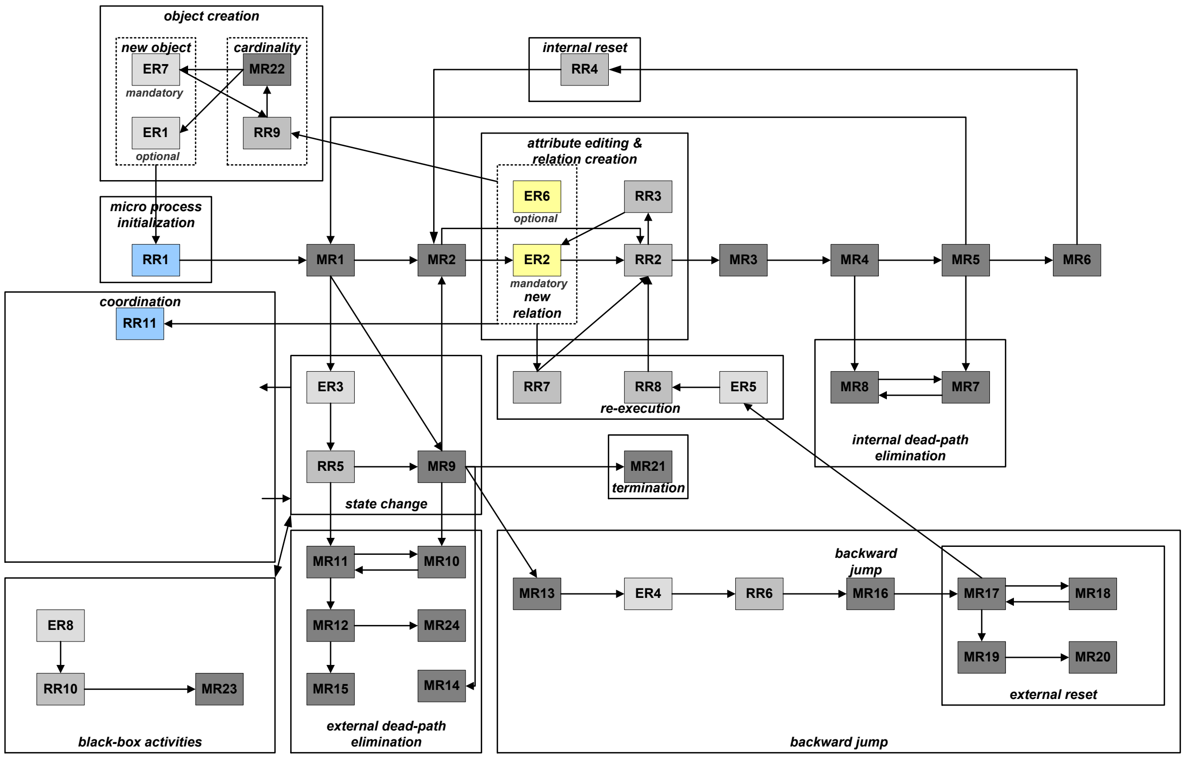


Figure 12.7: Rules for updating coordination components

c) $\forall$ trType $\in$ TransTypes with rel.relType $\in$ trType.TargetPathRelTypeSet:
$\forall$ trInst $\in$ transinstances(trType) $\wedge$ trInst.port.state $\in$ micProcInst$_1$.StateSet with micProcInst$_1$.oid $\in$ lowerlevelOIs(rel.soid) $\cup$ {rel.soid} $\wedge$ M_{Trans}(trInst) $\notin$ {CONFIRMED, SKIPPED}:

- trInst.higherOI := referencedOI(micProcInst$_1$.oid, trType.TargetPathRelTypeSet)
- $\forall$ s $\in$ trInst.SourceMicProcInstancesSet: s := micProcInst$_2$ with trInst.higherOI = referencedOI(micProcInst$_2$.oid, trType.SourcePathRelTypeSet)
- M_{Trans}(trInst) := ENABLED

d) $\forall$ trType $\in$ TransTypes with rel.relType $\in$ trType.SourcePathRelTypeSet:
$\forall$ trInst $\in$ transinstances(trType) $\wedge$ trInst.port.state $\in$ micProcInst$_1$.StateSet with trInst.higherOI $\in$ higherlevelOIs(rel.toid) $\cup$ {rel.toid} $\wedge$ M_{Trans}(trInst) $\notin$ {CONFIRMED, SKIPPED}:

- $\forall$ s $\in$ trInst.SourceMicProcInstancesSet: s := micProcInst$_2$ with trInst.higherOI = referencedOI(micProcInst$_2$.oid, trType.SourcePathRelTypeSet)
- M_{Trans}(trInst) := ENABLED

12.4.1 Updating Process Context Instances

At run-time, each process context instance belongs to exactly one micro process instance (cf. Fig. 11.6) or – to be more precise – to a port of a micro process instance state. When creating a new relation between a lower- and higher-level object instance, the micro process instance of the higher-level object instance must be assigned to the process context instances existing for the lower-level one (cf. Fig. 12.6a). We denote this as *direct process context instances*.

Example 12.2 (Direct process context instances):
Consider Fig. 12.8. Object instance `A` is processed by a respective micro process instance. The activation of state `a` of this instance depends on a process context instance. The source micro process instance of this process context instance corresponds to the micro process instance that belongs to an object instance `B`, which is referenced by object instance `A` as well. Hence, after creating the relation between object instances `A` and `B`, the micro process instance related to `B` is assigned to the process context of state `a`.

In general, transitive dependencies must be taken into account as well. More precisely, a newly added relation may belong to a path defined for a process context instance. Hence, the creation of the relation affects process context instances not directly belonging to the source or target object instances of the relation. We denote these process context instances as *indirect* ones (cf. Fig. 12.8). As a consequence, all process context instances belonging to lower-level object instances must be updated as well (cf. Reaction Rule RR11a).

Example 12.3 (Indirect process context instances):
Consider Fig. 12.8. The activation of state `c` depends on a process context instance. The source micro process instance of this process context instance corresponds to the micro process instance of object instance `B`. In turn, the latter is indirectly referenced via object instance `A`. Hence, when assigning the relation between object instances `A` and `B`, the micro process instance of `B` is assigned to all process context instances of object instance `C` which reference `A` transitively.

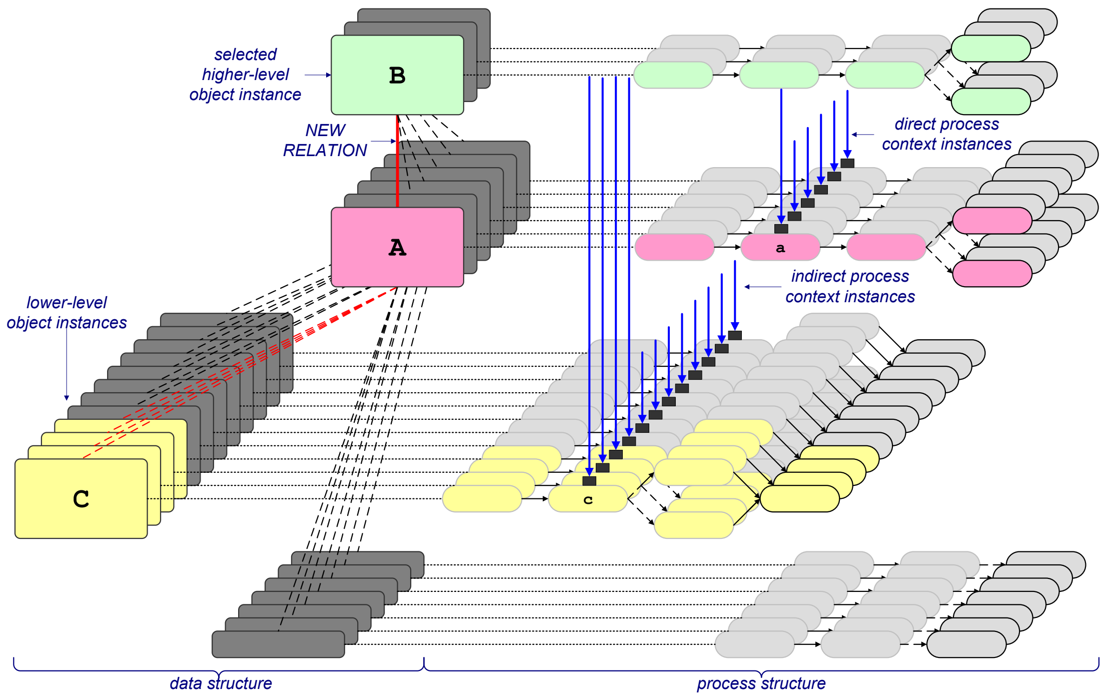


Figure 12.8: Updating process context instances

We take both direct and indirect process context instances into account when creating a new relation. Respective process context instances then must be determined as follows (cf. Reaction Rule RR11a):

1. Determine all process context types that refer to the type of the newly added relation (i.e., relType $\in$ PathRelTypeSet).
2. Determine all instances of these process context types that belong to the micro process instances related to the lower-level object instance of the added relation or to one of its lower-level instances. However, only process context instances that have not yet been distributed to any state activation are considered; i.e., process context instances currently marked as CONFIRMED or SKIPPED are not considered.
3. If a higher-level object instance of the source object instance can be determined using the defined path of relations (i.e., PathRelTypeSet), the corresponding micro process instance is assigned as source micro process instance to the respective process context instance. Further, the latter is re-marked as ENABLED. This indicates that the process context instance must be re-marked afterwards.

12.4.2 Updating Aggregation Instances

At run-time, each aggregation instance belongs to a port of a particular micro process instance state (cf. Fig. 11.9). When creating a relation between two object instances, the micro process instance of the lower-level object instance needs to be assigned to the set of source micro process instances (i.e., SourceMicProcInstancesSet) of the aggregation existing for the referenced higher-level object instance (cf. Fig. 12.6). We denote this as *direct aggregation instances*. A direct aggregation instance directly belongs to the micro process instance of the target object instance.

Example 12.4 (Direct aggregation instances):
Consider Fig. 12.9. The activation of state `a` depends on an aggregation instance. All micro process instances related to instances of object type `C`, which reference object instance `A`, belong to the set of source micro process instances of this aggregation instance. Hence, after creating a relation between object instances `C` and `A`, the micro process instance related to `C` is assigned to the set of source micro process instances of the aggregation instance of state `a`.

Again, transitive dependencies must be taken into account. More precisely, a newly created relation may belong to a path of a particular aggregation instance; i.e., the source and target object instances of the relation are not directly involved, but belong to a path of relations based on which the set of source micro process instances of an aggregation instance is determined. We denote the latter as *indirect aggregation instances* (cf. Fig. 12.9). As a consequence, all aggregation instances belonging to higher-level object instances of the object instance the newly created relation targets on must be updated as well (cf. Reaction Rule RR11b); i.e., their set of source micro process instances must be updated.

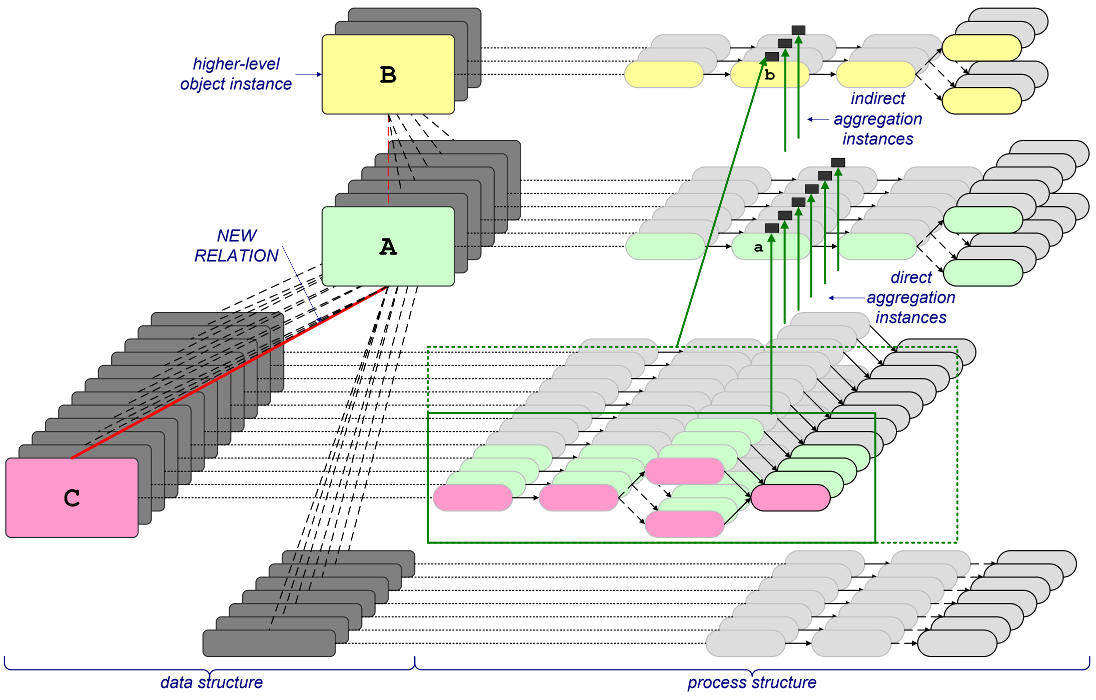

higher-level object instance
B
b
indirect aggregation instances
NEW RELATION
A
a
direct aggregation instances
C
data structure
process structure

Figure 12.9: Updating aggregation instances

Example 12.5 (Indirect aggregation instances):
Consider Fig. 12.9. The activation of state `b` depends on an aggregation instance. All micro process instances belonging to an instance of object type `C`, which indirectly references object instance `B` are assigned to the set of source micro process instances of this aggregation instance. Hence, after creating the relation between object instances `C` and `A`, the micro process instance related to `C` is assigned to the set of source micro process instance of the respective aggregation instance. The latter belongs to the micro process instance of the indirectly referenced object instance `B`.

We take both direct and indirect aggregation instances into account when creating a relation. Respective aggregation instances are determined as follows (cf. Reaction Rule RR11b):

1. Determine all aggregation types containing the relation type of the newly created relation; i.e., relType $\in$ PathRelTypeSet.
2. Determine all instances of these aggregation types that belong either to micro process instance of the higher-level object instance itself or to one of its higher-level instances referenced. However, only aggregation instances applied to any state activation are considered; i.e., aggregation instances currently marked as CONFIRMED or SKIPPED are not taken into account.
3. For each of these aggregation instances, its corresponding set of source micro process instances becomes updated; i.e., all micro process instances belonging to an object instance directly or indirectly referencing the object instances the aggregation instance be-

longs to are assigned as source micro process instances. Finally, the latter is re-marked as ENABLED, which indicates that it must be re-marked afterwards.

12.4.3 Updating Transvers Instances

Transverse instances comprise top-down as well as bottom-up dependencies in one component. On one hand, they refer to a higher-level object instance, which may be referenced transitively as well; on the other, a set of source micro process instances is aggregated in respect to this higher-level object instance. These micro process instances may also belong to object instances referencing the higher-level object instance transitively. As a consequence, both relation paths may be affected when a new relation is created. Thus, transverse instances must be updated in two steps (cf. Reaction Rules RR11c + d).

We consider Reaction Rule RR11c when creating a relation. The involved aggregation instances are then determined as follows:

1. Determine all transverse types containing the type of the newly created relation on the path defining the dependency between the object type the aggregation type belongs to and the common higher-level object type (i.e., relType $\in$ TargetPathRelTypeSet).
2. Determine all transverse instances of these types belonging to micro process instances corresponding to the source object instance of the relation or to one of its lower-level instances. Only transverse instances not yet applied to any state activation are considered; i.e., transverse instances currently marked as CONFIRMED or SKIPPED are not taken into account.
3. If a higher-level object instance can be determined based on the defined path of relations (i.e., TargetPathRelTypeSet), it is assigned as higher-level object instance to the respective transverse instance. Following this, the set of source micro process instances becomes updated (using SourcePathRelTypeSet). In addition, the transverse instance is re-marked as ENABLED indicating that it must be re-evaluated afterwards.

According to Reaction Rule RR11d, in turn, involved aggregation instances must be determined as follows when creating a relation:

1. Determine all transverse types that contain the type of the created relation on the path defining the dependency between the object type whose instance as aggregated at run-time and the common higher-level object type (i.e., relType $\in$ SourcePathRelTypeSet).
2. Determine all transverse instances of these types for which the common higher-level object instance corresponds to the relation's target object instance or to one of its higher-level instances. Only transverse instances not yet applied to any state activation are considered; i.e., transverse instances currently marked as CONFIRMED or SKIPPED are not taken into account.
3. For these transverse instances, the set of source micro process instances is updated (using SourcePathRelTypeSet). In addition, the transverse instance is re-marked as ENABLED indicating that it must be re-marked afterwards.

12.5 Re-marking Coordination Components

After updating a coordination component (i.e., after setting its source micro process instance or set of source micro process instances respectively), its marking must be updated as well; i.e., the marking of the coordination component must be re-set from ENABLED to ACTIVATED, WAITING, or BLOCKED.[7] For this purpose, we introduce *Marking Rules MR25, MR26, and MR27* (cf. Fig. 12.10).

In turn, when changing the marking of a coordination component, the port this coordination component belongs to must be re-marked as well. To indicate this, respective ports are re-marked as ENABLED. In turn, this marking triggers Marking Rule MR28 (cf. Sect. 12.6).

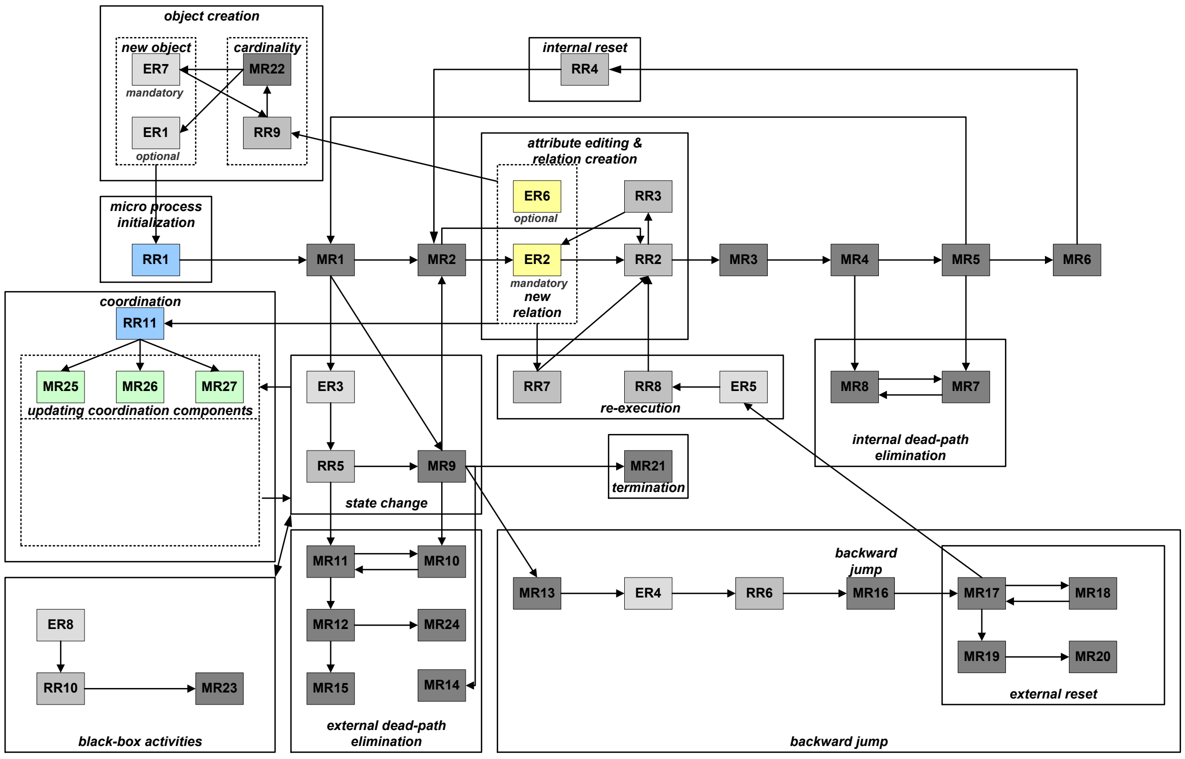


Figure 12.10: Rules for re-marking coordination components

12.5.1 Re-marking Process Context Instances

After assigning the source micro process instance to a particular process context instance, the marking of the latter is changed depending on the current state of the source micro process instance. According to *Marking Rule MR25*, the process context instance is re-marked as ACTIVATED, WAITING, or BLOCKED. In particular, if one of the states referred by the process context type corresponds to the current state of the source micro process instance, the respective process context instance is re-marked as ACTIVATED (cf. Fig. 12.11b). Otherwise, it is either marked as WAITING (at least one state the process context type refers to is still marked as WAITING) (cf. Fig. 12.11a) or as BLOCKED (cf. Fig. 12.11c).

[7] Note that marking BLOCKED is only applicable for process context instances.

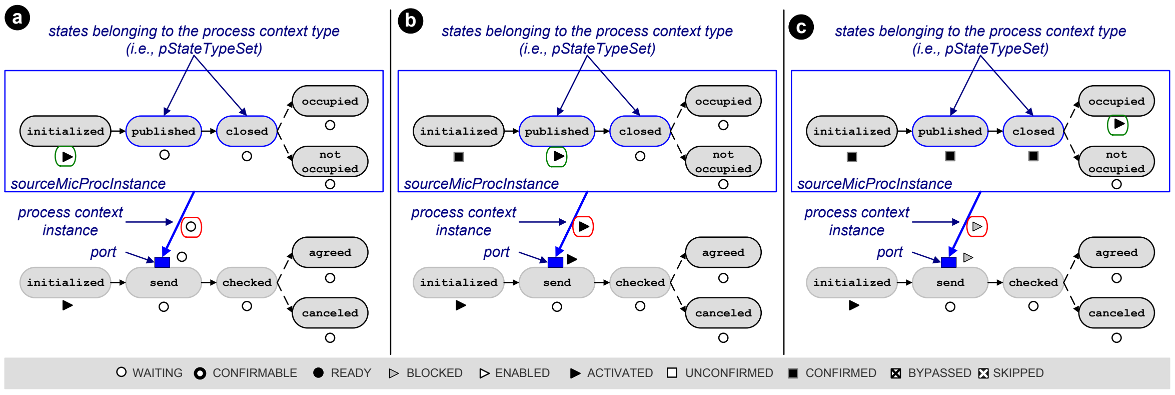


Figure 12.11: Applying Marking Rule MR25

Marking Rule (MR25: Re-marking process context instances):
Let pcInst = (pcType, port, sourceMicProcInstance) $\in$ PCInstances be an instance of process context type pcType = (portType, sourceMicProc, pStateTypeSet, PathRelTypeSet) $\in$ PCTypes. Then:

sourceMicProcInstance $\neq$ NULL $\wedge$ M_{Pc}(pcInst) = ENABLED, $\Rightarrow$

1. M_{Port}(port) := ENABLED

$$2.\ M_{Pc}(pcInst) := \begin{cases} \text{ACTIVATED}, & \text{if } \exists \text{ state} \in \text{sourceMicProc.StateSet with} \\ & \quad \text{state} \in \text{pStateTypeSet} \wedge M_{State}(\text{state}) = \text{ACTIVATED} \\ \text{WAITING}, & \text{if } \exists \text{ state} \in \text{sourceMicProc.StateSet with} \\ & \quad \text{state} \in \text{pStateTypeSet} \wedge M_{State}(\text{state}) = \text{WAITING} \\ \text{BLOCKED}, & \text{else} \end{cases}$$

12.5.2 Re-marking Aggregation and Transvserse Instances

After assigning a micro process instance to the set of source micro process instances of an aggregation instance, the corresponding predicate is re-evaluated and the aggregation instance is re-marked. According to *Marking Rule MR26*, an aggregation instance either is re-marked as ACTIVATED or WAITING. ACTIVATED is chosen if the aggregation predicate evaluates to true (cf. Fig. 12.12b). Otherwise, marking WAITING will be assigned (cf. Fig. 12.12a).

Marking Rule (MR26: Re-marking aggregation instances):
Let aggInst = (aggType, port, SourceMicProcInstancesSet) $\in$ AGGInstances be an instance of aggregation type aggType = (portType, sourceMicProcType, sourceStateType, aggConstraint, PathRelTypeSet) $\in$ AGGTypes. Then:

SourceMicProcInstancesSet $\neq \emptyset$ $\wedge$ M_{Agg}(aggInst) = ENABLED, $\Rightarrow$

1. M_{Port}(port) := ENABLED

$$2.\ M_{Agg}(aggInst) := \begin{cases} \text{ACTIVATED}, & \text{if aggConstraint(SourceMicProcInstancesSet) = TRUE} \\ \text{WAITING}, & \text{else} \end{cases}$$

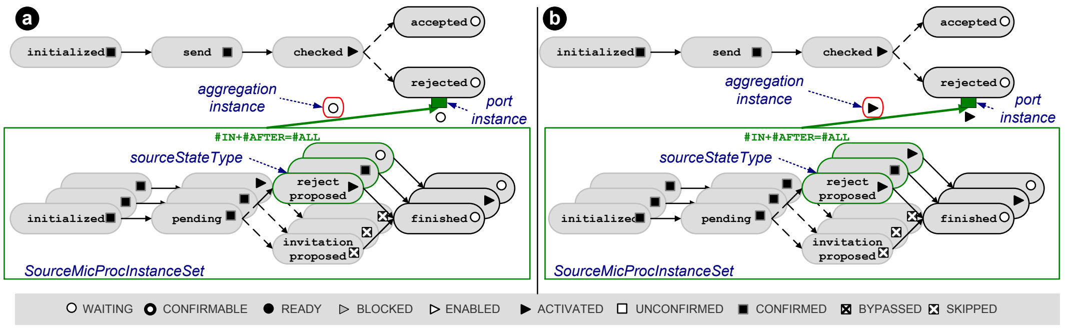


Figure 12.12: Applying Marking Rule MR26

The same applies to transverse instances (cf. *Marking Rule MR27*).

Marking Rule (MR27: Re-marking transverse instances):
Let trInst = (transType, port, higherOI, SourceMicProcInstancesSet) $\in$ TRANSInstances be an instance of transverse type trType = (portType, sourceMicProcType, sourceStateType, higherOT, transConstraint, SourcePathRelTypeSet, TargetPathRelTypeSet) $\in$ TRANSTypes. Then:

$M_{\text{Trans}}(\text{trInst}) = \text{ENABLED} \wedge \text{higherOI} \neq \text{NULL} \wedge \text{SourceMicProcInstancesSet} \neq \emptyset, \Rightarrow$

1. $M_{\text{Port}}(\text{port}) := \text{ENABLED}$
2. $M_{\text{Trans}}(\text{trInst}) := \begin{cases} \text{ACTIVATED}, & \text{if transConstraint(SourceMicProcInstancesSet) = TRUE} \\ \text{WAITING}, & \text{else} \end{cases}$

12.6 Re-marking Ports

Whether or not a particular port must be re-marked depends on the markings of its coordination components. More precisely, after re-marking a coordination component, the marking of the referred port must be updated as well. For this purpose, the port will be re-marked as ENABLED (cf. Marking Rules MR25, MR26, and MR27). In turn, this triggers *Marking Rule MR28* (cf. Fig. 12.13), which re-marks the respective port from ENABLED to WAITING, ACTIVATED, or BLOCKED.

According to Marking Rule MR28, a port is marked as ACTIVATED if its coordination components are all marked as ACTIVATED (cf. Fig. 12.14a). However, if at least one process context is marked as BLOCKED, the port itself is marked as BLOCKED (cf. Fig. 12.14b). Finally, if all coordination components are either marked as ACTIVATED or WAITING, the port becomes marked as WAITING (cf. Fig. 12.14c).

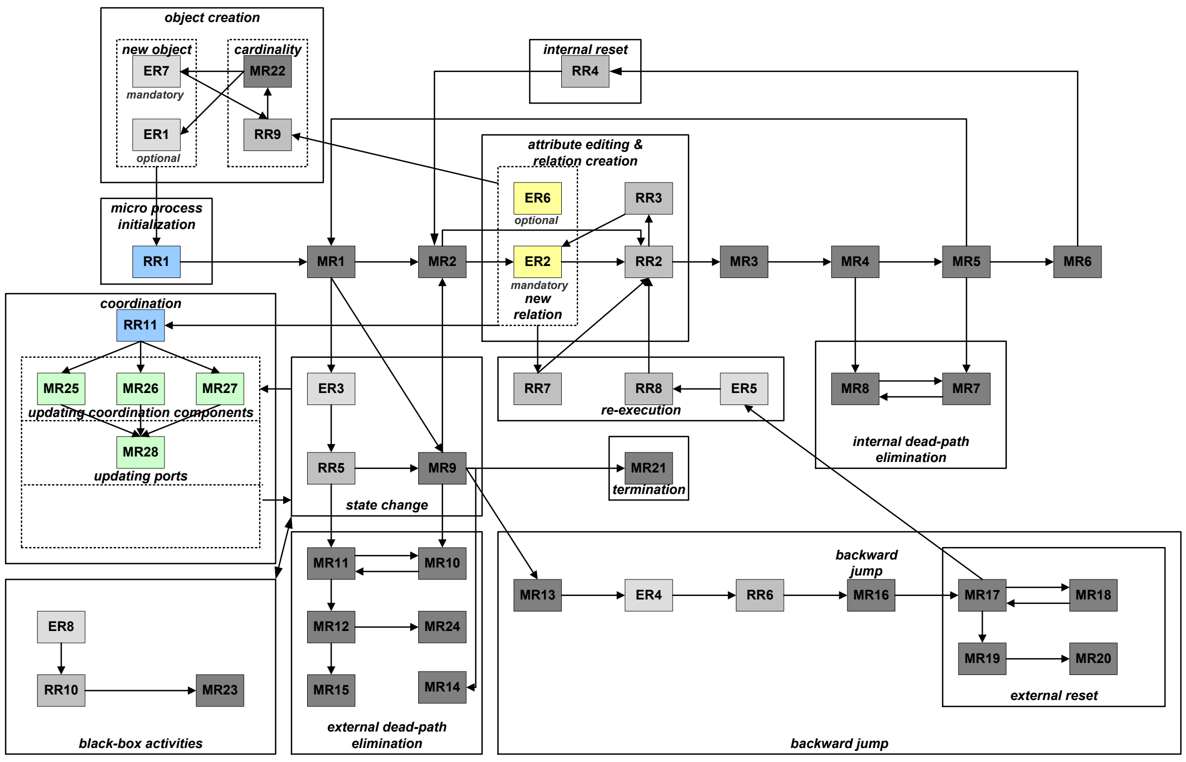


Figure 12.13: Rules for re-marking ports

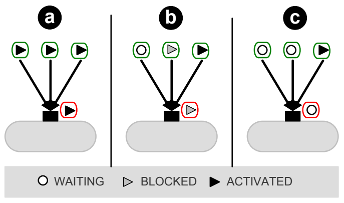


Figure 12.14: Applying Marking Rule MR28

Marking Rule (MR28: Re-marking ports):
Let port = (state, pPCSet, pAGGSet, pTRANSSet) ∈ Ports be a port. Then:

When re-marking at least one of the coordination components referring to port, the latter becomes re-marked as ENABLED indicating that its marking must be re-evaluated as follows:

$$M_{Port}(port) = \text{ENABLED}, \Rightarrow M_{Port}(port) := \begin{cases} \text{ACTIVATED}, & \forall\, pc \in pPCSet\text{: } M_{Pc}(pc) = \text{ACTIVATED} \wedge \\ & \forall\, agg \in pAGGSet\text{: } M_{Agg}(agg) = \text{ACTIVATED} \wedge \\ & \forall\, tv \in pTRANSSet\text{: } M_{Trans}(tv) = \text{ACTIVATED} \\ \text{BLOCKED}, & \exists\, pc \in pPCSet\text{: } M_{Pc}(pc) = \text{BLOCKED} \\ \text{WAITING}, & \text{else} \end{cases}$$

12.7 Executing Micro Process Instances

When executing a collection of inter-related micro process instances, corresponding coordination components must be considered; i.e., the activation of a state depends on its activated ports (if existing). Whether or not a subsequent state of a particular micro process instance may be activated is controlled by Marking Rule MR1. The latter re-marks a micro transition as READY (implicit transition) or as CONFIRMABLE (explicit transition). Thus, for correctly executing macro process instances, taking the various coordination components into account, Marking Rule MR1 (cf. Sect. 8.4) must be extended. In particular, an external micro transition may only be marked as READY or CONFIRMABLE, if at least one port of the state the target micro step belongs to is currently marked as ACTIVATED (or no port exists for this state) (cf. Marking Rule MR1""f). In order to block external micro transitions as long as the subsequent state cannot be activated, $\text{BLOCKED}_{\text{waiting}}$ and $\text{BLOCKED}_{\text{deadlocked}}$ are introduced as additional markings for micro transitions (cf. Def. 66). Note that in addition to these markings, marking $\text{BLOCKED}_{\text{act}}$ indicates that not all required black-box activities for leaving the source state have been executed. This differentiation is important to inform end-users about activities to be executed next.

As long as no port of the state the target micro step belongs is marked as ACTIVATED, all outgoing external micro transitions are marked as $\text{BLOCKED}_{\text{waiting}}$ or $\text{BLOCKED}_{\text{deadlocked}}$ (cf. Marking Rule MR1""f). The latter marking is assigned if there exists a port marked as BLOCKED.[8]

Marking Rule (MR1"": Marking external micro transitions as BLOCKED):
Let micProcInstance be an instance of micro process type micProc; i.e., micProcInstance ∈ micprocinstances(micProc). Further, let s, t ∈ StateSet be two states. Then:

a) see MR1 in Sect. 8.4.1

b) see MR1' in Sect. 8.4.2

c) see MR1" in Sect. 8.5.2

d)+e) see MR1"' in Sect. 9.2.3

f) $\forall$ micStep ∈ s.sMicStepSet with $M_{\text{MicStep}}(\text{micStep}) = \text{UNCONFIRMED}$:
  $\forall$ micTrans ∈ outtrans(micStep) with isexternal(micTrans) = TRUE:

$$M_{\text{MicTrans}}(\text{micTrans}) := \begin{cases} \textbf{CONFIRMABLE}, \text{ if explicit(micTrans) = TRUE} \wedge \\ \quad M_{\text{Act}}(\text{act}) = \text{UNCONFIRMED } \forall \text{ act} \in \text{Activities}_s \wedge \\ \quad \text{micTrans.target} \in \text{t.sMicStepSet} \wedge \\ \quad (\exists \text{ port} \in \text{t.sPortSet with } M_{\text{Port}}(\text{port}) = \text{ACTIVATED} \vee \text{t.sPortSet} = \emptyset) \\ \textbf{READY}, \text{ if explicit(micTrans) = FALSE} \wedge \\ \quad M_{\text{Act}}(\text{act}) = \text{UNCONFIRMED } \forall \text{ act} \in \text{Activities}_s \wedge \\ \quad \text{micTrans.target} \in \text{t.sMicStepSet} \wedge \\ \quad (\exists \text{ port} \in \text{t.sPortSet with } M_{\text{Port}}(\text{port}) = \text{ACTIVATED} \vee \text{t.sPortSet} = \emptyset) \\ \textbf{BLOCKED}_{\textbf{act}}, \text{ if} \\ \quad \exists \text{ act} \in \text{Activities}_{\text{sourceState}}: (M_{\text{Act}}(\text{act}) = \text{READY} \vee M_{\text{Act}}(\text{act}) = \text{ACTIVATED}) \\ \textbf{BLOCKED}_{\textbf{deadlocked}}, \text{ if target} \in \text{t.sMicStepSet} \wedge \text{t.sPortSet} \neq \emptyset \wedge \\ \quad \exists \text{ p} \in \text{t.sPortSet}: M_{\text{Port}}(\text{p}) = \text{BLOCKED} \\ \textbf{BLOCKED}_{\textbf{waiting}}, \text{ if target} \in \text{t.sMicStepSet} \wedge \text{t.sPortSet} \neq \emptyset \wedge \\ \quad \forall \text{ p} \in \text{t.sPortSet}: M_{\text{Port}}(\text{p}) = \text{WAITING} \end{cases}$$

[8]Note that this rule also applies to all micro transitions originating from a value step.

Example 12.6 (Applying Marking Rule MR1”” to activate a state without ports):
Consider Fig. 12.15a. State `checked` is currently ACTIVATED. For the two states succeeding `checked` (i.e., `accepted` and `rejected`) no ports exist. Hence, external micro transitions targeting at these states become marked as CONFIRMABLE when their source micro step (belonging to state `checked`) is marked as UNCONFIRMED.

Example 12.7 (Applying Marking Rule MR1”” to activate a state with ports):
Consider Fig. 12.15b. State `checked` is currently ACTIVATED. For both states succeeding `checked` (i.e., states `accepted` and `rejected`) a port exist. Since both ports are currently marked as ACTIVATED, the external micro transitions targeting at these states immediately become marked as CONFIRMABLE when their source micro step (belonging to state `checked`) is marked as UNCONFIRMED.

Example 12.8 (Applying Marking Rule MR1”” to block external micro transitions):
Consider Fig. 12.15c. State `checked` is currently ACTIVATED. For the states succeeding `checked` (i.e., `accepted` and `rejected`) a port exist. The port belonging to state `accepted` is currently marked as WAITING. Thus, the external micro transition connecting states `checked` and `accepted` is re-marked as $\text{BLOCKED}_{\text{waiting}}$. Since the port belonging to state `rejected` is currently marked as BLOCKED, in turn, the external micro transition targeting at this state becomes re-marked as $\text{BLOCKED}_{\text{deadlocked}}$ when the source micro step belonging to state `checked` is marked as UNCONFIRMED.

While an external and explicit micro transition is blocked (i.e., it is marked as $\text{BLOCKED}_{\text{waiting}}$ or $\text{BLOCKED}_{\text{deadlocked}}$), the subsequent state must not be activated. Accordingly, the activity for committing the state change is automatically disabled (cf. Fig. 12.16).

Example 12.9 (Disabling committments):
Consider Fig. 12.16. State `accepted` of an `application` micro process instance may only be activated if all corresponding `interviews` propose accepting the candidate. Regarding the first `application` instance, this condition is met. Hence, the corresponding coordination component and its port respectively are re-marked as ACTIVATED. In addition, the affected micro transition becomes marked as CONFIRMABLE. This way, the commit button to activate state `accepted` becomes enabled. In turn, for the second `application` instance, the port of state `accepted` is still marked as WAITING. Hence, the micro transition is marked as $\text{BLOCKED}_{\text{waiting}}$ and the corresponding commit button is disabled.

If a port is later re-marked as ACTIVATED, incoming external micro transitions are re-marked as READY (implicit transitions) or CONFIRMABLE (explicit transitions). This is expressed by *Marking Rule MR29*, which allows continuing with the execution of the micro process instance; i.e., the subsequent state may now be activated.

Example 12.10 (Applying Marking Rule MR29):
Consider Fig. 12.15c. The external micro transition between states `checked` and `accepted` is currently marked as $\text{BLOCKED}_{\text{waiting}}$. If the port belonging to state `accepted` becomes re-marked as ACTIVATED (cf. Fig. 12.15d), however, this external micro transitions changes its marking from $\text{BLOCKED}_{\text{waiting}}$ to CONFIRMABLE as long as the source micro step is still marked as UNCONFIRMED.

Finally, when a port is re-marked from ACTIVATED to WAITING or BLOCKED, it is not possible to activate the subsequent state as long as no other port is currently marked as ACTIVATED. In

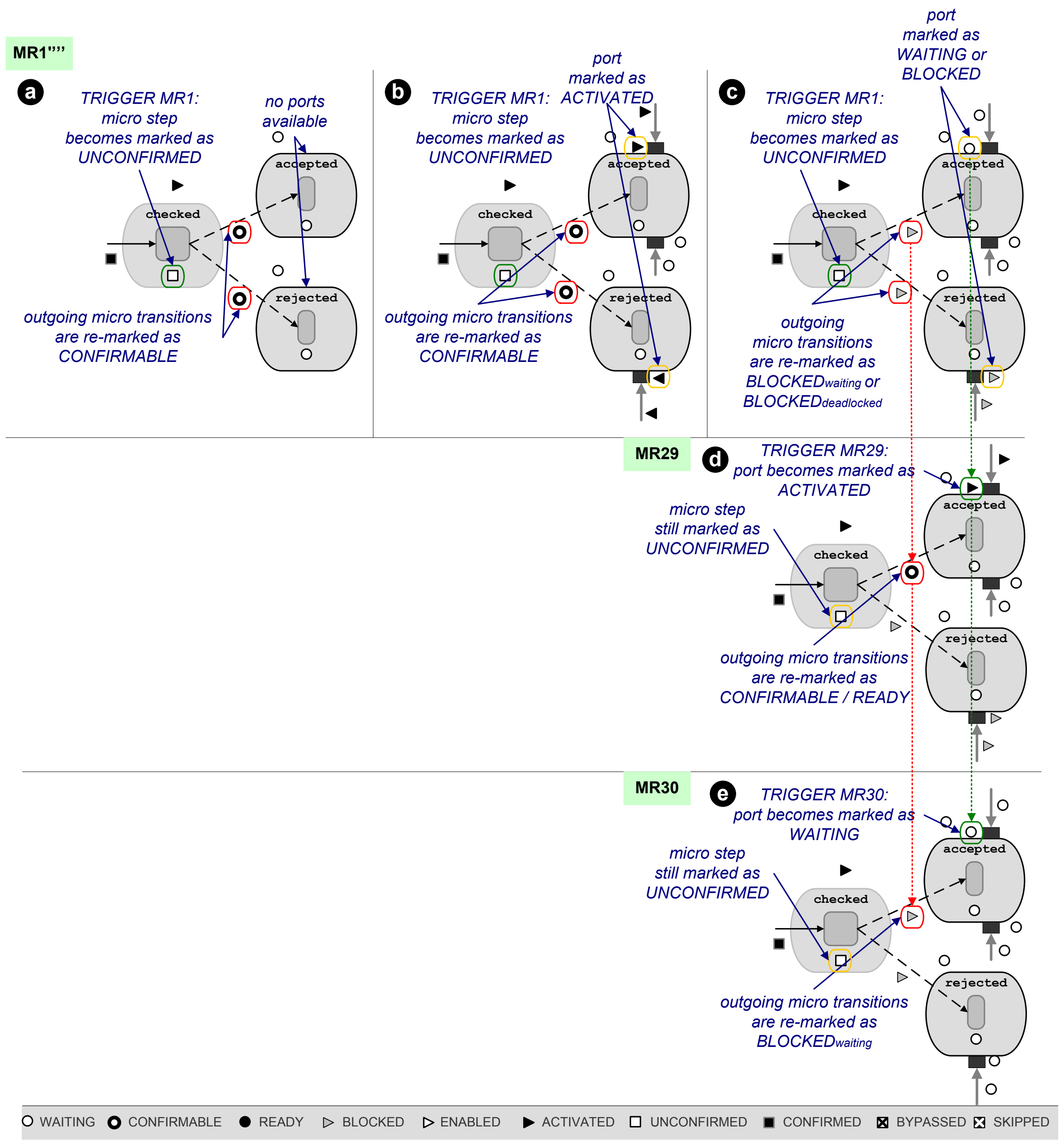
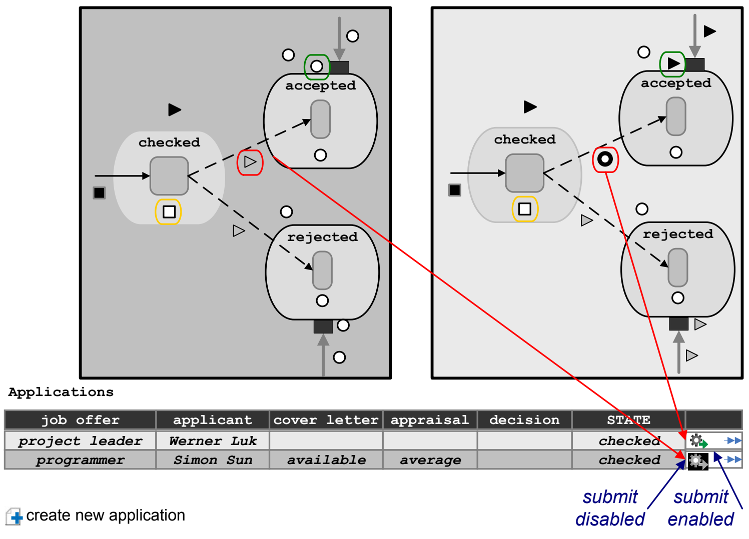

MR1""
a
TRIGGER MR1: micro step becomes marked as UNCONFIRMED
no ports available
outgoing micro transitions are re-marked as CONFIRMABLE
b
TRIGGER MR1: micro step becomes marked as UNCONFIRMED
port marked as ACTIVATED
outgoing micro transitions are re-marked as CONFIRMABLE
c
TRIGGER MR1: micro step becomes marked as UNCONFIRMED
port marked as WAITING or BLOCKED
outgoing micro transitions are re-marked as BLOCKEDwaiting or BLOCKEDdeadlocked
MR29
d
TRIGGER MR29: port becomes marked as ACTIVATED
micro step still marked as UNCONFIRMED
outgoing micro transitions are re-marked as CONFIRMABLE / READY
MR30
e
TRIGGER MR30: port becomes marked as WAITING
micro step still marked as UNCONFIRMED
outgoing micro transitions are re-marked as BLOCKEDwaiting
checked
accepted
rejected
WAITING
CONFIRMABLE
READY
BLOCKED
ENABLED
ACTIVATED
UNCONFIRMED
CONFIRMED
BYPASSED
SKIPPED

Figure 12.15: Applying Marking Rules MR1"", MR29, and MR30

this case, the external micro transitions targeting at this state are re-marked as BLOCKED$_{\text{WAITING}}$ or BLOCKED$_{\text{DEADLOCKED}}$ (cf. *Marking Rule MR30*). However, micro transition instances that have been already used for any state activation are not re-marked afterwards; i.e., micro transitions already marked as CONFIRMED or SKIPPED retain this marking.

Marking Rule (MR29: Unblocking external micro transitions):
Let micProcInstance be an instance of micro process type micProc; i.e., micProcInstance $\in$ micprocin-



Applications

job offer	applicant	cover letter	appraisal	decision	STATE	
project leader	*Werner Luk*				*checked*	
programmer	*Simon Sun*	*available*	*average*		*checked*	

create new application

Figure 12.16: Disbaling the committment of state transitions

stances(micProc). Further, let sourceState and targetState $\in$ micProc.StateSet be two states. Then:

$\exists$ port $\in$ targetState.sPortSet with M_{Port}(port) = ACTIVATED, $\Rightarrow$
$\forall$ mT $\in$ micProc.MicTransSet with mT.source $\in$ sourceState.sMicStepSet
$\wedge$ mT.target $\in$ targetState.sMicStepSet $\wedge$ $M_{MicTrans}$(mT) $\in$ {BLOCKED$_{WAITING}$, BLOCKED$_{DEADLOCKED}$}:

$$M_{MicTrans}(mT) := \begin{cases} \textbf{CONFIRMABLE}, & \text{if explicit(mT) = TRUE} \\ \textbf{READY}, & \text{else} \end{cases}$$

i.e., when re-marking a port as ACTIVATED, it becomes enabled to activate the subsequent state by marking implicit micro transitions as READY and explicit ones as CONFIRMABLE.

Example 12.11 (Applying Marking Rule MR30):
Consider Fig. 12.15d. The external micro transition between states `checked` and `accepted` is currently marked as CONFIRMABLE. When the port belonging to state `accepted` is re-marked as WAITING (cf. Fig. 12.15e), the external micro transition changes its marking from CONFIRMABLE to BLOCKED$_{waiting}$ (if the source micro step is still marked as UNCONFIRMED).

Marking Rule (MR30: Blocking external micro transitions):
Let micProcInstance be an instance of micro process type micProc; i.e., micProcInstance $\in$ micprocinstances(micProc). Further, let sourceState and targetState $\in$ micProc.StateSet be two states. Then:

targetState.sPortSet $\neq \emptyset \wedge \nexists$ port $\in$ targetState.sPortSet with M_{Port}(port) = ACTIVATED, $\Rightarrow$
$\forall$ mT $\in$ MicTransSet with mT.source $\in$ sourceState.sMicStepSet
$\wedge$ mT.target $\in$ targetState.sMicStepSet $\wedge$ $M_{MicTrans}$(mT) $\in$ {READY, CONFIRMABLE}:

$$M_{MicTrans}(mT) := \begin{cases} \textbf{BLOCKED}_{\textbf{deadlocked}}, \text{ if mT.target} \in \text{targetState.sMicStepSet} \wedge \\ \qquad \exists \text{ port} \in \text{targetState.sPortSet: } M_{Port}(\text{port}) = \text{BLOCKED} \\ \textbf{BLOCKED}_{\textbf{waiting}}, \text{ if mT.target} \in \text{targetState.sMicStepSet} \wedge \\ \qquad \forall \text{ port} \in \text{targetState.sPortSet: } M_{Port}(\text{port}) = \text{WAITING} \end{cases}$$

For detecting blocked micro process instances during run-time, markings $\text{BLOCKED}_{\text{WAITING}}$ or $\text{BLOCKED}_{\text{DEADLOCKED}}$ are propagated to the micro process instance level; i.e., the marking of the micro process instance the external micro transition instance belongs to becomes re-marked as well. According to *Marking Rule MR31*, if at least one micro transition of a micro process instance is marked as $\text{BLOCKED}_{\text{DEADLOCKED}}$ (or $\text{BLOCKED}_{\text{WAITING}}$), the micro process instance itself is re-marked as BLOCKED (or WAITING) (cf. Sect. 12.2).

Marking Rule (MR31: Propagating coordination information to micro process instances):
Let micProcInstance be an instance of micro process type micProc; i.e., micProcInstance $\in$ micprocinstances(micProc). Then:

$$M_{MicProc}(\text{micProcInstance}) := \begin{cases} \textbf{BLOCKED}, \text{if } \exists \text{ mT} \in \text{micProc.MicTransSet with } M_{MicTrans}(\text{mT}) = \text{BLOCKED}_{deadlocked} \\ \textbf{WAITING}, \text{if } \exists \text{ mT} \in \text{micProc.MicTransSet with } M_{MicTrans}(\text{mT}) = \text{BLOCKED}_{waiting} \\ \textbf{RUNNING}, \text{else} \end{cases}$$

Altogether, Fig. 12.17 illustrates the rules required for coordinating micro process instances. In particular, external micro transitions become marked as $\text{BLOCKED}_{\text{waiting}}$ or $\text{BLOCKED}_{\text{deadlocked}}$ using Marking Rules MR1, MR29, and MR30. Finally, respective markings are propagated to the micro process instance (cf. Marking Rule MR31).

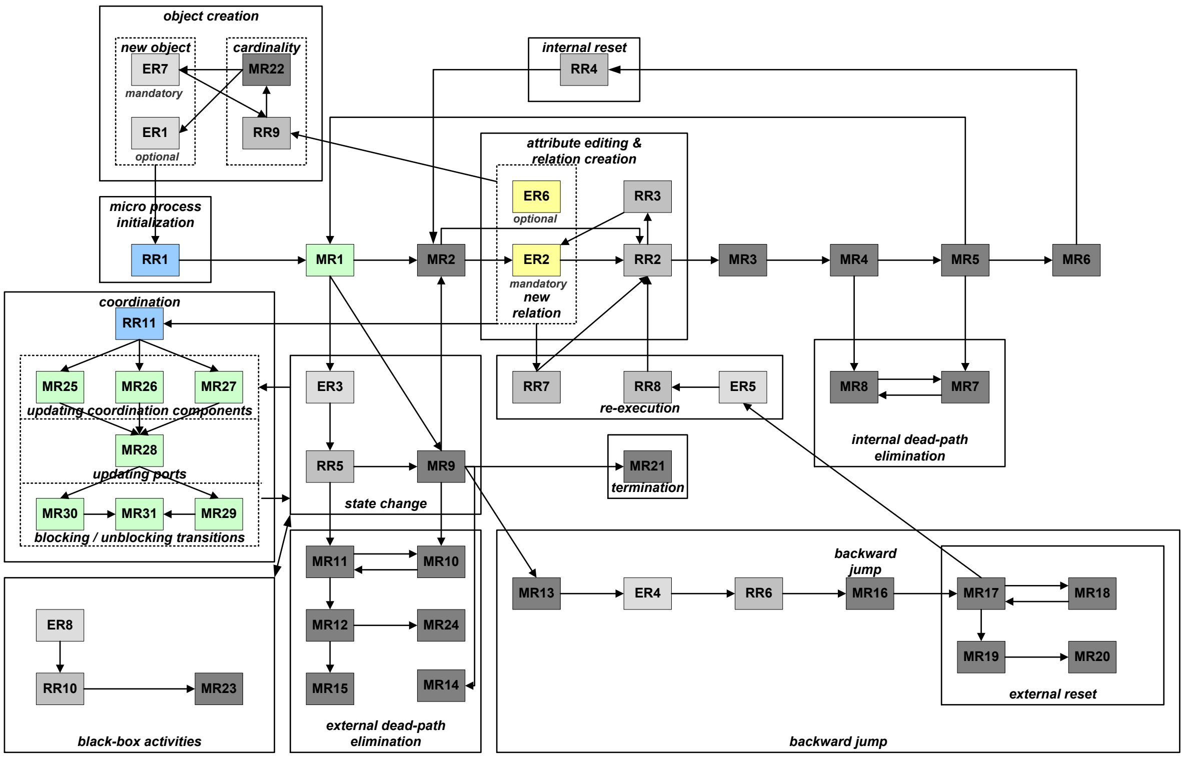


Figure 12.17: Rules for micro process coordination

12.8 State Changes

As discussed in Sect. 12.5, whether a particular coordination component may be activated during run-time depends on the activated state of the source micro process instance(s). Hence, the marking of respective coordination components must be updated when a state change occurs. For this purpose, as illustrated in Fig. 12.18, Marking Rule MR9' is extended to MR9".

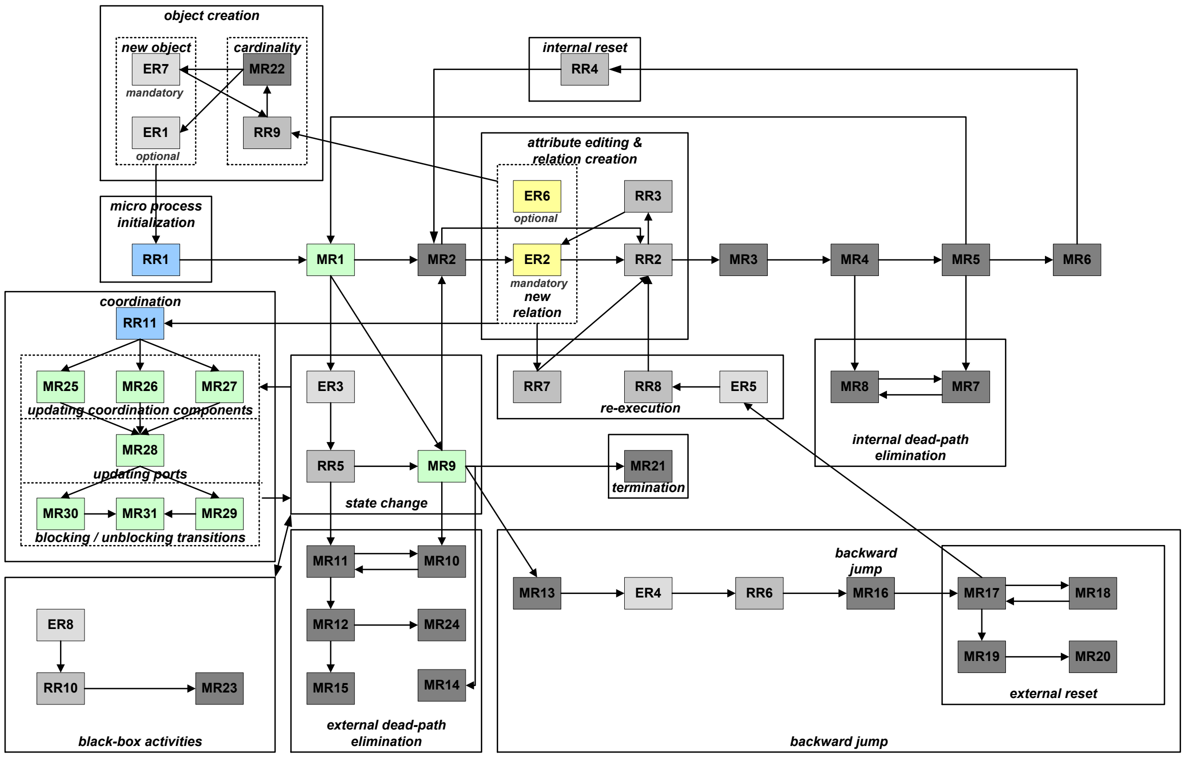


Figure 12.18: Rules for state change considering coordination components

When performing a state change, two cases must be differentiated (cf. Fig. 12.19). The first case refers to state changes that either occur during the execution of the source micro process instance (for process context instances) or a micro process instance belonging to the set of source micro process instances (for aggregation and transverse instances). Then, the marking of the respective coordination component, which is currently marked as ACTIVATED, WAITING, or BLOCKED, must be re-evaluated. For this reason, the respective coordination components are re-marked as ENABLED. As discussed in Sect. 12.4, this marking acts as trigger for Marking Rules MR25, MR26, and MR27, which are required to update the marking of the coordination components (cf. Sect. 12.5). Regarding process context instances (cf. Marking Rule MR9"j), the process context instance is re-marked as ACTIVATED (cf. Marking Rule MR25) if the newly activated state belongs to the states defined by the process context (i.e., pStateTypeSet in Def. 57). In turn, if the currently activated state does not belong to the defined state set, the process context is re-marked as BLOCKED or WAITING (cf. Marking Rule MR25). In the latter case, it is possible to activate the process context later on; i.e., at least one state referred by the corresponding process context type is still marked as WAITING.
Regarding aggregation and transverse instances, their marking is adjusted when a state change occurs during the execution of the source micro process instances (cf. Marking Rule MR9"k

and MR9”l). In this case, corresponding aggregation predicates are evaluated and the marking is adjusted accordingly (cf. Marking Rules MR26 and MR27).

The second case deals with state changes that occur during the execution of the micro process instance the port of the coordination component belongs to. The coordination components and ports used for activating a particular state must then be re-marked as CONFIRMED (cf. Marking Rule MR9”m). In turn, all coordination components (and ports respectively) not used for any state activation must be re-marked as SKIPPED (cf. Marking Rule MR9”n).

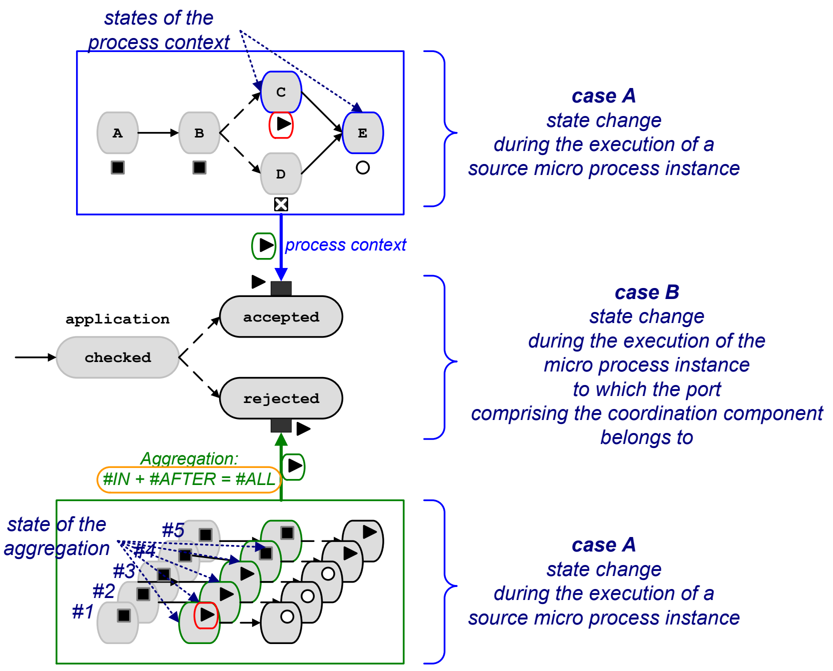


Figure 12.19: Updating coordination components when executing a state change

Marking Rule (MR9”: State Change considering coordination components):
Let micProcInstance be an instance of micro process type micProc; i.e., micProcInstance ∈ micprocinstances(micProc). Further, let $state_1$, $state_2$ ∈ StateSet be two states with $M_{State}(state_1)$ = ACTIVATED and $M_{State}(state_2)$ = WAITING. Finally, let micTrans ∈ MicTransSet be a micro transition with micTrans.source ∈ $state_1$.sMicStepSet and micTrans.target ∈ $state_2$.sMicStepSet (i.e., $state_2$ is a successor of $state_1$). Then:

$M_{MicTrans}$(micTrans) = READY ∧ isexternal(micTrans) = TRUE, ⇒

a-f) see MR9 in Sect. 8.5.1

g-i) see MR9’ in Sect. 9.2.3

j) ∀ pcType ∈ PCTypes with $state_1$ ∈ pcType.pStateTypeSet ∨ $state_2$ ∈ pcType.pStateTypeSet:
∀ pcInst ∈ pcinstances(pcType) with
pcInst.sourceMicProcInstance = micProcInstance ∧ M_{Pc}(pcInst) ∉ {CONFIRMED, SKIPPED}:
M_{Pc}(pcInst) := ENABLED;
i.e., each process context instance pcInst for which micProcInstance corresponds to pcInst.sourceMicProcInstance, must be re-evaluated and re-marked as ENABLED. In turn, this marking triggers Marking Rule MR25.

k) ∀ aggType ∈ AGGTypes with $state_1$ = aggType.sourceStateType ∨ $state_2$ = aggType.sourceStateType:
∀ aggInst ∈ agginstances(aggType) with
micProcInstance ∈ aggInst.SourceMicProcInstancesSet ∧ M_{Agg}(aggInst) ∉ {CONFIRMED, SKIPPED}:

M_{Agg}(aggInst) := ENABLED;
i.e., each aggregation instance aggInst, for which micProcInstance is contained in aggInst.SourceMicProcInstancesSet, must be re-marked as ENABLED. In turn, this marking triggers Marking Rule MR26.

l) $\forall$ transType $\in$ TransTypes with $state_1$ = transType.sourceStateType $\vee$ $state_2$ = transType.sourceStateType:
$\forall$ trInst $\in$ transinstances(transType) with
micProcInstance $\in$ trInst.SourceMicProcInstancesSet $\wedge$ M_{Trans}(trInst) $\notin$ {CONFIRMED, SKIPPED}:
M_{Trans}(trInst) := ENABLED;
i.e., each transverse instance trInst, for which micProcInstance is contained in trInst.SourceMicProcInstancesSet, must be re-marked as ENABLED. In turn, this marking triggers Marking Rule MR27.

m) $\forall$ port with port.state = $state_2$ $\wedge$ M_{Port}(port) = ACTIVATED:
M_{Port}(port) := CONFIRMED $\wedge$
$\forall$ pcInst $\in$ port.pPCSet: M_{Pc}(pcInst) := CONFIRMED $\wedge$
$\forall$ aggInst $\in$ port.pAGGSet: M_{Agg}(aggInst) := CONFIRMED $\wedge$
$\forall$ transInst $\in$ port.pTRANSSet: M_{Trans}(transInst) := CONFIRMED;
i.e., all ports and corresponding coordination components, which enable the activation of the respective state, are re-marked as CONFIRMED.

n) $\forall$ port = (state, pPCSet, pAGGSet, pTRANSSet) with port.state = $state_2$ $\wedge$ M_{Port}(port) $\neq$ ACTIVATED:
M_{Port}(port) := SKIPPED $\wedge$
$\forall$ port.pcInst $\in$ pPCSet: M_{Pc}(pcInst) := SKIPPED $\wedge$
$\forall$ port.aggInst $\in$ pAGGSet: M_{Agg}(aggInst) := SKIPPED $\wedge$
$\forall$ port.transInst $\in$ pTRANSSet: M_{Trans}(transInst) := SKIPPED;
i.e., all ports and corresponding coordination components, which are not used to activate the respective state, are re-marked as SKIPPED.

If the marking of a coordination component changes, the port this component belongs to may have to be re-marked as well (cf. Sect. 12.6). In particular, if all coordination components of a port are marked as ACTIVATED, the port itself is re-marked as ACTIVATED (cf. Marking Rule MR28). In turn, if at least one coordination component of a port is still marked as WAITING, the port itself will be marked as WAITING. The latter triggers Marking Rules MR29 and MR30, according to which the corresponding external micro transitions are blocked or unblocked (cf. Sect. 12.7).

Finally, when performing an external dead-path elimination (cf. Sect. 8.5), coordination components and ports respectively must be marked as SKIPPED as well. In particular, when the state they belong to is marked as SKIPPED, they are re-marked as SKIPPED as well (cf. *Marking Rule MR32* and Fig. 12.20).

Marking Rule (MR32: External dead-path elimination considering coordination):
Let micProcInstance be an instance of micro process type micProc; i.e., micProcInstance $\in$ micprocinstances(micProc). Further, let state $\in$ micProc.StateSet be a state. Then:

M_{State}(state) = SKIPPED, $\Rightarrow$

- $\forall$ $port_1$ $\in$ state.sPortSet: $M_{Port}(port_1)$:= SKIPPED $\wedge$
  $\forall$ $pcInst_1$ $\in$ $port_1$.pPCSet: $M_{Pc}(pcInst_1)$:= SKIPPED $\wedge$
  $\forall$ $aggInst_1$ $\in$ $port_1$.pAGGSet: $M_{Agg}(aggInst_1)$:= SKIPPED $\wedge$
  $\forall$ $transInst_1$ $\in$ $port_1$.pTRANSSet: $M_{Trans}(transInst_1)$:= SKIPPED
- $\forall$ pcType with state $\in$ pcType.pStateTypeSet:
  $\forall$ $pcInst_2$ $\in$ pcinstances(pcType) with $pcInst_2$.sourceMicProcInstance = micProcInstance:
  $M_{Pc}(pcInst_2)$:= ENABLED $\wedge$ $M_{Port}(pcInst_2.port)$:= ENABLED

- $\forall$ aggType with state = aggType.sourceStateType:
  $\forall$ aggInst$_2$ $\in$ agginstances(aggType) with micProcInstance $\in$ aggInst$_2$.SourceMicProcInstancesSet:
  M$_{\text{Agg}}$(aggInst$_2$) := ENABLED $\wedge$ M$_{\text{Port}}$(aggInst$_2$.port) := ENABLED
- $\forall$ transType with state = transType.sourceStateType:
  $\forall$ trInst$_2$ $\in$ transinstances(transType) with micProcInstance $\in$ trInst$_2$.SourceMicProcInstancesSet:
  M$_{\text{Trans}}$(trInst$_2$) := ENABLED $\wedge$ M$_{\text{Port}}$(trInst$_2$.port) := ENABLED

i.e., if a state becomes re-marked as SKIPPED, all ports and their coordination components are re-marked as skipped as well. In addition, all coordination components and ports depending on this state are then re-evaluated and re-marked as ENABLED.

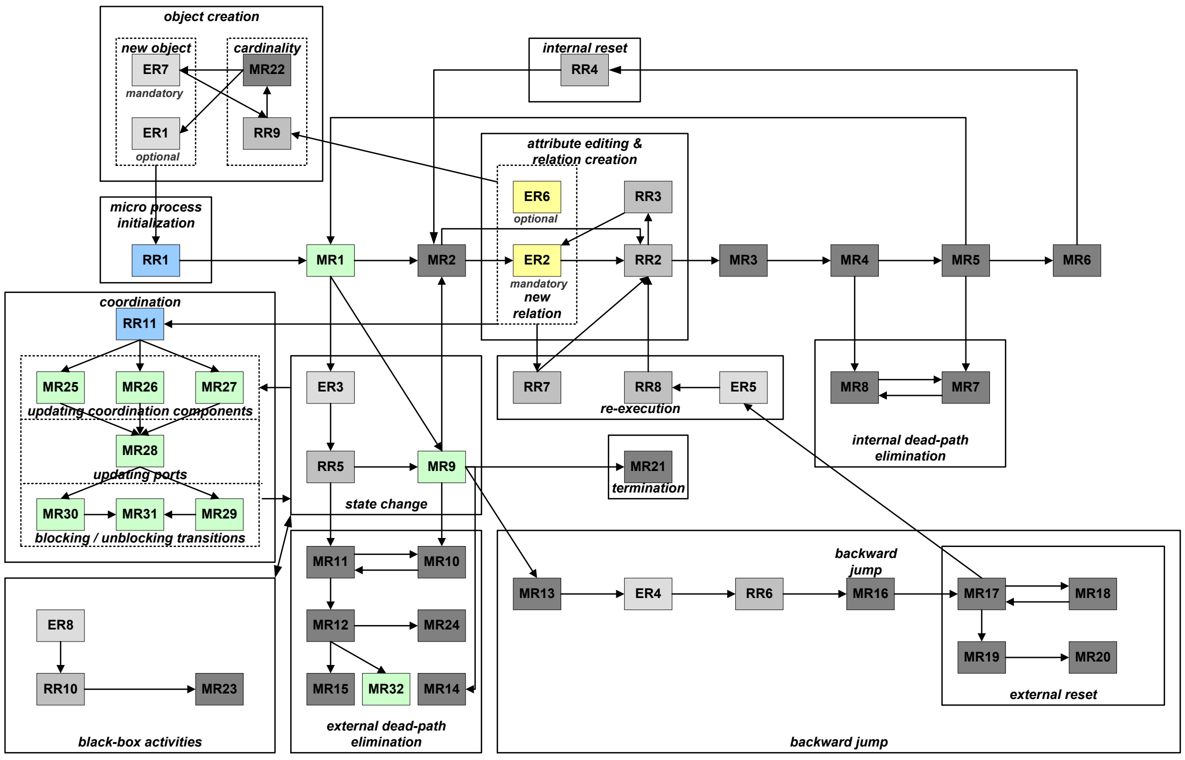


Figure 12.20: Rules for external dead-path elimination with coordination components

12.9 Terminating Macro Process Instances

A macro process instance will terminate if the micro process instance corresponding to its primary object instance reaches an end state. To ensure proper termination of all involved micro process instances, PHILharmonicFlows automatically re-marks them as BYPASSED. This indicates that they are no longer needed in respect to the execution of the macro process instance. Consequently, a macro process instance might not necessarily be sound at each point during its execution.

Each macro process instance refers to a primary object instance (cf. Def. 65) dividing the emerging process structure into different subsets of micro process instances (cf. Sect. 12.2).

The termination of a macro process instance, therefore, strongly depends on the termination of the micro process instance of its primary object instance. A micro process instance terminates when one of its end states becomes activated (cf. Marking Rule MR21). In this case, the micro process instance is marked as FINISHED. If the micro process instance that corresponds to the primary object instance of the macro process instance is FINISHED, this macro process instance is considered as FINISHED as well; i.e., the marking of the micro process instance indicates the processing state of the corresponding macro process instance. However, other micro process instances belonging to this macro process instance may still be RUNNING. These are no longer required for macro process instance termination. However, it might be desired that selected ones continue their execution. Opposed to this, other micro process instances must not be continued. To handle both situations, we introduce markings BYPASSED and SKIPPED for micro process instances (cf. Def. 68).

Example 12.12 (Termination of a macro process instance):
Regarding the recruitment example, each `job offer` object instance acts as primary object instance identifying a particular macro process instance. In turn, each `job offer` comprises a set of related `applications`, `reviews` and `interviews`. When a `job offer` terminates (i.e., end state `occupied` or `not occupied` is activated), some `applications` may still be marked as RUNNING; e.g., applicants not hired may have to be rejected. Opposed to this, `reviews` are no longer required since the decision on which `applicant` shall get the offered `job` has been already made.

Definition 68 (Micro process markings for detecting instances no longer required):
Let $M_{MicProc}$: MicProcInstances $\mapsto$ MicroProcessMarkings be the function assigning to each micro process instance micProcInstance its current marking $M_{MicProc}$(micProcInstance) with MicroProcessMarkings = {RUNNING, FINISHED, WAITING, BLOCKED}. We extend the markings of a micro process instance as follows:

MicroProcessMarkings := MicroProcessMarkings $\cup$ {BYPASSED, SKIPPED} (cf. Tab. 12.6).

Marking	Label	Description
☒	BYPASSED	A micro process instance will be marked as BYPASSED if the macro process instance it belongs to is marked as FINISHED, and the micro process instance itself has not terminated yet.
☒	SKIPPED	Authorized users may manually skip micro process instances whose execution is then skipped.

Table 12.6: Micro process markings detecting instances no longer required

Definition 69 (Micro process responsibility):
Let UserRoles be the set of all defined user roles. Then:

An **micro process responsibility** is a tuple **micResp = (micProcType, role)** where

- micProcType $\in$ MicProcTypes is a micro process type.
- role $\in$ UserRoles is a user role.

MicroProcessResponsibilities corresponds to the set of all definable micro process responsibilities.

To indicate to users when micro process instances are no longer mandatorily required, respective instances are marked as BYPASSED. More precisely, all micro process instances belonging to a macro process instance whose primary micro process instance becomes marked as FINISHED will be marked as BYPASSED indicating that their execution is no longer required in the context of the macro process instance. For this purpose, the micro process instances belonging to a particular macro process instance need to be adequately determined. In this context, we have to consider that a micro process instance may belong to several macro process instances. To handle this, we extend the micro process instance definition given in Def. 24. In particular, each micro process instance additionally comprises a set of oids representing the macro process instances they belong to (cf. Def. 70).[9]

Definition 70 (Identifying micro process instances):
Let micProcInstance be a micro process instance belonging. Then:

$\text{MacProcInstanceSet}_{\text{micProcInstance}}$ is a set of oids identifying the macro process instances to which micProcInstance belongs to.

In the following, a micro process instance represents a tuple **micProcInstance = (micProc, oid, M_{State}, M_{MicStep}, M_{MicTrans}, $M_{\text{BackTrans}}$, M_{Data}, M_{Act}, MacProcInstanceSet)**.

Marking Rule (MR33: Marking micro process instances as BYPASSED):
Let macProcInstance be a macro process instance and micProcInstance its primary micro process instance; i.e., macProcInstance.oid = micProcInstance.oid. Then:

M_{MicProc}(micProcInstance) = FINISHED, $\Rightarrow$

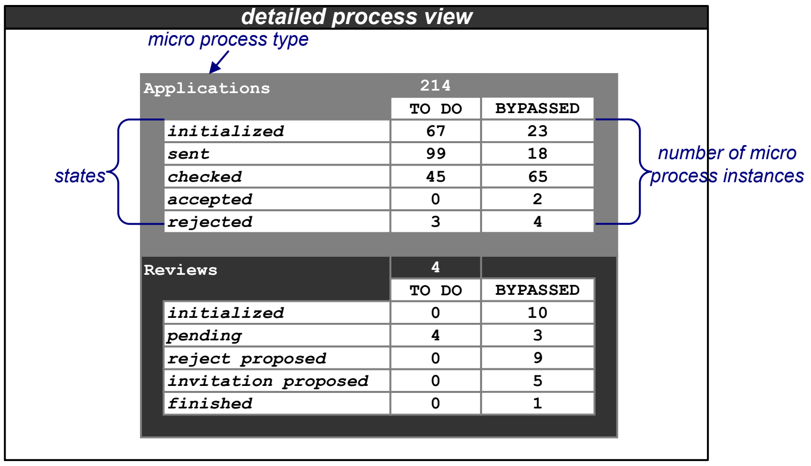
    ∀ micProcInstance' with micProcInstance'.oid ∈ micProcInstance.MacProcInstanceSet ∧
        M_{MicProc}(micProcInstance') ∉ {FINISHED, SKIPPED} ∧ ∄ micProcInstance" with
            micProcInstance".oid ∈ micProcInstance.MacProcInstanceSet ∧
            M_{MicProc}(micProcInstance") ∈ {WAITING, BLOCKED, RUNNING}:

                M_{MicProc}(micProcInstance') := BYPASSED;

i.e., if the primary micro process instance of a macro process instance terminates, all other micro process instances that belong to this macro process instance and have not been marked as FINISHED or SKIPPED yet, will be re-marked as BYPASSED (if the micro process instance does not belong to another currently running macro process instance).

A micro process instance marked as BYPASSED is displayed to responsible users who may then decide on whether to continue or terminate it. For this purpose, an additional column is displayed in the process-oriented view (cf. Fig. 12.21) of authorized users. Based on this view, an overview table can be invoked listing the respective micro process instances (cf. Fig. 12.22). Additionally, overview tables contain a column displaying the current marking of the respective micro process instance (cf. Fig. 12.22). As defined by *Execution Rule ER9*, authorized users may skip selected instances. (cf. Chapt. 13 for details). In turn, if a micro process instance is skipped (cf. Def. 71), it will be automatically re-marked as SKIPPED (cf. *Reaction Rule RR12*) and hence its execution stops.

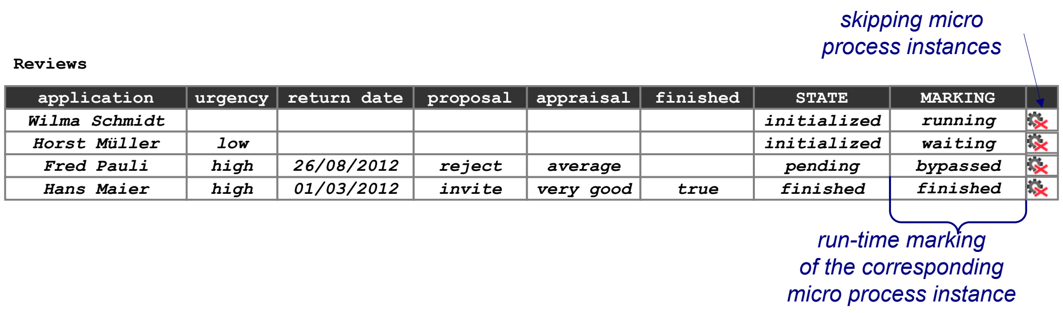
[9]Note that a micro process instance is further treated as RUNNING as long as one corresponding macro process instance is still RUNNING.



Applications	214	
	TO DO	BYPASSED
initialized	67	23
sent	99	18
checked	45	65
accepted	0	2
rejected	3	4

Reviews	4	
	TO DO	BYPASSED
initialized	0	10
pending	4	3
reject proposed	0	9
invitation proposed	0	5
finished	0	1

Figure 12.21: Process-oriented user view displaying bypassed micro process instances


skipping micro process instances
run-time marking of the corresponding micro process instance

Reviews

application	urgency	return date	proposal	appraisal	finished	STATE	MARKING	
Wilma Schmidt						*initialized*	*running*	
Horst Müller	*low*					*initialized*	*waiting*	
Fred Pauli	*high*	*26/08/2012*	*reject*	*average*		*pending*	*bypassed*	
Hans Maier	*high*	*01/03/2012*	*invite*	*very good*	*true*	*finished*	*finished*	

Figure 12.22: Data-oriented user view with bypassed micro process instances

Execution Rule (ER9: Skipping micro process instances):
Let MicroProcessResponsibilities be the set of all definable micro process responsibilities. Then:

$\forall$ micResp = (micProcType, r) $\in$ MicroProcessResponsibilities:
A micro process instance of micProcType may be skipped by any user owning r.

Definition 71 (Skipping micro process instances):
Let MicProcInstances be the set of all instances of any micro process type. Then:

skipped: MicProcInstances $\mapsto$ BOOLEAN defines whether or not a particular micro process instance has been skipped.

Reaction Rule (RR12: Marking micro process instances as SKIPPED):
Let micProcInstance $\in$ MicProcInstances be a micro process instance. Then:

skipped(micProcInstance) = TRUE, $\Rightarrow$ $M_{MicProc}$(micProcInstance) := SKIPPED;

i.e., when a user skips a micro process instance, it will be marked as SKIPPED.

Fig. 12.23 illustrates all rules required to handle the correct termination of a macro process instance and its corresponding micro process instances respectively. After marking the primary micro process instance of a macro process instance as FINISHED (cf. Marking Rule MR21), all other micro process instances belonging to this macro process instance, which have not been FINISHED or SKIPPED yet, are re-marked as BYPASSED (cf. Marking Rule MR33). Authorized users may then skip micro process instances no longer required (cf. Execution Rule ER9 and Reaction Rule RR12). Note that micro process instances not marked as BYPASSED may be SKIPPED as well. For example, this is required to handle deadlock situations (i.e., micro process instances marked as BLOCKED). For details we refer to Sect. 13.2.

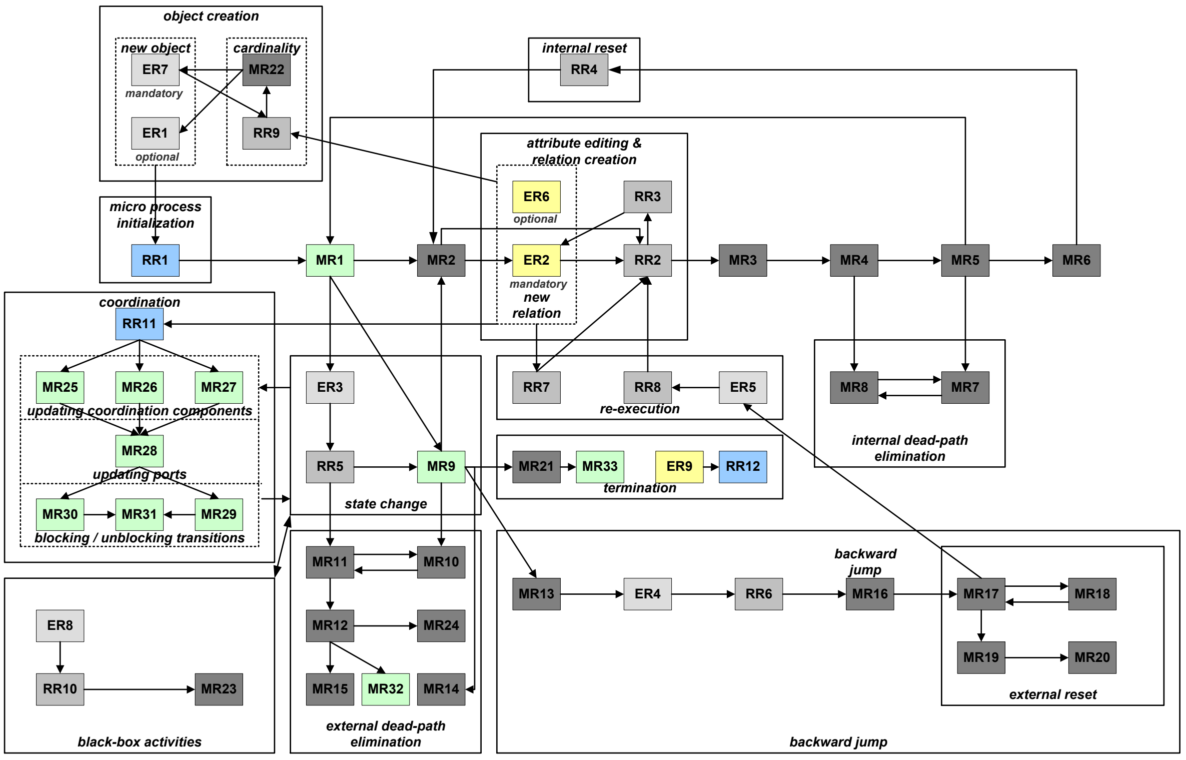


Figure 12.23: Rules for terminating macro and micro process instances

12.10 Further Issues

This thesis focuses on fundamental concepts enabling a proper support of object-aware processes. Regarding the coordination of micro process instances, however, more detailed concepts are required in practice. Due to lack of space, this section only sketches basic ideas. First, in addition to the activation of states, the creation of the object instances (and micro process instances respectively) may depend on coordination components; i.e., whether or not another instance of a particular object type may be created in a given context not only depends on cardinalities defined by the data model, but on coordination components as well (cf. Sect. 12.10.1). Second, the creation of relations may depend on the defined coordination components as well (cf. Sect. 12.10.2). Third, it must be ensured that backward jumps which may be applied during the execution of a particular micro process instance do not result in incorrect processing states of a process structure (cf. Sect. 12.10.3).

12.10.1 Creation of Object Instances

Regarding macro process types, we need to consider several special cases: For example, a macro step type may refer to a start state type of a micro process type (cf. Fig. 12.24). In this case, macro processes and corresponding coordination components not only influence the execution of running micro process instances, but also their creation; i.e., not only the activation of a certain state, but also the creation of an object instance and micro process instance may depend on the execution of other micro process instances.

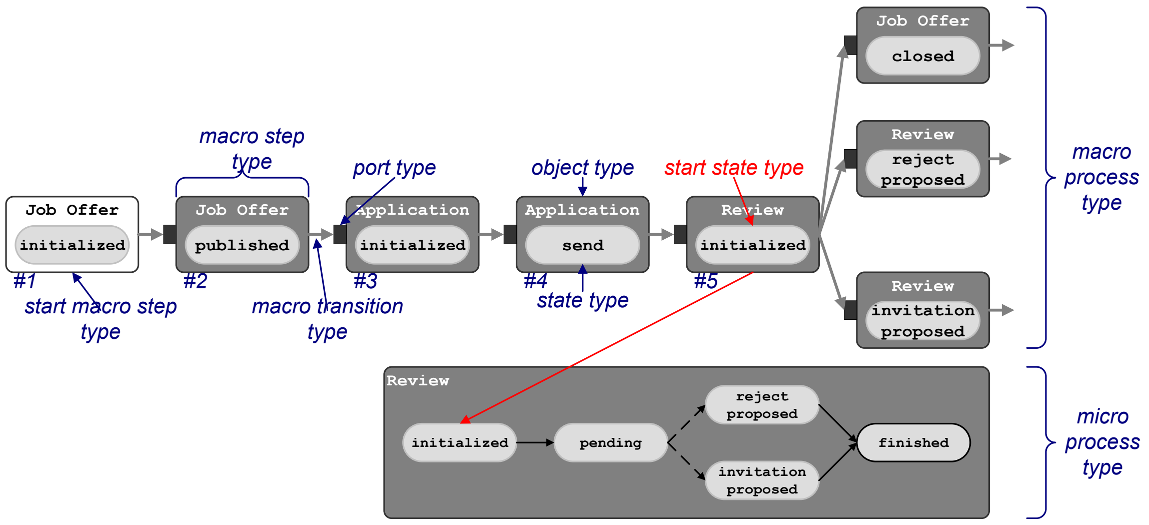


Figure 12.24: Macro step types referring to start state types

Example 12.13 (Macro step types referring to start state types):
Consider Fig. 12.24 and the macro step type referring to object type `review` and state type `initialized`. The macro transition type targeting at the latter originates from a macro step type referring to object type `application` and state type `send`. Since the relationship between the `application` and `review` object types represents a top-down dependency, a corresponding process context type is defined. This way, the creation of a new `review` object instance targeting at this particular `application` object instance depends on the currently activated state of the `application` micro process instance.

As discussed in Sect. 9.1.2, whether or not an object instance of a lower-level object type may be created is managed by a creation context that controls the defined cardinalities. According to Def. 34, a creation context belongs to a higher-level object instance managing the set of corresponding lower-level object instances of a certain type. Depending on the number of currently existing lower-level object instances, the creation context is either marked as ACTIVATED, CONFIRMED, or BLOCKED (cf. Def. 35). Hence, coordination components must be considered as well.

Example 12.14 (Creation context considering process contexts):
Consider Fig. 12.25 and the creation context managing all `review` object instances that refer to a particular `application` object instance. A newly created `review` object instance may only be assigned to a particular `application` object instance if either state `send` or `checked` is activated for the micro process instance corresponding to the `application` object instance.

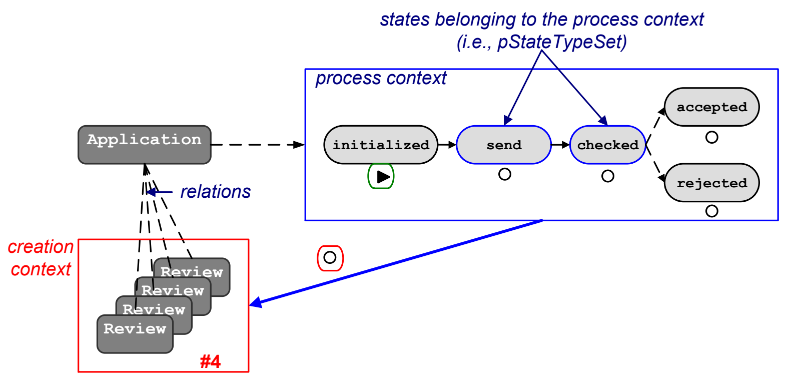


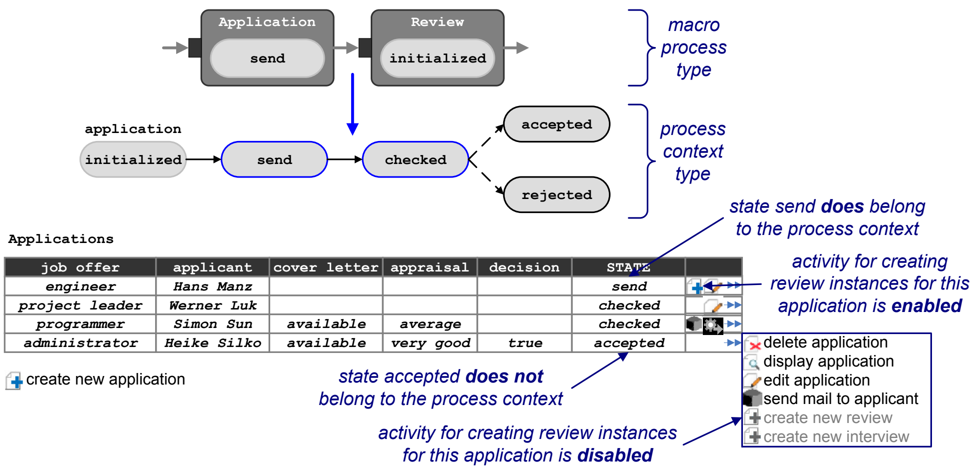
Figure 12.25: Creation context in response to a process context

When re-marking a creation context instance, in addition to the defined cardinalities, selected coordination components must be considered.[10] As a consequence, coordination types may not only belong to particular port types (cf. Defs. 57, 60, and 62), but to creation context types as well. In turn, a creation context (cf. Def. 34) may comprise a set of coordination components. In the latter case, markings WAITING and ACTIVATED will be assigned if all coordination components are marked as ACTIVATED.

Lower-level object instances, required by the defined cardinality constraints, are usually created in the context of a higher-level object instance (e.g., to fulfill a minimum cardinality). The activity for creating a new object instance is then only executable in the context of higher-level object instances for which the creation is allowed. In particular, if not all coordination components of the respective creation context are marked as ACTIVATED, the creation of new lower-level object instances must be disabled.

Example 12.15 (Disabling creation):
Consider Fig. 12.26. For the `application` object instance displayed in column `#4`, state `accepted` is currently activated. Since this state does not belong to the defined process context type, additional `review` object instances cannot be created. Hence, no activity for creating `review` object instances may be invoked. Opposed to this, for `application` object instance `#1`, state `send` is still activated. Since this state belongs to the corresponding process context type, the activity for creating `review` object instances is enabled. Note that for this `application` object instance the desired minimal cardinality is not met. Hence, the activity for creating lower-level `review` object instances is mandatory.

[10] Note that macro transition types targeting at start state types are either categorized as top-down or transverse (since lower-level object instances do not exist before the higher-level ones they refer to are created). We ensure this at build-time according to Def. 54h. In both cases (i.e., top-down and transverse relationships), the creation of a lower-level object instance depends on the referenced higher-level one. Regarding top-down dependencies, a corresponding process context type defines in which states of the higher-level micro process instance, lower-level object instances may be created. In turn, when using a transverse relationship, a lower-level object instance may only be created if the defined constraint evaluates to true. The latter is based on micro process instances of another lower-level object type.


Application
send
Review
initialized
macro process type
application
initialized
send
checked
accepted
rejected
process context type
state send **does** belong to the process context
activity for creating review instances for this application is **enabled**
state accepted **does not** belong to the process context
activity for creating review instances for this application is **disabled**
delete application
display application
edit application
send mail to applicant
create new review
create new interview
create new application

Applications

job offer	applicant	cover letter	appraisal	decision	STATE	
engineer	Hans Manz				send	
project leader	Werner Luk				checked	
programmer	Simon Sun	available	average		checked	
administrator	Heike Silko	available	very good	true	accepted	

Figure 12.26: Disabling the creation of object instances

12.10.2 Handling Relations

When creating a new object instance independently from a higher-level one, the relation to a selected higher-level object instance is usually assigned afterwards; i.e., when invoking a form for editing attribute values and relations. Therefore, a combobox is offered that allows users to select the desired higher-level object instances (by displaying the label attribute specified). In this context, only those higher-level object instances, for which the creation of a corresponding lower-level object instance is currently allowed, should be displayed in the combobox; i.e., the corresponding creation context must be evaluated. All other relations not allowed are considered as *unsound relations*. Further, consider the case for which a macro step type refers to the start state of the source object instance of the relation. Here, the creation context must take the corresponding coordination components into account as well (cf. Sect. 12.10.1). Regarding the generated user form, all higher-level object instances, for which the creation context is not marked as WAITING or ACTIVATED, are not considered when assigning the relation.

Example 12.16 (Preventing unsound relations):
Consider Fig. 12.26. `Review` object instances may only be created for `applications` being in state `send` or `checked`. Hence, only `application` object instances displayed in columns #1, #2, and #3 are offered in the combobox when invoking the form for editing a `review` object instance; i.e., the `application` object instance displayed in column #4 is removed since state `accepted` has already been activated.

If a relation, which is required for micro process coordination, is missing, process execution is blocked until the required relation is assigned. In this context, no source micro process is currently defined for the respective coordination component. According to Reaction Rule RR1, the respective coordination component is then marked as ENABLED. Since a port is only marked as ACTIVATED if its coordination components are all marked as ACTIVATED, it is not possible to activate the respective state (or to create a new object instance); i.e., the correct execution of macro process instances is ensured (when preventing unsound relations as discussed). However, deleting relations and re-assigning them must be considered as well. In this case, parts of the process structure may be disconnected and coupled to another higher-level object instance (and micro process instance respectively), which may belong to different macro process instances. Regarding aggregations and transverse instances, for example, the assignment of an additional lower-level object instance may affect the evaluation of the respective predicate (e.g., the predicate may not evaluate to true any longer). Especially, this is critical when coordination components already marked as CONFIRMED or SKIPPED are involved.

Example 12.17 (Incorrect processing state):
Consider Fig. 12.27. State `finished` of a `review` micro process instance may only be activated if for the corresponding `application` either state `accepted` or `rejected` is marked as ACTIVATED. As a consequence, re-assigning a `review`, for which state `finished` is currently activated, to an `application` in state `send` or `check` would result in an incorrect processing state.

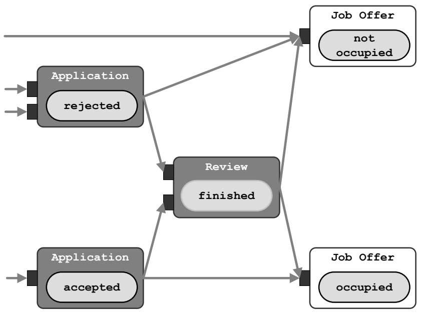


Figure 12.27: Macro process type for finishing reviews

For the above reasons, not all relations between object instances are allowed. This must be considered when automatically creating user forms; i.e., not all higher-level object instances are displayed in the corresponding combo box. According to Marking Rules MR25, MR26, and MR27, only those coordination components are re-marked that have not already been marked as CONFIRMED or SKIPPED. This must be taken into account when re-assigning relations. In particular, if the deletion of a relation affects coordination components already marked as CONFIRMED or SKIPPED, it will be prohibited.

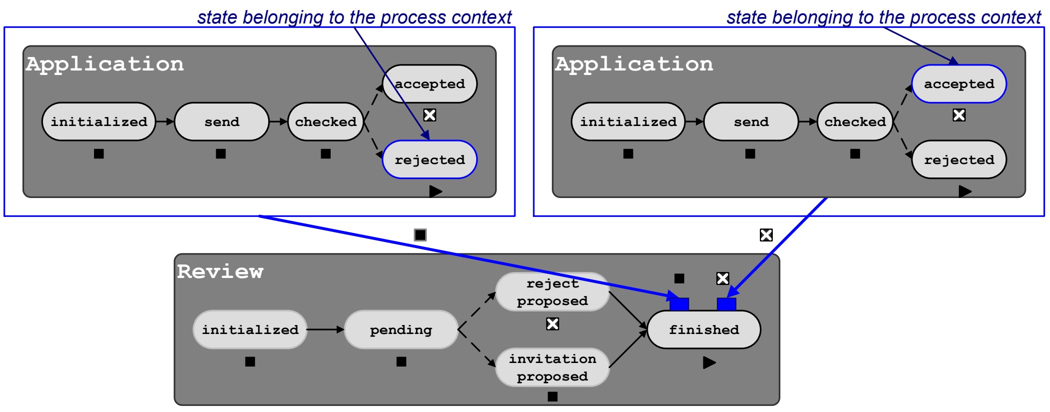


Figure 12.28: Process context instances for finishing reviews

Example 12.18 (Handling coordination components already applied):
Consider Fig. 12.28. State `finished` of a `review` micro process instance is currently marked as ACTIVATED. Regarding the `application` micro process instance this `review` refers to, state `rejected` is marked as ACTIVATED. This has led to the activation of a port belonging to state `finished` of the `review`, which is therefore marked as CONFIRMED; i.e., this port was used to activate state `finished`. In turn, the other port is marked as SKIPPED. Since the execution of the `review` micro process instances has been dependent on the `application` instance, it is not possible to delete and potentially re-assign the corresponding relation afterwards.

Generally, testing the coordination components in advance is not always appropriate. For specific use cases it might be required to flexibly re-assign their markings anyway. In this context, advanced concepts for adapting the resulting process structure are required. For example, incorrect parts of the process structure must be identified and marked accordingly. Additionally, respective information must be propagated to the macro process instance level; i.e., sophisticated techniques for exception handling are required.

12.10.3 Backward Jumps

When performing a backward jump during the execution of a micro process instance, a previous state becomes re-marked as ACTIVATED (cf. Marking Rule MR16). In addition, according to Marking Rule MR19, all states succeeding the target state of the backward transition are reset; i.e., they are re-marked as WAITING. Hence, if the micro process instance corresponds to the source of one or more coordination components, its marking must be updated accordingly. For this purpose, as highlighted in Fig. 12.29, Marking Rule MR16 for performing backward jumps as well as Marking Rule MR19 for re-setting states must be both extended (similar to Marking Rule MR9). The latter performs a normal state change (cf. Sect. 8.5). In particular, all affected process context, aggregation, and transverse instances must be re-marked as ENABLED. In turn, this triggers Marking Rules MR25, MR26, and MR25 for updating the respective marking.

However, if coordination components already marked as CONFIRMED or SKIPPED are involved, a backward jump might lead to an improper processing state of the macro process instance. Consequently, it is not allowed to execute a backward jump if coordination components that have been already marked as CONFIRMED or SKIPPED are involved.[11] This must be evaluated when a new state becomes marked as ACTIVATED. Only the backward transitions originating from this state that do not affect already used coordination components are then marked as CONFIRMABLE. For this purpose, Marking Rule MR13 (which re-marks backward transitions as CONFIRMABLE) must be extended as well (cf. Fig. 12.29).

Again, there exist specific use cases requiring the execution of backward jumps even if already applied coordination components are concerned (e.g., for exception handling). Then, additional concepts for adapting the resulting process structure are required.

[11] This is similar to the consideration of coordination components when assigning relations.

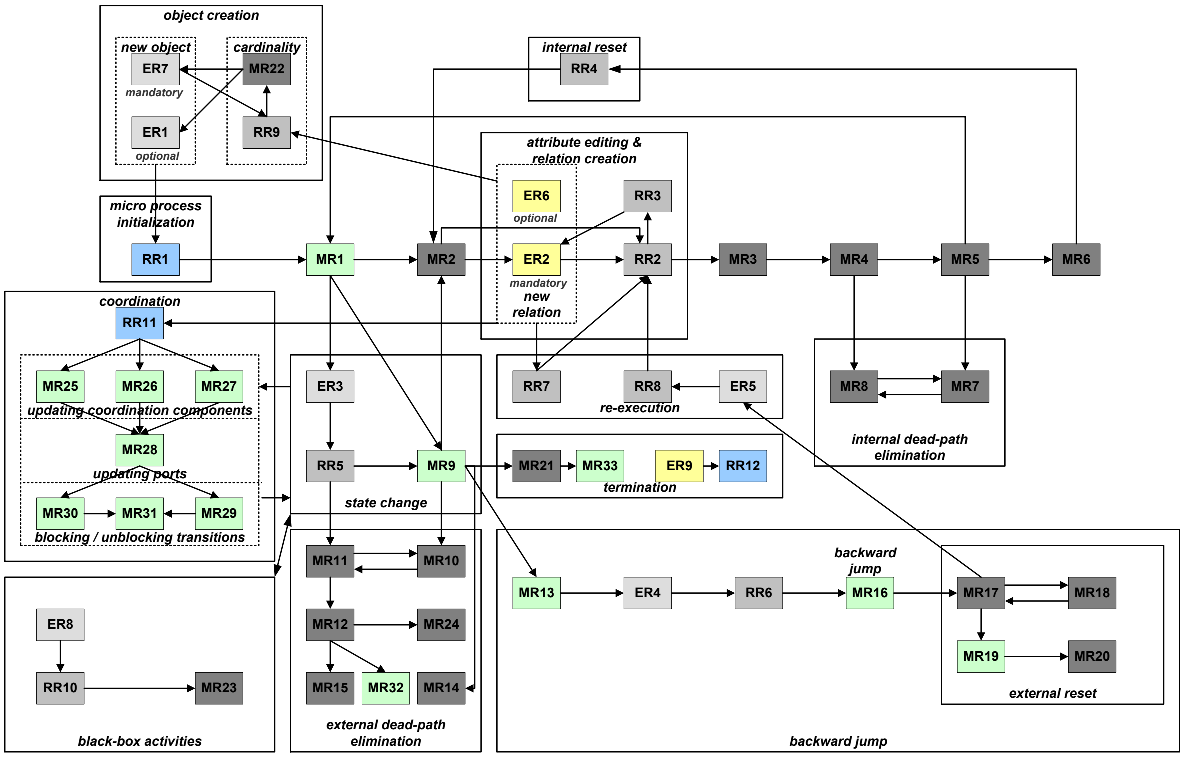


Figure 12.29: Rules for backward jumps

12.11 Summary

In summary, for each macro process instance, a complex process structure emerges during run-time. Regarding the latter, individual micro process instances need to be coordinated. Coordination component types define define synchronization constraints between micro process instances based on their corresponding states. In particular, whether a particular state may be activated additionally depends on its ports and their corresponding coordination components. For this purpose, the operational semantics introduced for micro process execution (cf. Chapt. 8) is extended in two directions. First, whether a certain state of a particular micro process instance involved in this process structure can be activated depends on the various involved coordination components. Second, the creation of additional object instances and micro process instances respectively is now managed considering the defined cardinality constraints. Taking the comprehensive set of rules and their dependencies to another into account, macro process execution seems to be very complex. However, this complexity is required to provide to desired execution flexibility at run-time and to enable an integrated view on processes and data for its users.

13 Further Issues

This thesis provides basic concepts for modeling and executing object-aware processes. However, there exist several issues which are also relevant in order to provide a comprehensive framework. Regarding the process life-cycle, this chapter sketches additional ideas for process monitoring (cf. Sect. 13.1) and exception handling (cf. Sect. 13.2). Furthermore, advanced concepts for user integration and data processing are discussed in Sects. 13.3 and 13.4. Since these topics do not constitute the focus of this thesis, we discuss them only informally.

13.1 Monitoring

Regarding a process structure emerging during run-time, its micro process instances (of same and different type) are executed asynchronously to each other. Consider thereto Fig. 13.1 in which the currently activated state of each micro process instance is colored blue. Based on such a view, however, it is a challenging task to figure out the overall processing state of this process structure.

In traditional PrMS, the state of a process instance is characterized by its currently activated process steps. Opposed to this, a macro process step aggregates a number of micro process instances for which different states may be activated; i.e., only for a sub-set of these micro process instances the state specified by the macro step may be reached (cf. Fig. 13.2). In turn, other micro process instances may have already left this state or skipped it. Consequently, it is not that easy to figure out the overall state of a macro process instance.

To consider the asynchronous micro process execution within an aggregated macro process view, similar counters as specified by Def. 59 are provided; i.e., each macro step comprises counters (cf. Fig. 13.2) managing the total number of corresponding micro process instances (#ALL), the number of instances for which the respective state is currently activated (#IN), the number of instances which have not reached the respective state (#BEFORE), have skipped

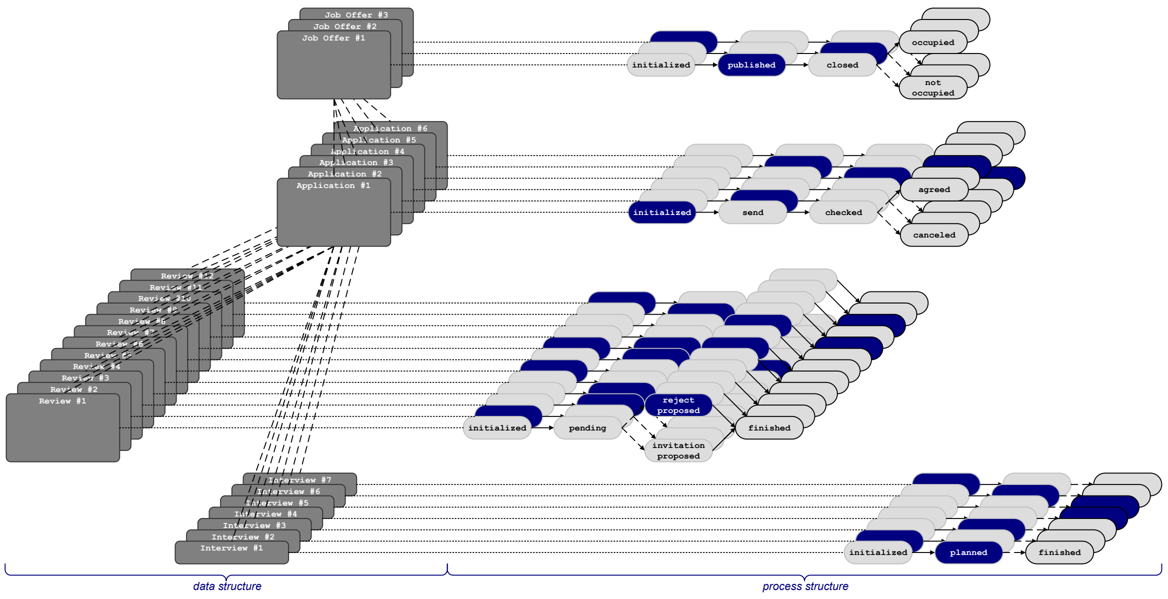


Figure 13.1: Asynchronous execution of micro process instances

it (#SKIPPED), or the state has been left and a subsequent state is currently activated (#AFTER). These counters aggregate the processing states of the corresponding micro process instances. Furthermore, when selecting a particular counter, corresponding object instances (and micro process instances respectively) are listed in an overview table. This way, features for navigating through the process structure are realized. Consider therefore our prototypical implementation as introduced in Sect. 14.1. There, Fig. 14.9 illustrates monitoring facilities using the aforementioned counters.

13.2 Exception Handling

During process execution, exceptions cannot be always avoided. First, regarding a micro process instance, *inconsistencies* between process and data state may occur when object attribute values committed in a previous state are changed later on (cf. Sect. 13.2.1). Second, macro process execution might lead to *deadlocks* when an external micro transitions becomes marked as BLOCKED (or WAITING). PHILharmonicFlows provides concepts to identify and handle such exceptions. In any case, the framework guarantees correct process execution.

13.2.1 Exception Handling During Micro Process Execution

When assigning optional write permissions, users may change attribute values, which have been already committed in a previous state, later on. More precisely, a user may own optional write permissions for object attributes mandatorily required in a previous state. When the values of these attributes are changed afterwards, inconsistencies between process and data state

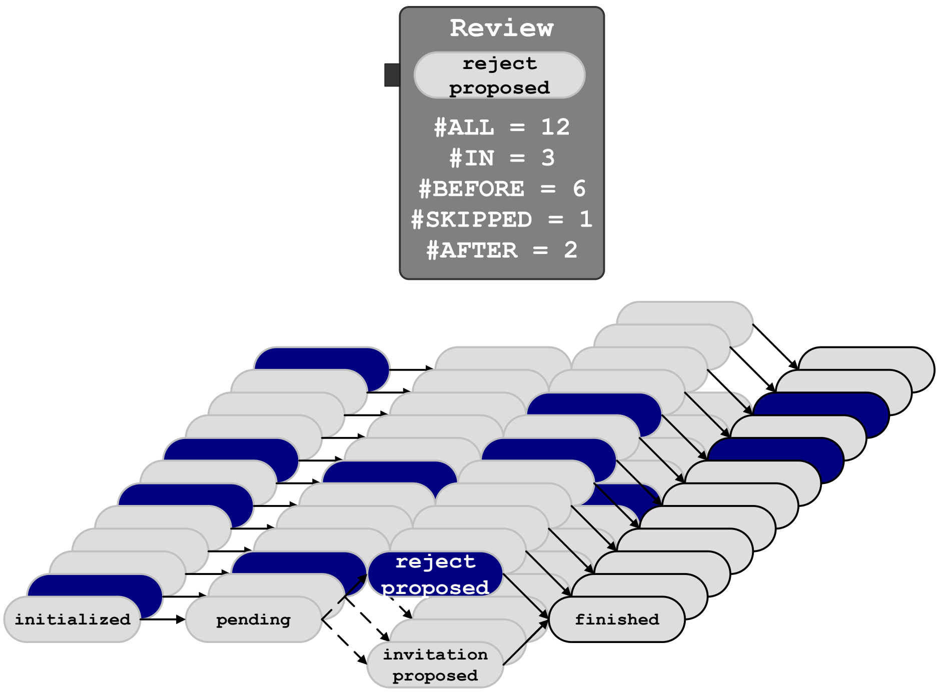


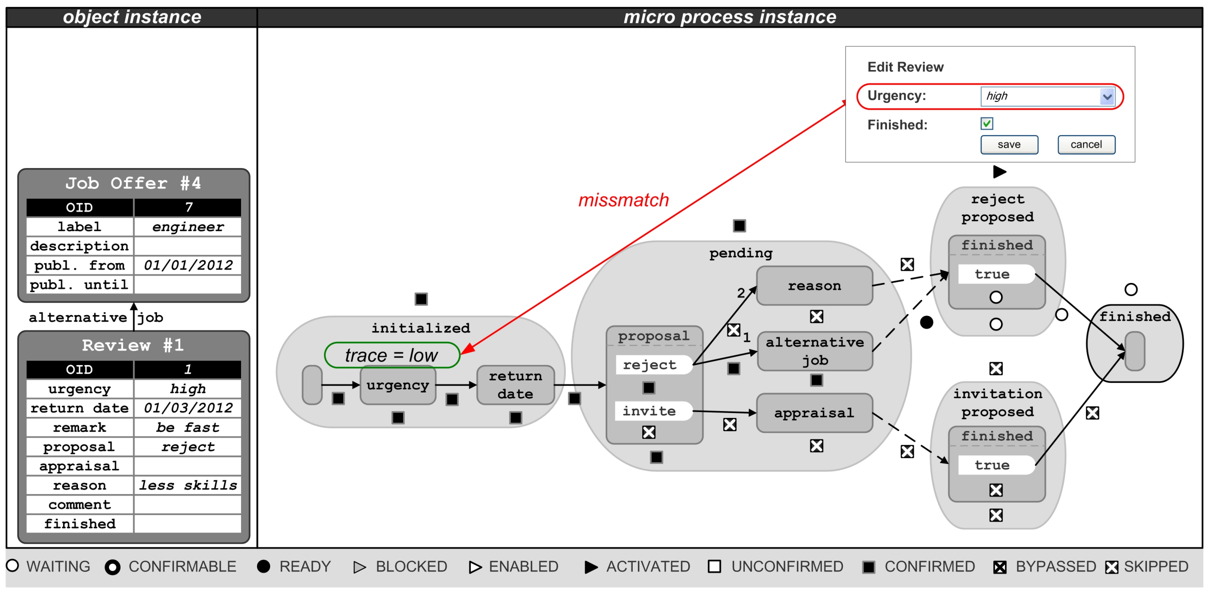
Figure 13.2: Macro steps for aggregating micro process instances

might occur. However, such inconsistencies cannot always be prohibited since certain real-world scenarios require late changes of already committed object attribute values (cf. Ex. 13.1).

Example 13.1 (Changing attribute values later on):
Consider Fig. 13.3. When state `initialized` was activated, the personnel officer assigned value "low" to object attribute `urgency`. Later on, when state `reject proposed` becomes activated, again, the personnel officer owns write permissions for object attribute `urgency`. This allows him to change the value from "low" to "high". However, the employee editing the `review` in state `pending` has assumed that the `urgency` for filling in the `review` is "low".

For detecting such inconsistencies between process state and data state, similar concepts as introduced in Sect. 8.4.6 are applied. In particular, *traces* (cf. Def. 26) are used in this context. If a trace of a micro step currently marked as CONFIRMED does not coincide with the actual value of the corresponding object attribute, the latter is marked as INCONSISTENT; i.e., an additional *data marking* is defined (cf. Def. 29). Further, in addition to execution and transition responsibilities (cf. Defs. 19 and 20), a *micro process responsibility* is introduced. In case of an inconsistency, the latter must perform an *error treatment* (see below). As long as the mismatch between the object attribute value and the trace is not dissolved, it is not possible to further process the object instance; i.e., all activities are deactivated.
In this context, for treating errors the user owning a respective micro process responsibility is enabled to perform one of the following actions:

- reset the object attribute value to the value stored in the trace,
- perform a backward jump to the state the micro step of the trace belongs to, or



Job Offer #4

OID	7
label	*engineer*
description	
publ. from	*01/01/2012*
publ. until	

Review #1

OID	1
urgency	*high*
return date	*01/03/2012*
remark	*be fast*
proposal	*reject*
appraisal	
reason	*less skills*
comment	
finished	

Edit Review

Urgency:	*high*
Finished:	✓

Figure 13.3: Late changes of attribute values

- accept the inconsistent attribute value (i.e., proceed with process execution despite the inconsistency).

After performing one of these actions, the data marking (cf. Def. 29) of the object attribute value changes from INCONSISTENT to ASSIGNED and all activities are activated again.

13.2.2 Exception Handing During Macro Process Execution

As discussed, a macro process instance comprises coordination components for synchronizing the execution of corresponding micro process instances. If a subsequent state of a micro process instance cannot be ACTIVATED, the respective micro transition is marked as BLOCKED or WAITING (cf. Marking Rules MR1"" and MR30). [1] In addition, to quickly recognize such situations, these markings are propagated to the micro and macro process instance (cf. Marking Rule MR31) and are used for monitoring issues. To get an impression about the end-user perspective on deadlocks, consider the proof-of-concept prototype introduced in Sect. 14.1.3. Fig. 14.9 illustrates a view showing a macro process instance in a deadlock.

For dissolving deadlocks, in addition to micro process responsibilities, a *macro process responsibility* is introduced. In addition, marking BYPASSED is used for coordination components and ports respectively; i.e., the set of markings provided for coordination components and ports is extended (cf. Def. 65). The user owning the macro process responsibility at run-time may then

[1] Marking BLOCKED is only assigned in connection with a process context (cf. Def. 58). Regarding aggregations and transverse components, the defined predicates depend on the actual number of currently created object instances and micro process instances respectively. Therefore, it is not possible to automatically determine whether a predicate may evaluate to true later on. To detect deadlock situations in this context, marking WAITING is used and visualized when monitoring macro processes. Whether or not an intervention is required must then be decided by the responsible user.

mark particular coordination components and ports as BYPASSED. Respective components are then no longer considered for coordinating the micro process instances.

Details about the end-user interfaces provided for performing such assignments are provided in [Sch10].

13.3 Advanced Concepts for User Integration

Access control mechanisms (i.e., authorization) protect data from unauthorized access (*confidentiality*) and changes (*integrity*) [Ber98]. Coincidently, one has to ensure that each user gets access to all required data, functions and processes (*availability*) [Ber98, FK92]. In principle, access control must be considered at different layers of abstraction [SV01] (cf. Fig. 13.4): strategies, models, and mechanisms. *Strategies* determine which components (e.g. data, functions, processes) within a system shall be protected, and define the required kinds of privileges. In turn, a *model* formally represents the applied strategy, whereas the used *mechanism* determines its technical implementation. Existing systems have earned many benefits by applying *Role-Based Access Control (RBAC)* [FK92] as strategy. The additional layer between users and privileges allows for faster and less cumbersome administration [BFA99]. Furthermore, users with same positions or duties get same rights [FK92]. Hence, our strategy for access control is generally based on role-based concepts for user integration. However, our work extends existing concepts to fulfill the requirements arising from the tighter integration of data and functions with the corresponding process support.

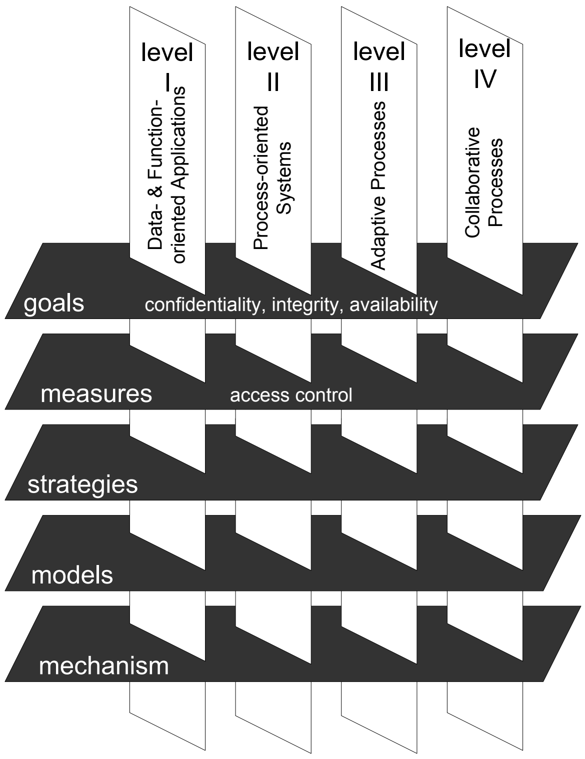


Figure 13.4: Classification of access control (according to [KR09b, KR10])

Complementary to these abstraction layers (i.e., strategies, models and mechanisms), different kinds of systems make different claims in respect to the strategy needed. As illustrated in Fig. 13.4, access control can be arranged in four levels, each of them depending on the functionality of the system [Pfe05]. Since our goal is to provide integrated access to data, functions and processes in a generic way, we need a strategy comprising Level I and Level II. In turn, Levels III and IV are used for process-aware information systems enabling adaptive or collaborative processes [WRWR05]. Such processes are out of the scope of this thesis. We first consider the abstraction layer of the strategy dimension when evaluating properties of object-aware processes. In particular, we discuss which requirements must be fulfilled by a strategy for access control when simultaneously addressing Level I and Level II. We further provide a concrete model for this strategy in the context of our PHILharmonicFlows solution framework. One mechanism making this model work is then implemented in our prototype.

Regarding the dependencies between data and user discussed in Sect. 3.1, user permissions do not only depend on roles, but also on data properties (cf. Prop. 16) and on relationships between users and object instances (cf. Prop. 17). Consequently, it is not sufficient to manage the user and role definitions as well as the organizational model independent from application data. Instead, PHILharmonicFlows allows integrating users with objects. For this purpose, an object type may be flagged as *user type* (cf. Fig. 13.5). At run-time, each object instance corresponding to a user type represents a particular user. To authenticate this user when he logs in, each user type must include attribute types uniquely identifying users at run-time (e.g., username and password).

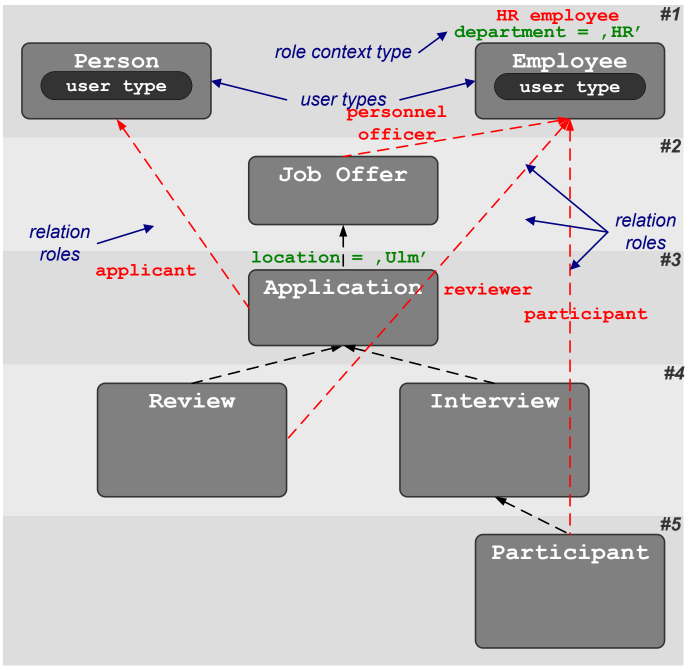


Figure 13.5: User types

Example 13.2 (User types):
Regarding the recruitment example (cf. Fig. 13.5), relevant user types are `person` and `employee`.

Each user type automatically represents a *user role*. When a user logs in, PHILharmonicFlows determines, which instance of the user types corresponds to the entered user name and password. Based on this, the user role represented by the corresponding user type is then automatically assigned.

Example 13.3 (Login):
When a user enters his user name and password, all instances of the user types `person` and `employee` are checked. This way, he is either identified as `person` or `employee`.

Further, for each user type more specific user roles may be defined by constraining the attribute types of the user type. Like for value step types (cf. Sect. 7.1.2), the options existing for the definition of such constraints depend on their technical implementation (e.g., whether OR- and AND-operators can be used). We denote these constraints as *role context types*. When a user logs in, PHILharmonicFlows evaluates whether the role context types defined for the respective user type are met; i.e., whether the constraint is fulfilled. In this case, the role represented by the role context type is additionally assigned (cf. Ex. 13.4).

Example 13.4 (Role context types):
Regarding the recruitment example (cf. Fig. 13.5), for user type `employee`, a role context type is defined identifying all `employees` of the human resource (HR) department; i.e., role `HR employee` is defined.

As advantage of this close integration of data and users, we can automatically determine additional user roles from a given user type and its relations to other object types. We denote these as *relation role types*. In particular, each relation type targeting at a user type represents a relation role type. At run-time, each user owns the relation role if there exists at least one object instance of the source object type of the respective relation which references the user instance. [2] .

Example 13.5 (Relation roles):
Regarding the user types illustrated in Fig. 13.5, the relation role types `applicant`, `personnel officer`, `reviewer`, and `participant` (in an interview) can be derived. Hence, each user represented by a `person` user instance additionally owns user role `applicant` if there exists at least one `application` object instance referencing the instance of type `person` identifying him.

Usually, for a particular user, access to data should be restrictable to a subset of the object instances of an object type. We denote such fine-grained access control as *instance-specific role assignment*. In this context, PHILharmonicFlows differentiates between cases in which access depends on object attribute values and cases which require a specific relationship between the object and the user instance. Since both depend on the object type, for which access shall be granted, such specific assignments are captured in the authorization table maintained for each object type (cf. Sect. 9.1.1). In particular, permissions (cf. Def. 32) may be associated with

[2]Note that a relation role type does not enable instance-specific access. Each `person`, for which an `application` exists, owns relation role `applicant`. In order to restrict access to particular object instances (e.g., on his own `application`) additional concepts as discussed in the following are required.

a constraint which is defined based on attribute values. We denote such a constraint as *data context*. At run-time, users owning the respective role may only access those object instances for which the constraint evaluates to true. [3] .

Example 13.6 (Instance-specific access using data contexts):
Using data contexts in connection with permissions, it becomes possible to grant access only to those `job offers` that belong to location "ULM".

We further consider instance-specific role assignment based on relations between user and object instances. While relation roles can be determined based on the relation types at build-time, instance-specific role assignment is based on the concrete relations existing at run-time. For this purpose, PHILharmonicFlows allows flagging permissions as *instance-specific* based on function **instance-specific: Permissions** $\mapsto$ **BOOLEAN**. Using this flag, the relationship between a certain user and object instance may be taken into account. In this context, PHILharmonicFlows differentiates between top-down, bottom-up, and transverse relationships (cf. Exs. 13.7, 13.8, and 13.9). [4] Similar concepts as used for defining coordination components (cf. Chapt. 11) are applied.

Example 13.7 (Instance-specific access based on a top-down relationship):
`Applicants` may only access their own `applications`.

Example 13.8 (Instance-specific access based on a bottom-up relationship):
`Participants` may only access those `interviews` in which they participated.

Example 13.9 (Instance-specific access based on a transverse relationship):
`Reviewers` may only access those `interviews` that belong to an `application` they have to evaluate.

Such fine-grained access control based on relations may require high efforts to determine the permissions of a user. Our current proof-of-concept prototype therefore does not determine these relationships at run-time.[5] Instead, relations to transitively referenced object instances are stored directly; i.e., additional relations representing transitive relationships are used. We denote these as *virtual attributes*. Note that this approach requires higher efforts when saving object instances and relations, but less effort is needed to determine the modeled relationships.

[3]This way, for a particular object type, different variants of an authorization table can be realized.
[4]Note that there may be several paths between the object type and the respective user type.
[5]Note that this would require inner-joins and sub-selects on the underlying data structure.

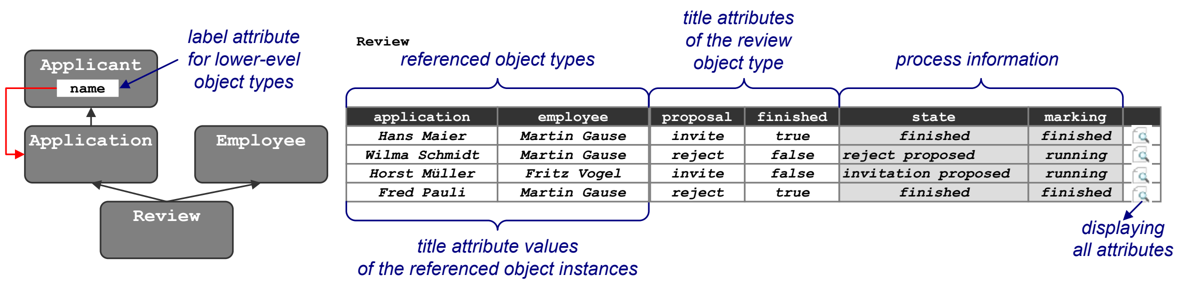
13.4 Advanced Concepts for Data Processing

13.4.1 Displaying Overview Tables

In most cases, the number of attributes corresponding to a particular object type is too large to display all of them in the overview table. Hence, only the label attributes defined for the object type and the relations to other object attributes are displayed. In this context, label attributes (cf. Def. 8) to be displayed can be derived from several other attributes and relations by composing their values .

Example 13.10 (Label attribute composing several attributes):
Consider the overview table listing `review` object instances (cf. Fig. 13.6). Attributes `proposal` and `finished` are defined as label attributes.

Example 13.11 (Relation as label attribute):
Consider again Fig. 13.6. The name of the `applicant` is used as label attribute for the lower-level object type `application`.



application	employee	proposal	finished	state	marking	
Hans Maier	*Martin Gause*	*invite*	*true*	*finished*	*finished*	
Wilma Schmidt	*Martin Gause*	*reject*	*false*	*reject proposed*	*running*	
Horst Müller	*Fritz Vogel*	*invite*	*false*	*invitation proposed*	*running*	
Fred Pauli	*Martin Gause*	*reject*	*true*	*finished*	*finished*	

Figure 13.6: Displaying overview tables

Using this approach, additional information on object instances can be viewed on-demand involving a respective optional activity (cf. Fig. 13.6). Furthermore, users may select additional columns (i.e., attribute or relation types) they would like to be displayed in the overview table. For further details we refer to [Sch10] (cf. Sect. 6.2.10).

Finally, overview tables contain information about the corresponding micro process instance; i.e., there is a column reflecting the run-time marking of the micro process instance and its currently activated state (cf. Chapt. 8 for details).

13.4.2 Restricting Attribute Values and Relations

When editing object instances, in some cases, the set of possible attribute values must be restrictable (cf. Ex. 13.12). Using value types, a predefined set of values can be specified. However, even then only a subset of the value instances may be chosen. PHILharmonicFlows allows defining such restrictions based on constraints.

Example 13.12 (Restricting attribute values):
When editing an `application`, the `desired salary` must be higher than 1000 and lower than 100.000 Euro.

Not only attribute values, but also the assignment of relations must be restrictable (cf. Ex. 13.13). In this context, a *data context*, as introduced in Sect. 13.3, is used.

Example 13.13 (Restricting relations):
When editing an `application`, only `job offers` belonging to `location` 'Ulm' shall be selectable.

PHILharmonicFlows allows defining such restrictions in the context of particular *write permissions*. This way, it is possible to define different restrictions for different users. In addition, for a particular user, the restriction for one and the same attribute or relation may differ depending on processing states. Generally, advanced concepts which ensure that these restrictions are compliant with existing value step types are required.

Part IV

Evaluation and Discussion

14 Evaluation

Part III presented the PHILharmonicFlows framework, which supports the fundamental properties of object-aware processes as discussed in Part II. Part IV concludes the thesis with two chapters. Chapt. 14 evaluates the framework and Chapt. 15 summarizes major results and gives an outlook.

14.1 Proof-of-Concept Prototype

To demonstrate the technical feasibility of the presented concepts, this section introduces the proof-of-concept prototype developed in the context of this thesis. First, Sect. 14.1.1 lists the functional and technical requirements that were addressed in this context. Following this, Sect. 14.1.2 gives insights into the system architecture of this prototype and summarizes the core technologies used to realize it. Finally, we illustrate end-user interfaces in Sect. 14.1.3.

14.1.1 Motivation

The main concepts presented in this thesis were implemented in a powerful proof-of-concept prototype. It aims at demonstrating the practical feasibility of the PHILharmonicFlows framework and evaluating its core concepts. In addition, sophisticated end-user interfaces were realized to give an impression how users will perceive this framework.

In particular, the following functional requirements were addressed:

- Graphically defining data models, micro process types, macro process types, and authorization tables.
- Enabling a "correctness-by-construction" -principle and extensive correctness checks including a proper visualization of modeling errors.

- Enabling persistent storage of all created modeling artifacts (i.e., object and process types) as well as application data (i.e., object and process instances).
- Automatically generating user interface components (i.e., overview tables, forms, and worklists) based on the defined models.
- Implementing the operational semantics defined for micro and macro processes.

Further, the following technical requirements were considered when realizing the build- and run-time environment of the PHILharmonicFlows framework:

- Ensuring a high degree of usability.
- Enabling scalability and high performance.
- Enabling short development times.
- Enabling extensibility and changeability.

We first realized mock-ups and investigated in user interface design [Wag10, Sch10]. After implementing a first demonstrator of the prototype with Java [Prö11], we decided to switch to C# due to a better support regarding the implementation of graph-based models. Further, we simultaneously started with engineering the build-time [Bec12] and run-time components [Sch12], and integrating them at a later stage [Spi13].

14.1.2 Architecture and Technology

The developed proof-of-concept prototype comprises a build- as well as a run-time environment (cf. Fig. 14.1). The *build-time environment* provides graphical user interfaces for defining data models, micro and macro process types, and user authorizations. In this context, a "correctness-by-construction" is realized and correctness checks for the various models are provided. All created models are stored in the build-time database. Before enacting object-aware processes, the defined models need to be deployed to the run-time environment. For the run-time database, an application schema is automatically generated based on the deployed models. This database is then used to persistently store instance data. Opposed to the build-time environment, which constitutes a single-user-system, the *run-time environment* is implemented as web application that may be accessed concurrently by various users. Thereby, the run-time environment provides generic implementations for automatically generating all user interface components (i.e., overview tables, forms, and worklists) based on the defined data and process models. It further comprises the logic required to correctly execute the micro and macro process instances; i.e., it fully supports the operational semantics described in Part III.

The first version of the build-time environment was implemented using Eclipse RCP (i.e., Rich Client Platform) and Java [Prö11]. Implementing graph-based models from scratch, however, required high efforts. To reduce development efforts, therefore, we were looking for a framework providing the basic functions for graph-based modeling editors. In this context, we evaluated the JGraph framework and the Graphical Editing Framework (GEF). Opposed to JGraph, GEF is well documented and easy to integrate with the Eclipse RPC. However, with GEF it was not possible to effectively realize the designed usability concept [Wag10].

With the goal of improving usability, we started implementing a new software version using the .NET Framework (i.e., C#) and Visual Studio. The latter enabled a faster development of a user

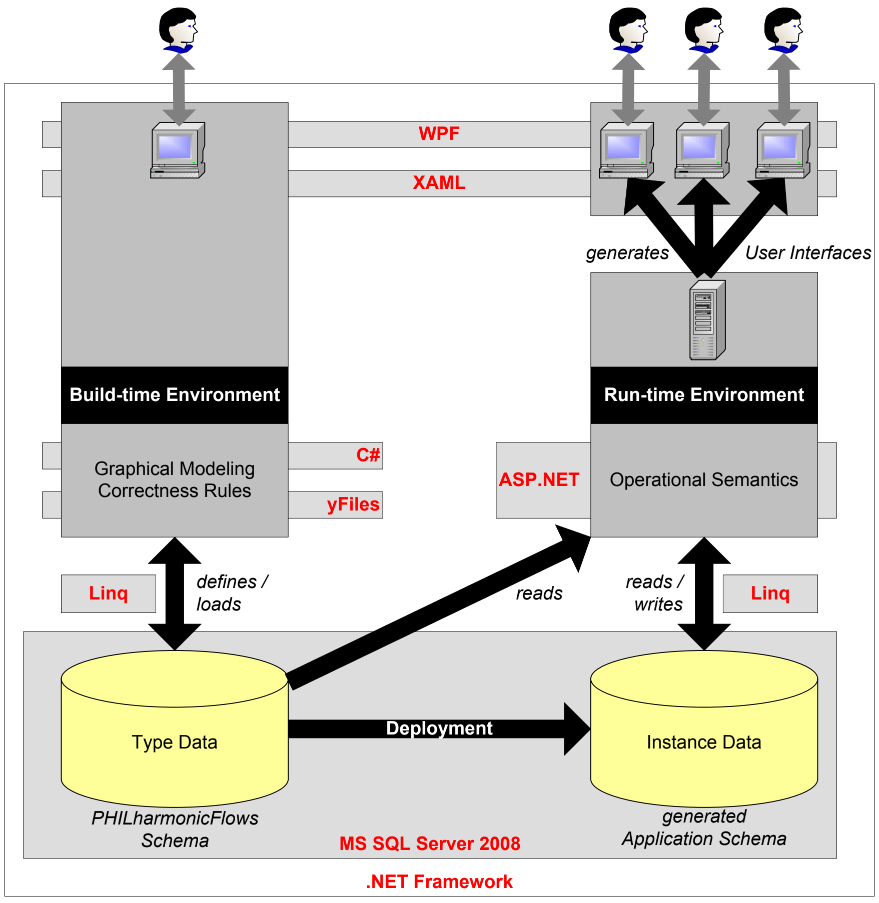


Figure 14.1: Technologies used for the proof-of-concept prototype

interface that complies with the usability concept we designed. This became possible based on WPF (i.e., Windows Presentation Foundation). Its selection was driven by its capability to use the yFiles framework. The latter provides extensive features for modeling graphs. Furthermore, WPF allows defining required user interface components in a declarative way based on XAML (i.e., Extensible Application Markup Language). In addition, XAML allows separating design and behavior. In turn, this enables the re-use of components for different tasks to reduce overall development time.

The main components of the build-time environment are as follows (cf. Fig. 14.2):

- The *build-time DB manager* controls the database connection and provides methods to insert, delete, and change database entries.
- The *permission manager* checks permissions and responsibilities (e.g., which attributes may be accessed by the respective user).
- The *consistency manager* comprises classes for the modeling components (e.g., object types, micro process types), verifies the modeling (i.e., "correctness-by-construction"), and runs correctness checks.

- The *view manager* updates the editors grouped to different workspaces.
- The *project manager* opens, saves, and closes projects.

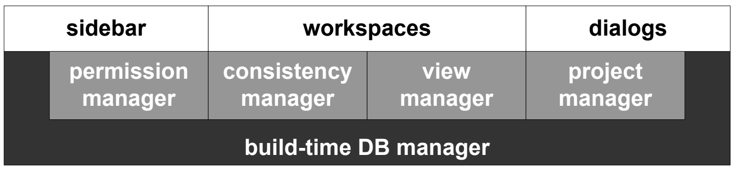


Figure 14.2: Architecture of the build-time environment

The run-time environment is based on ASP.NET (i.e., Active Server Pages), which is a server-side web application framework for realizing dynamic web pages. The latter comprises user-specific data controls. Both the data controls and the web page itself are modified using stylesheets (CSS). With each user call, the web page is newly generated and adapted for the respective browser (without requiring any plugins). Opposed to other server-side script languages (e.g., PHP), which require programming of HTML pages, WPF pages need to be created. The ASP compiler automatically generates a corresponding HTML page with embedded javascript code. This way, it becomes possible to use existing .NET functionalities in the context of a web application as well.

Finally, MS SQL Server 2008 is used for realizing the databases for the build- as well as the run-time environment. We use Linq (Language-Integrated Query) to establish the database connections. This way, it is not required to work with SQL statements, but rather with "get-" and "set-functions". Hence, we can benefit from the advantages offered by object-orientation.

The prototype of the run-time environment comprises several components with different functionalities (cf. Fig. 14.3):

- The *run-time DB manager* communicates with the database of the run-time environment and manages instance data. Therefore, read and write access in provided.
- The *build-time DB manager* communicates with the database of the build-time environment and manages type data. Here, only read access is required.
- The *process manager* controls running micro and macro process instances.
- The *form manager* generates user forms based on object type definitions, permissions and micro process instances.
- The *permission manager* is responsible for the correct integration of user roles; i.e., depending on the currently activated states of the micro process instances, it controls which users own which permissions at a certain point in time.
- The *activity manager* controls which functions may be applied to which object instances.
- The *run-time manager* is responsible for newly created object instances and newly assigned attribute values.
- The *model manager* maintains all models for which instances are currently in use.

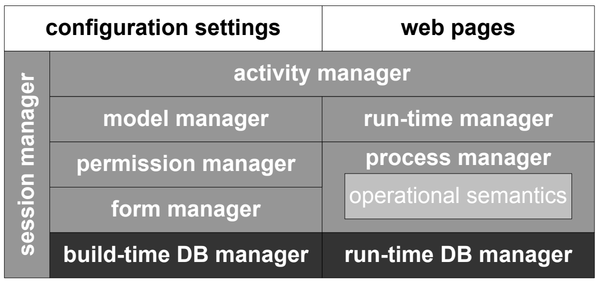


Figure 14.3: Architecture of the run-time environment

14.1.3 End-User View

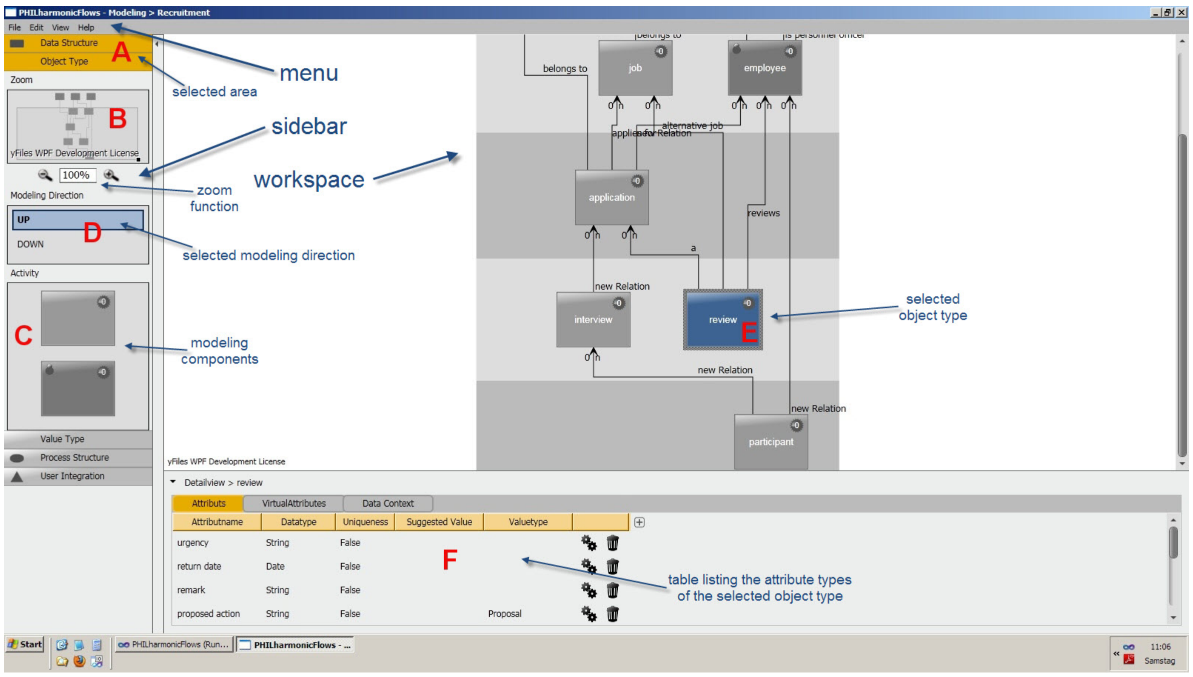
We give an overview of the user interfaces provided by the developed proof-of-concept prototype. To ensure a high usability, we first investigated in user interface design and developed an advanced *usability concept* for modeling object-aware processes with the build-time environment [Wag10] as well as for enacting micro and macro process instances in the run-time environment [Sch10]. All user interfaces ensure a consistent use of colors (i.e., yellow for navigation, red for marking errors, blue for highlighting selections, and green for displaying open tasks), symbols, and fonts.

Build-time Environment

The build-time environment is split into three main parts comprising a *menu*, a *sidebar* and a *workspace*. The sidebar has been realized by applying the "accordion navigation" pattern to the following areas: *Data Structure* (represented by a rectangle), *Process Structure* (represented by an oval), and *User Integration* (represented by a triangle). In addition, each area comprises sub-areas. When selecting an area (cf. Fig. 14.4A), it is expanded and highlighted using the navigation color defined (i.e., yellow). Depending on the selected area or sub-area, additional features are then displayed. For example, consider the zoom function depicted in Fig. 14.4B or the information box depicted in Fig. 14.5A. Finally, the sidebar provides modeling elements (cf. Fig. 14.4C and 14.5B), which may be dragged and dropped into the workspace for defining respective models.

When defining a data model (cf. Fig. 14.4), required object types and user types respectively can be dragged to the workspace and be assigned to different layers (cf. Fig. 14.4C). For modeling object relations, two directions are provided (cf. Fig. 14.4D); i.e., a relation may refer to an object type at a higher level (using direction "UP") or a lower level (using direction "DOWN"). If this results in a cycle, the respective relation is automatically categorized as cycle relation; i.e., resolvesCycle is set to true (cf. Def. 49). When selecting a particular object type (cf. Fig. 14.4E), a table is displayed listing the attributes of the object type. Based on this table, attribute types can be created, edited, or deleted (cf. Fig. 14.4F).

For each object type, a corresponding micro process type needs to be defined. This is illustrated in Fig. 14.5. For selecting an object type, the data structure can be displayed and used as "Structure Compass" (cf. Fig. 14.5C). To distinguish a compass area from a normal workspace, different background colors are used. While normal workspaces are colored white, compass



Detailview > review

Attributs | VirtualAttributes | Data Context

Attributname	Datatype	Uniqueness	Suggested Value	Valuetype
urgency	String	False		
return date	Date	False		
remark	String	False		
proposed action	String	False		Proposal

Figure 14.4: Data modeling in PHILharmonicFlows

areas are grey coloured. When selecting a particular object type (cf. Fig. 14.5D), the corresponding micro process type is displayed at the bottom (cf. Fig. 14.5E).[1] In this workspace, states, micro steps, and micro transitions (including backward transitions) may be defined.

When defining a macro process type (cf. Fig. 14.6), in turn, it is associated with a set of object types (cf. Def. 45). To enable this selection, the "Structure Compass" for navigating within the defined data model is re-displayed (cf. Fig. 14.6A). Furthermore, when defining macro step types and coordination components, additional information about the states of an object type (as defined in the corresponding micro process type) is required. For this reason, respective micro process types are dynamically displayed in the right panel of the compass area (cf. Fig. 14.6B). To define a macro process type, first of all, the corresponding object type must be selected (cf. Fig. 14.6C). Macro step types can then be dragged and dropped to the workspace by selecting them from the sidebar (cf. Fig. 14.6D). In addition, they may be connected using micro transition types. The latter are automatically categorized depending on the relationships of the referred object types (cf. Def. 53). In this context, consider the different colors of the macro transition types as illustrated in Fig. 14.6. Dialogs for specifying the required coordination components are provided when selecting a particular macro transition type.

We omit a description of additional features, like role definition and permission assignment.

Run-time Environment

The user interface of the run-time environment comprises three main parts: a *header* (cf. Fig. 14.7A) displaying login information, a *selection area* (cf. Fig. 14.7B) implemented based on

[1]Note that for each object type a minimal micro process type, comprising a start as well as an end state, is automatically generated.

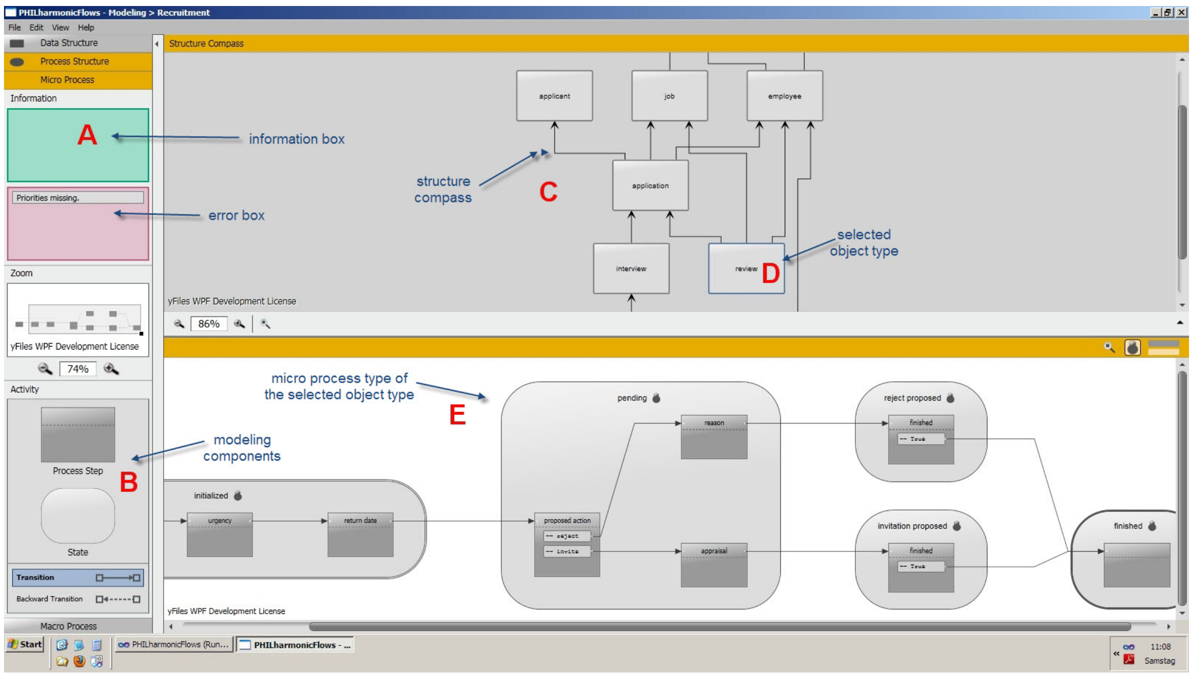


Figure 14.5: Micro process modeling in PHILharmonicFlows

the "accordion navigation" pattern, and a *main workspace* (cf. Fig. 14.7C) displaying overview tables and worklists. Regarding the selection area, a choice among three different user views can be made: *task*, *data*, and *monitoring*.

The *data-oriented view* comprises a panel for selecting the desired object type on the left (cf. Fig. 14.7D) and the overview table listing the object instances (for which the respective user is authorized) on the right (cf. Fig. 14.7E). In addition, the right panel contains a view illustrating the context of the selected object type; i.e., its related object types (cf. Fig. 14.7F). Using the data-oriented view, users can search for object types (cf. Fig. 14.7G), filter object instances (cf. Fig. 14.7H), create new object instances (cf. Fig. 14.7I), and execute optional as well as mandatory activities. Thereby, mandatory activities are listed in the row of the respective object instance (cf. Fig. 14.7J), while optional ones are displayed separately in an instance-specific context menu (cf. Fig. 14.7K). In addition, batch execution is possible (cf. Fig. 14.7L). When selecting a particular object instance, it is possible to navigate through the data structure. In this context, related object instances are displayed below in additional overview tables (cf. Fig. 14.7M).

The *process-oriented user view* (i.e., tasks) is illustrated in Fig. 14.8. It is based on a matrix which represents micro process types and corresponding states as rows, and tasks (see below) as columns (cf. Fig. 14.8A). Each cell of the matrix contains a number counting all object instances (or micro process instances respectively) of the respective category.
The different kinds of task are as follows:

- *Todo:* The respective user either has to

  a) set a mandatorily required object attribute value (cf. Sect. 8.4),

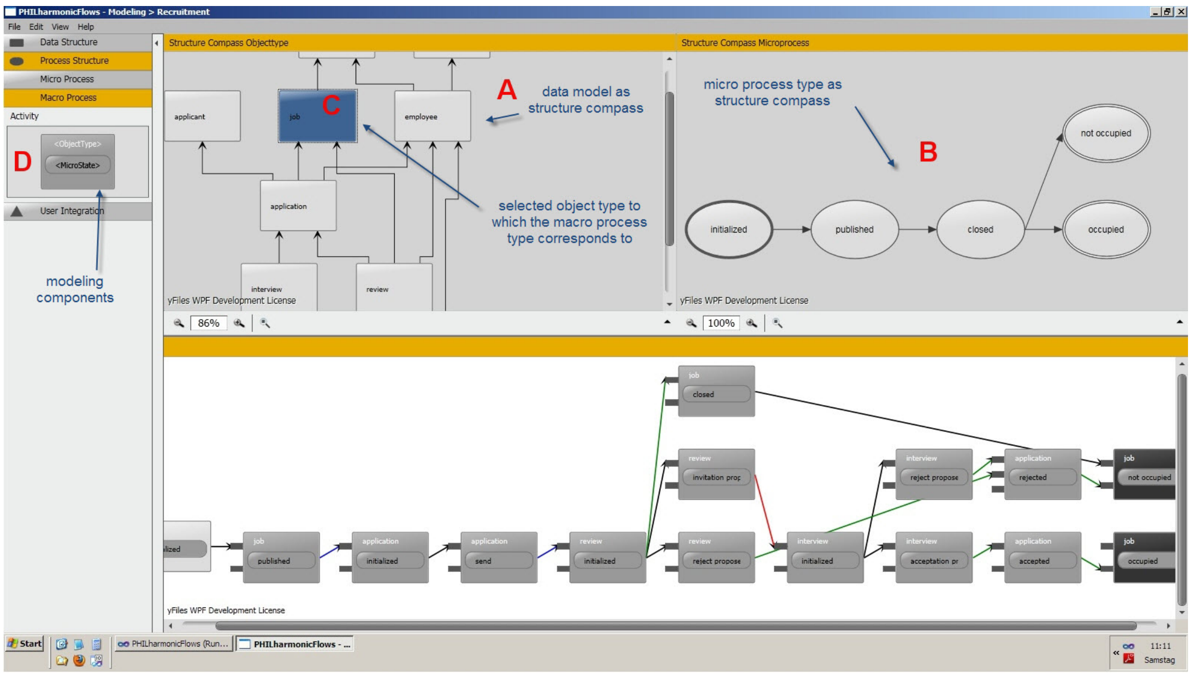

PHILharmonicFlows - Modeling > Recruitment
File Edit View Help
Data Structure
Process Structure
Micro Process
Macro Process
Activity
<ObjectType>
<MicroState>
User Integration
D
modeling components
Structure Compass Objecttype
applicant
job
employee
application
interview
review
C
A
data model as structure compass
selected object type to which the macro process type corresponds to
Structure Compass Microprocess
micro process type as structure compass
B
initialized
published
closed
not occupied
occupied
yFiles WPF Development License
86%
100%
Start
PHILharmonicFlows (Run...
PHILharmonicFlows - ...
11:11
Samstag

Figure 14.6: Macro process modeling in PHILharmonicFlows

   b) create a corresponding lower-level object instance, if the minimum cardinality has not been reached yet (cf. Sect. 9.1),

   c) commit a state change (cf. Sect. 8.5), or

   d) execute mandatory black-box activities (cf. Sect. 9.2).

- *Responsible:* The respective user may track states of the micro and macro instances, he or she is responsible for (cf. Sect. 12.9).

- *Error:* The respective user either has to

   a) check inconsistent object instances (cf. Sect. 13.2),

   b) handle bypassed micro process instances (cf. Sect. 12.9), or

   c) dissolve a deadlock (cf. Sect. 13.2).

Based on this matrix, a user may select a cell belonging to a micro process type or one of its states as well as a desired task (cf. Fig. 14.8B). Following this, a corresponding overview table is displayed in the bottom, listing all object instances of the selected category (cf. Fig. 14.8C). Such an overview table can be considered as worklist comprising a number of object instances for which (the same) mandatory activities must be executed. Note that batch execution is possible in this context as well (cf. Fig. 14.8D).

Regarding the monitoring view (cf. Fig. 14.9), object instances representing a macro process instance (of a selected macro process type) are listed in an overview table (cf. Fig. 14.9A). In this context, activities for handling macro process instances (and micro process instances respectively) are provided; e.g., consider the activity for skipping a micro process instance

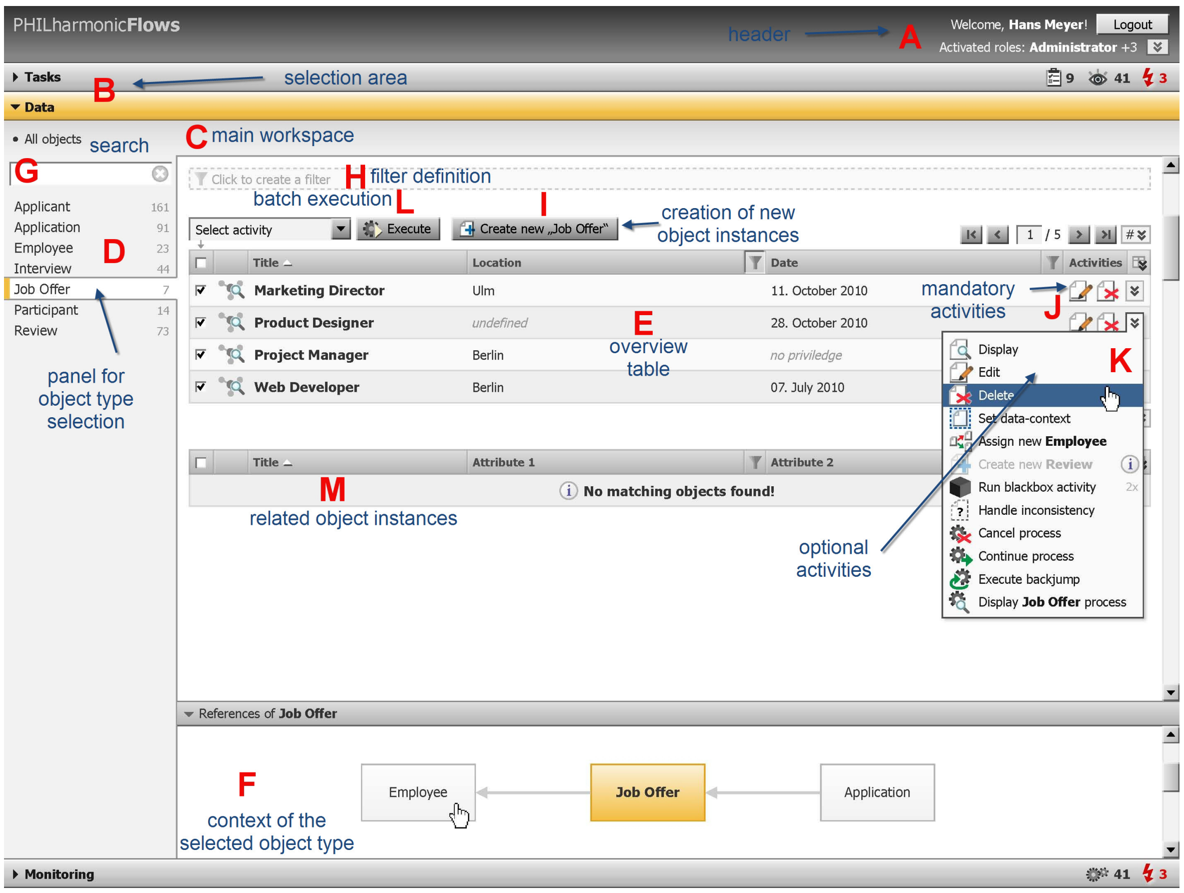


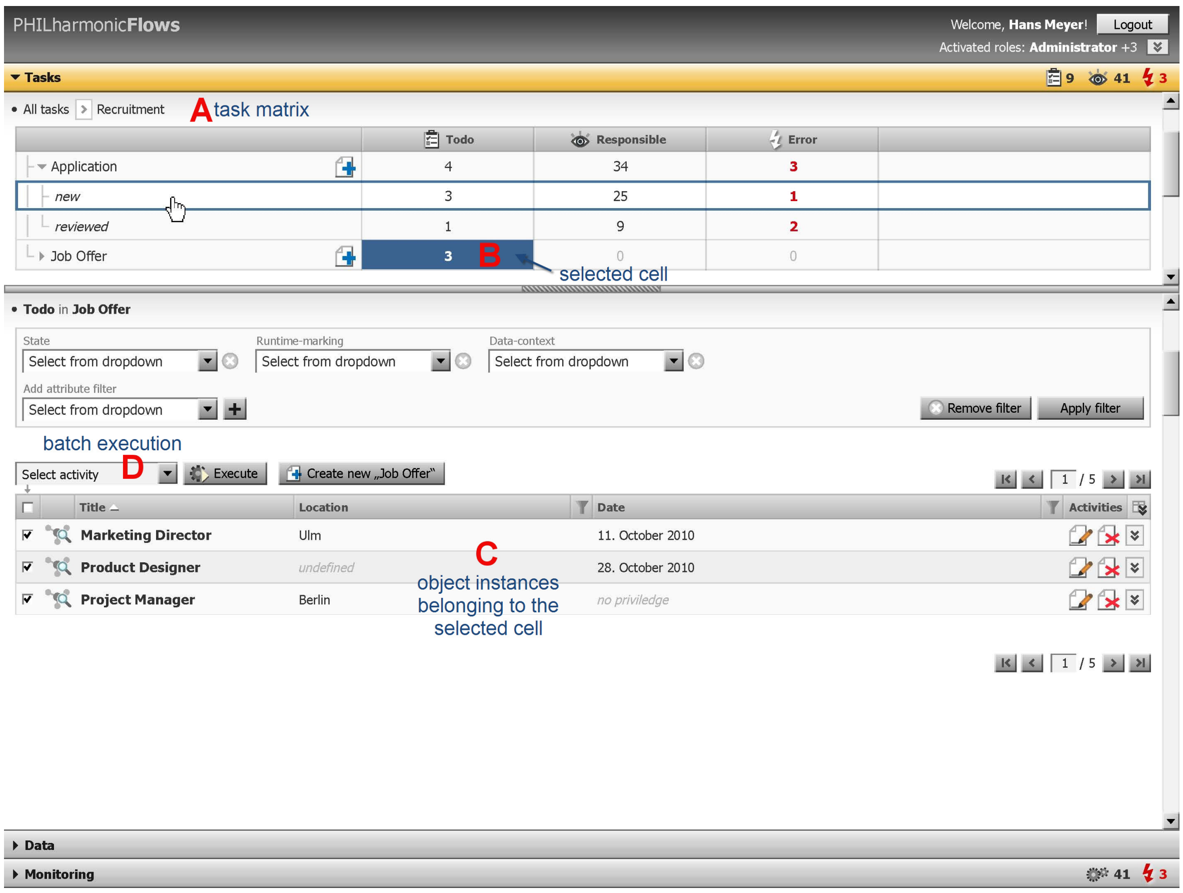
Figure 14.7: Data-oriented user view

in Fig. 14.9B or for dissolving deadlocks in Fig. 14.9C. In addition, using the popup menu displayed in Fig. 14.9D, users can navigate to micro process instances related to the currently selected macro process instance. When selecting a macro process instance (cf. Fig. 14.9E), a special monitoring view (cf. Fig. 14.9F) is displayed visualizing it. A macro step may be expanded (cf. Fig. 14.9G) for displaying the various instance counters (cf. Def. 59). For example, counter "in" indicates for how many micro process instances the respective state is currently activated. When selecting a counter, an overview table is generated that contains the corresponding micro process instance. This way, it becomes possible to navigate through the whole process structure.

Furthermore, based on macro transitions, deadlocks can be visualized (cf. Fig. 14.9H). When selecting a macro transition, a special view is generated displaying the invoked micro process instances together with their coordination components. Based on this view, features for dissolving deadlocks can be provided (cf. Sect. 13.2).

14.1.4 Summary and Discussion

The developed prototype shows the technical feasibility of the concepts presented in Part III. Moreover, it helps to visualize the concepts in a comprehensible and consistent way. Thus, our



	Todo	Responsible	Error	
Application	4	34	3	
new	3	25	1	
reviewed	1	9	2	
Job Offer	3	0	0	

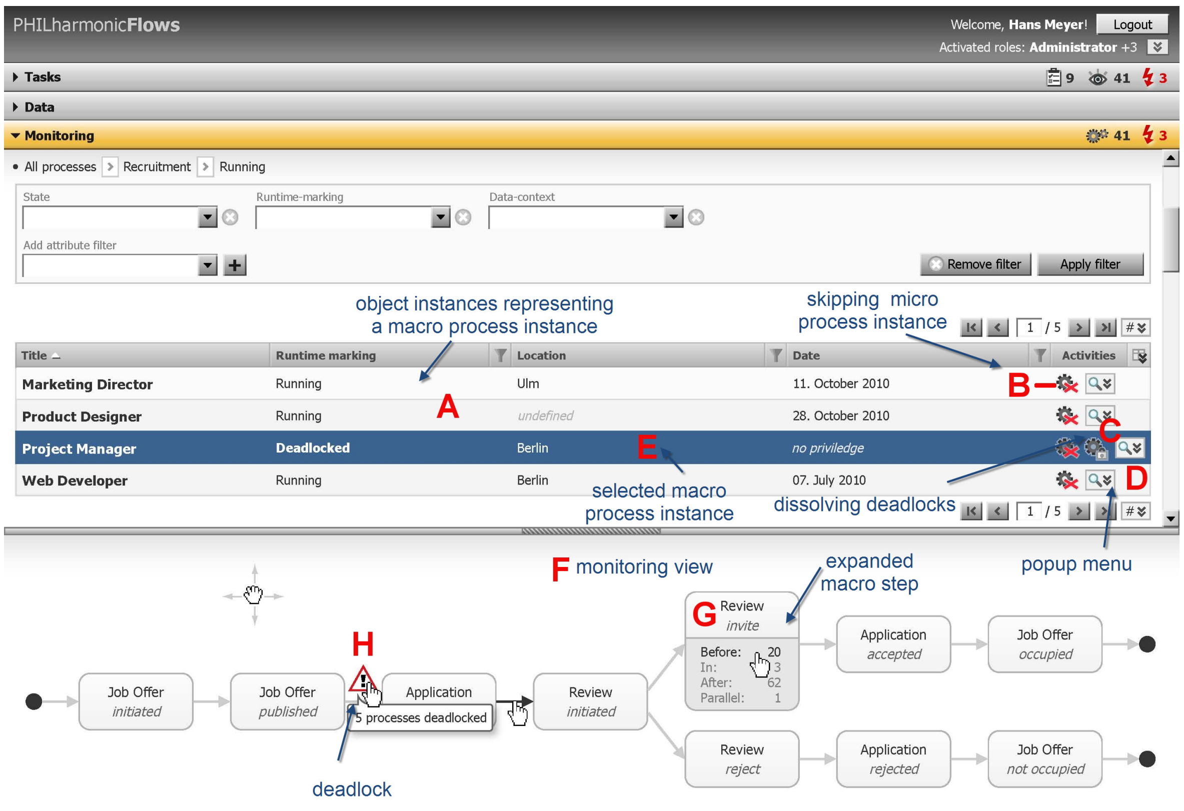
	Title	Location	Date	Activities
✓	Marketing Director	Ulm	11. October 2010	
✓	Product Designer	*undefined*	28. October 2010	
✓	Project Manager	Berlin	*no priviledge*	

Figure 14.8: Process-oriented user view

preparing investment into the design of a sophisticated user interface has paid off [W10, S10]. The prototype is still under development. In further work, we will use it to realize additional features and concepts (e.g., schema evolution [CKR12a]).

14.2 Practical Application

This section gives additional impressions on our proof-of-concept prototype and illustrates its application to characteristic cases to validate the concepts developed. This way, we aim at a verification of the usability of the PHILharmonicFlows approach in practice. For this purpose, PHILharmonicFlows is applied to the medical domain (cf. Sect. 14.2.1), an extension course proposal (cf. Sect. 14.2.2), a vacation request (cf. Sect. 14.2.3), and a house building scenario (cf. Sect. 14.2.4). Finally, we discuss the benefits of our framework as well as the lessons learned along these process scenarios in Sect. 14.2.5.



Title	Runtime marking	Location	Date	Activities
Marketing Director	Running	Ulm	11. October 2010	
Product Designer	Running	undefined	28. October 2010	
Project Manager	Deadlocked	Berlin	no priviledge	
Web Developer	Running	Berlin	07. July 2010	

Figure 14.9: Monitoring

14.2.1 Medical Domain

As healthcare scenario we consider a *breast cancer diagnosis process* as described in the process handbook of a Women's hospital (see [CKR12b] for details).

The breast cancer diagnosis process comprises an anamnesis, a physical examination (including the collection of symptoms), a set of medical examinations (e.g., MRI, mammography, blood analysis), and a tumor biopsy. During the anamnesis the doctor asks the patient specific questions; e.g., about her history of diseases, family diseases, or current medication. The doctor examines the patient and checks her for the presence of any symptom. The doctor asks the patient about breast nodules and performs a physical examination in order to confirm or exclude the symptoms. If the symptoms brought up by the patient are not confirmed during the physical examination, the presence of the tumor will be denied. In this case the diagnosis process is finished. Otherwise, the doctor decides about a battery of examinations based on the symptoms confirmed. Examinations required to detect the presence of a breast tumor or to exclude it are mammography and MRI examination. A mammography must be scheduled by the secretary of the radiology department. At the day of the examination, the mammography is performed and the resulting images become available. The images from both examinations are then analyzed by a specialized doctor of the radiology department and added to the respective medical reports. As opposed to the mammography examination, for which the equipment does not cause claustrophobia, during the MRI examination the patient may have a case of elevated anxiety due to the enclosure of the MRI equipment. In such cases, the radiology specialist, responsible for the examination, must decide whether or not the patient shall be sedated before continuing with the procedure. In the meanwhile, the doctor may request further examinations; e.g., another MRI examination or additional blood tests. Otherwise, if the existence of a tumor is confirmed, the doctor may want to biopsy this mass in order to confirm the malignancy of the tumor. In this case, however, the consent of the patient is required. The biopsy report is returned to the doctor who will inform the patient about the malignancy status of the tumor. Finally, the diagnosis process is finished as positive, confirming the presence of a breast tumor.

The breast cancer diagnosis process as whole interacts with numerous smaller processes like MRI examinations or mammographies. The execution of the diagnosis process depends on the results of various examinations; e.g., data from the MRI examination is needed. Some examinations are mandatory (e.g., anamnesis, patient information), while others are optional (e.g., MRI, mammography, blood analysis, and tumor biopsy); i.e., it depends on the decision of the responsible doctor which examinations shall be executed. For example, a radiology specialist decides whether or not to sedate a patient during an MRI examination. Like most healthcare processes, the sketched scenario is characterized by a large number of medical forms to be filled by authorized medical staff (e.g., doctors, nurses, and laboratory staff) with information relevant for patient treatment. For example, consider the information obtained when interviewing the patient about her anamnesis.

We illustrate how are modeled this scenario using PHILharmonicFlows.

Data Model. The data model of the breast cancer diagnosis process is depicted in Fig. 14.10. There is one object type for each of the phases of the diagnosis process. Thereby, the cardinality of object type `anamnesis` in relation to object type `diagnosis` corresponds to 1; i.e., there must be exactly one instance of object type `anamnesis` for each `diagnosis` instance. Opposed to this, the existence of an instance of object type `mammography` is not mandatory for a given `diagnosis` instance. Furthermore, it is up to the respective doctor to initiate a specific number of instances of this `examination` as long as cardinality constraints are fulfilled.

Micro Process Types. An example of a micro process type is depicted in Fig. 14.11. In order to request a `mammography`, an authorized user must set the `order date`; i.e., to complete micro step `order date` a value needs to be assigned to the corresponding attribute. In our example, the micro transition type between state types `requested` and `scheduled` is explicit. This ensures that the doctor may still review the examination request before sending it to the secretary of the radiology department. In turn, in state `scheduled`, the secretary must fill attributes `scheduled date`, `doctor`, and `room`. Further she has to decide when to notify the patient about the scheduled appointment; i.e., the subsequent state `patient notified` will only be activated if this is explicitly confirmed by the secretary.

Fig. 14.12 shows a fragment of the `MRI` micro process type; here the radiology specialist must decide whether or not to sedate the patient.

Macro Process Type. The macro process type begins with the creation of an instance of object type `diagnosis`, which triggers the creation of the micro process instance. Then, object type `anamnesis` is instantiated. During `patient examination`, `symptoms` are `collected`, which are then `confirmed` after the `physical examination` has taken place. If the symptoms are `not confirmed`, the `diagnosis` will be finished as `negative`, indicating that no tumor was found. Otherwise, the

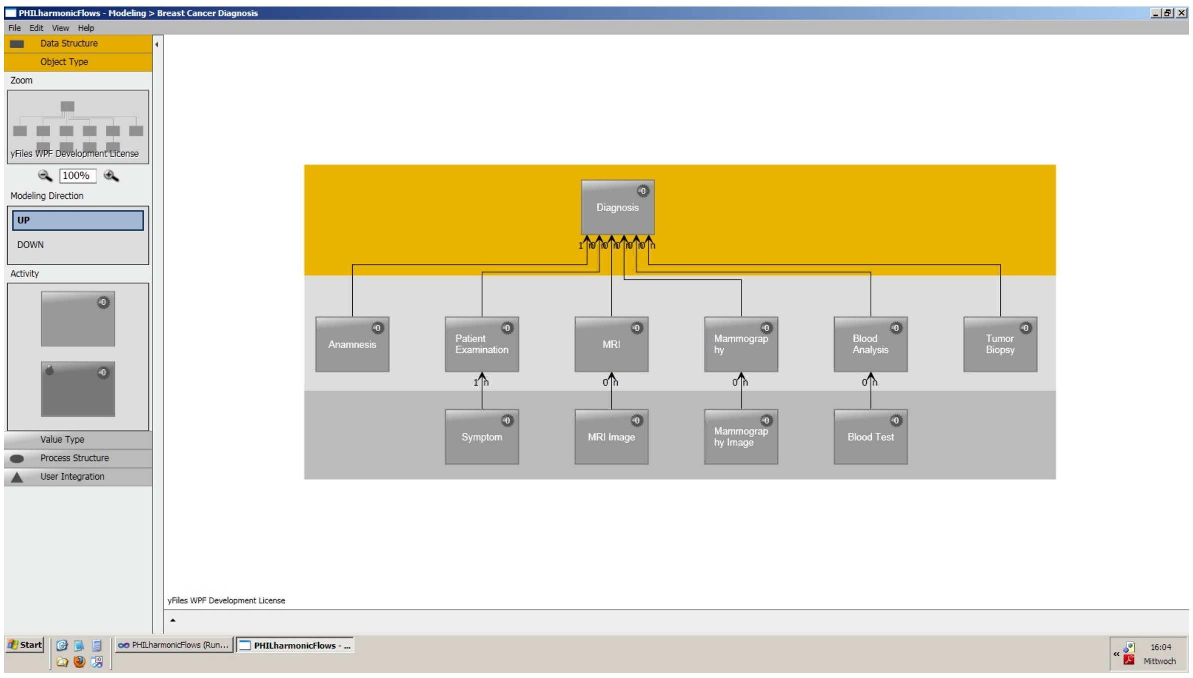


Figure 14.10: Breast cancer data model

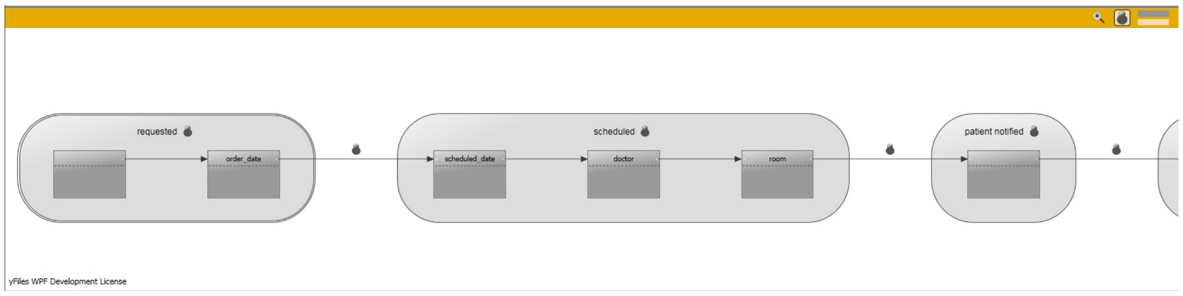


Figure 14.11: Mammography micro process type

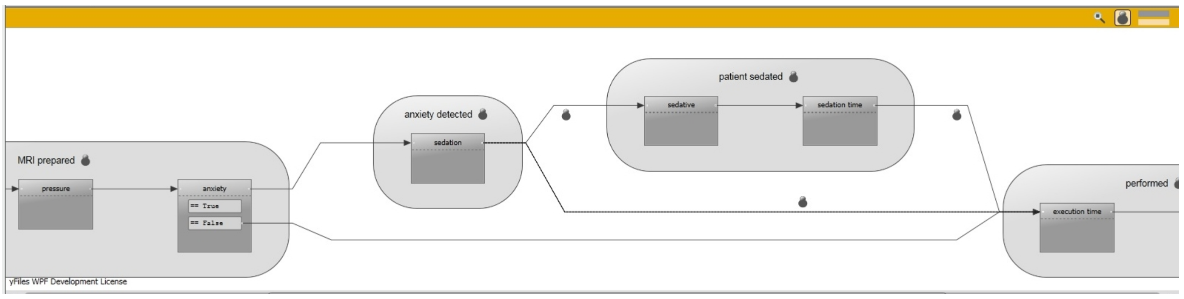


Figure 14.12: MRI micro process type

`diagnosis` process continues with the request of further examinations (e.g., `mammography`, `blood analysis`, `MRI`, etc.).

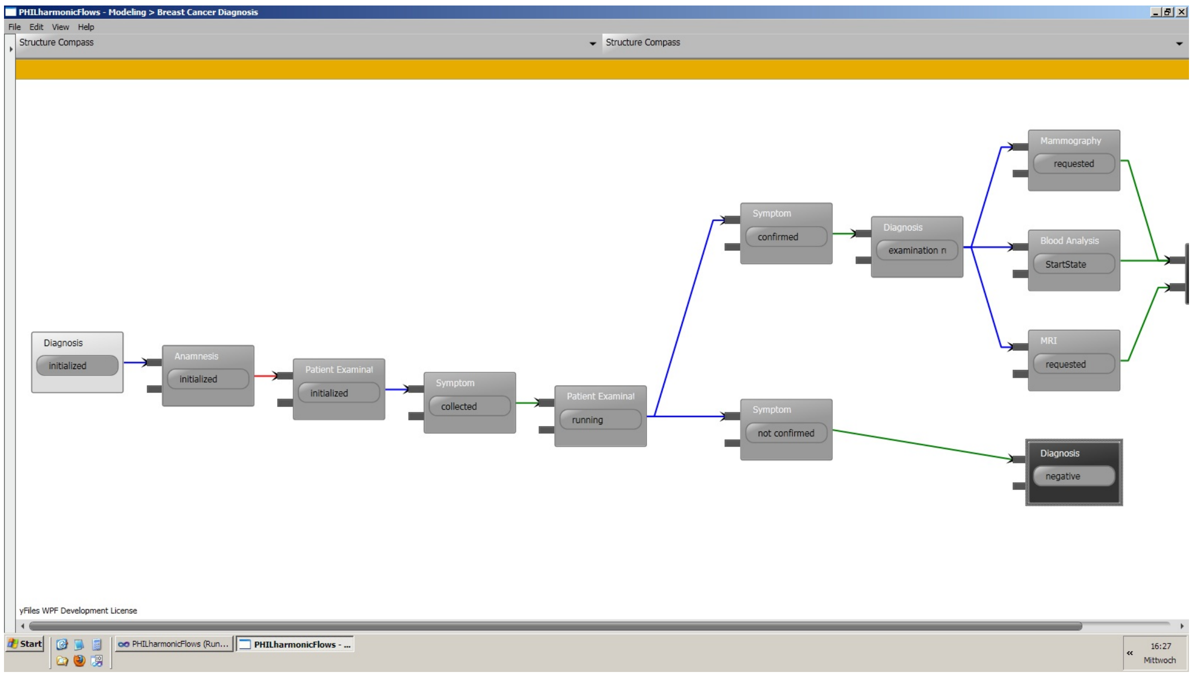


Figure 14.13: Breast cancer macro process type

Overall, the methodology provided by PHILharmonicFlows ensures that each procedure (e.g., anamnesis, primary examination, mammography, etc.) is modeled from a data-oriented perspective (i.e., object types) as well as from a process-oriented one (i.e., micro process types).

14.2.2 Extension Course Proposal

Another scenario to which we applied the development methodology deals with proposing *extension courses* at a university; i.e., courses for professionals that aim at refreshing and updating their knowledge in a certain area of expertise. To propose an extension course, the course coordinator must create a project describing it. The latter must be approved by the faculty coordinator as well as the extension course committee. The result of this case study is discussed in detail in [CKR12a].

The course coordinator creates an extension course project using a form. In this context, he must provide details about the course, like name, start date, duration, and description. Following this, professors may start creating the lectures for the extension course. In turn, each lecture must have detailed study plan items, which describe the activities of the lecture. To each lecture, (external) invited speakers may be assigned. The latter either may accept or reject the invitation. After receiving the responses for these invitations and creating the lectures, the coordinator may request an approval for the extension course project. First, it must be approved by the faculty director. If he wants to reject it, he must provide a reason for his decision and the course must not take place. Otherwise, the project is sent to the extension course committee, which will evaluate it. If there are more rejections than approvals, the extension course project is rejected. Otherwise, it is approved and hence may take place. In the following, we illustrate the modeling of this scenario using PHILharmonicFlows.

Data Model. Fig. 14.14 illustrates the data model. Object types `lecture` and `decision committee` refer to object type `extension course`. In turn, object types `invitation` and `study plan item` refer to `lecture`. At run-time, these relations allow for a varying number of interrelated object instances whose processing must then be coordinated.

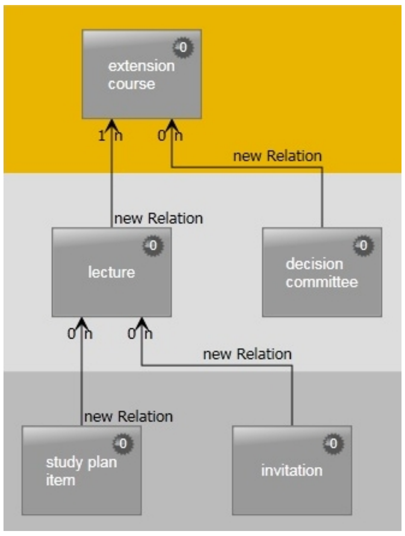


Figure 14.14: Extension course data model

Micro Process Type. Fig. 14.15 shows the micro process type related to object type `extension course`. While the `extension course` is in state `under creation`, the course coordinator may set the attributes the corresponding micro step types refers to (e.g., `name`, `start date`, and `faculty`). Following this, a user decision is made in state `under approval`; i.e., the faculty director either approves or rejects the `extension course`. If the value of attribute `decision` corresponds to rejected, a value for attribute `reason` is required.

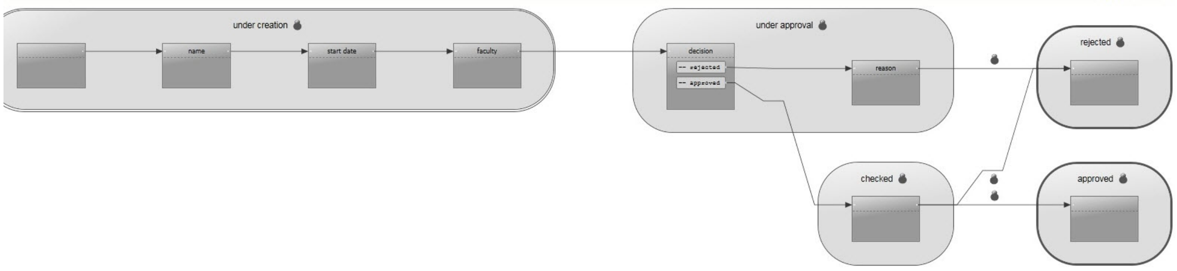


Figure 14.15: Extension course micro process type

The macro process type for extension courses is depicted in Fig. 14.16.

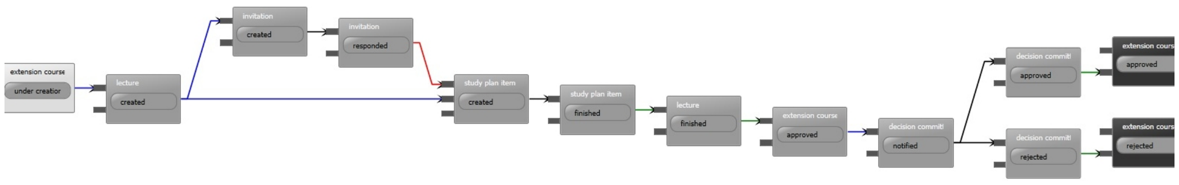


Figure 14.16: Extension course macro process type

14.2.3 Vacation Request

This case relates to vacation requests [KR11b], i.e., it deals with short running processes. Therefore, we use it to provide a more sophisticated example concerning the modeling of micro processes (cf. Fig. 14.17).

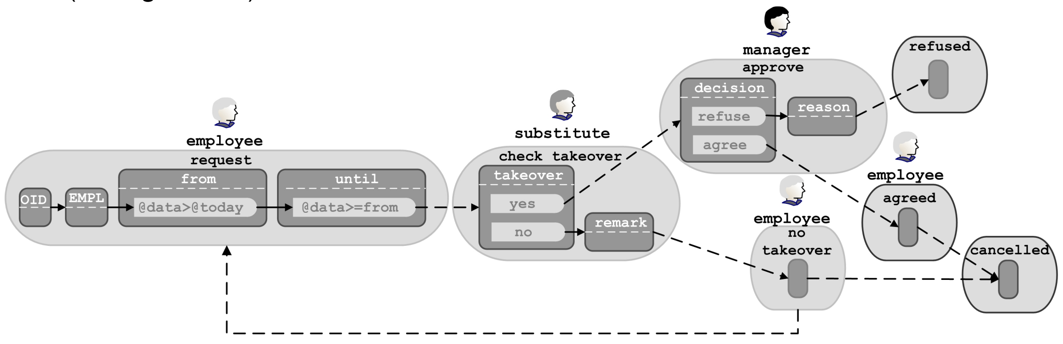


Figure 14.17: Vacation request micro process type (according to [KR11b])

Micro process type. An employee applying for a leave must specify a start as well as an end date. In this context, two micro steps are introduced to ensure that the start data lies in the future and the end data follows the start date; both micro steps are assigned to the same state. Following this, the employee submits her request to her substitute who must decide whether he is willing to take over. If he agrees the request is submitted to the manager of the employee. She must decide whether to agree with the request or to refuse it. If the substitute is not willing to take over, the employee may decide whether to re-submit the request or cancel it. The latter is also possible for already agreed upon requests.

14.2.4 House Building

This scenario deals with a complex, long running process that involves numerous object types. For this reason, we use it for evaluating the modeling of macro processes in more detail [KR11b]. Figs. 14.18, 14.19, and 14.20 depict extracts of the data model as well as the macro process type we created for this case.

Data model. As illustrated in Fig. 14.18, each house is built on a plot belonging to a town. The plot must be bought; i.e., a sale contract between the town and builder is required. The house is built up on basis of a construction plan, which must be approved by the town. It further consists of brickwork comprising a base plate, stonework, a roof truss, and a frame carpentry. In order to build-up higher floors and the roof, a scaffold is required. Finally, the interior finish comprising installation, electrican stuff, painters and so on.

Macro process type. Consider the corresponding macro process type depicted in Fig. 14.19. Before the brickwork can be started, the sale contract must be signed and the construction plan be approved. The sub-parts belonging to the brickwork must be constructed in sequence within a narrow time frame; i.e., the construction of a particular sub-part depends on the construction of other subparts (of different type). In particular, roof construction may only be started when the frame carpentry has finished the sub-structure of the roof (cf. Fig. 14.20). Consequently, the macro transitions which connect the macro steps relating to these individual sub-parts refer to transverse relationships.

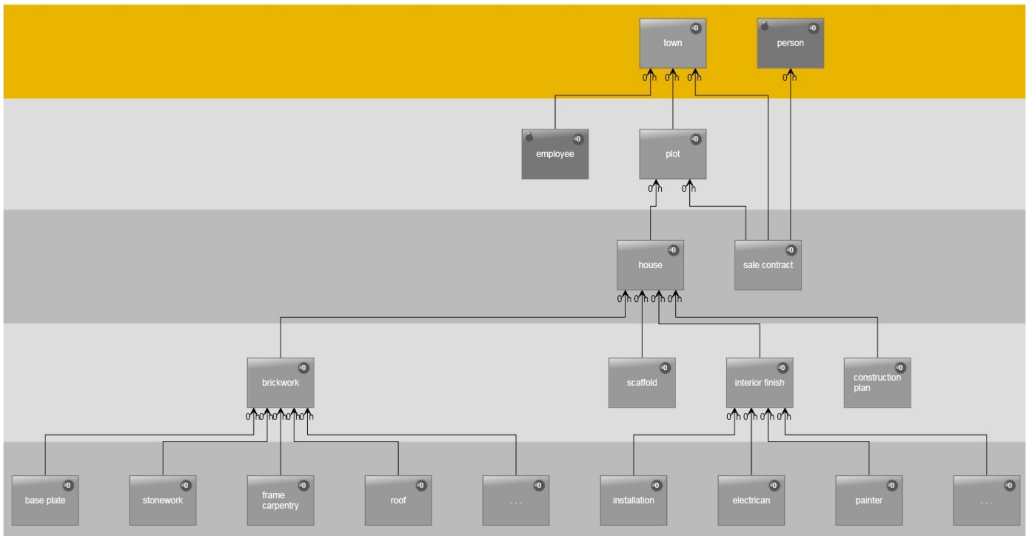


Figure 14.18: House building data model

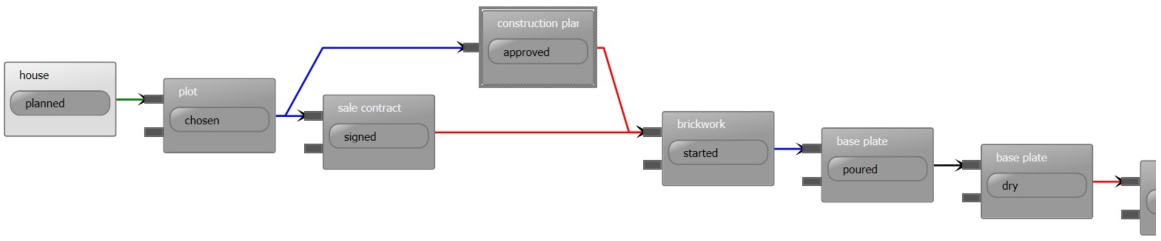


Figure 14.19: House building macro process type extraction 1

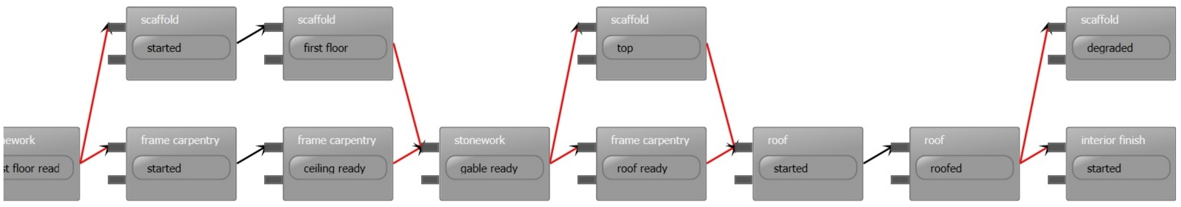


Figure 14.20: House building macro process type extraction 2

14.2.5 Summary and Discussion

As another use case not discussed in detail in this thesis, consider "Bricolage", an online shop which enables business owners to sell their products. This scenario is discussed in [Wag10]. The considered process includes several steps ranging from the selling of products to their delivery to customers.

Applying our proof-of-concept prototype and the developed concepts to the above process scenarios has proven that our basic modeling approach works well. Amongst others, it confirmed that the unambiguous granularity of processes allows for an unambiguous modeling methodology. The modeling of micro processes, which comprise micro steps defined in terms of object attributes, enabled the required data-based modeling. The ordering of attributes as well as their grouping to states further turned out to be very intuitive for modelers. Process coordination based on object types and their corresponding states has proven to be extremely useful as well. In particular, it provided an adequate and comprehensible level of abstraction to users. The "flat" way of modeling processes additionally facilitated modeling. Finally, the different kinds of relationships offered in respect to process coordination contributed to hide the complexity of the actual process structure from both modelers and end users.

At run-time, the integrated view on processes and data offers numerous benefits. For example, state-based process monitoring allows for a more natural and intuitive view on business processes for end-users. Since activities are not pre-fixed and are not rigidly associated with one process, a more flexible process execution becomes possible. First, through the execution of optional activities authorized users may accomplish certain actions up-front, i.e., before they are mandatorily required in the course of process execution. Similarly, completed activities may be re-executed if desired. Second, batch activities improve the processing of a large number of object instances.

However, our evaluation has also revealed some limitations of our approach to be tackled in future work. For example, the historization of attribute values and state changes was considered as important feature for enabling proper process traceability. In addition, advanced features like time events should be supported as well. Furthermore, different specializations of an object type may require different models for related micro processes; i.e., we must cope with process variability in the context of object-aware processes as well (see [HBR10a] and [HBR10b] for how process variability is handled in traditional approaches). For example, job applications belonging to different locations may require different process definitions. Other scenarios, in turn, required an overlap and synchronization of different macro process instances. For example, consider the interdependencies between purchase orders and warehousing processes. Further, this thesis focuses on the modeling and the execution phase in process lifecycle (including exception handling). Additional challenges, however, emerge from the support of ad-hoc changes during run-time as well as from schema evolution. Regarding the latter, major requirements are discussed in [CKR12a]. In this context, the evolution of data models needs to be compliant with the one of the respective process models.

15 Summary and Outlook

Providing integrated access to business processes, business data, and business functions for users constitutes a fundamental goal of any business application. While upcoming tasks must be assigned to the right actors at the right point in time, users should be able to flexibly access relevant business information and business functions at any point in time presuming they have proper authorizations for this. On one hand, business information should include information about business processes, on the other, adequate context information is required during process execution.

Traditional PrMS allow for a strict separation of concerns; i.e., data, functions, and processes are managed by different kinds of information systems. In turn, this makes it almost impossible to provide integrated and consistent access to these different artifacts. For exactly this reason, many processes supported by existing business applications (e.g., ERP systems) are still hard-coded; i.e., their implementation requires programming efforts and due to the missing generic process support it is not possible to automatically generate worklists and other artifacts during run-time.

This thesis has shown that, based on a tighter integration of the different business perspectives, many of the several limitations of existing PrMS may be overcome. In particular, such an integration will help to provide more generic business software being able to automatically and dynamically generate end-user and application components at run-time based on the specified process and data mmodels.

15.1 Contribution

This thesis has conducted both *natural research* and *design research*, and thus made two major contributions. First, it has been shown that *"object-awareness"* is fundamentally required in order to provide generic support for the business processes, which are usually hard-coded in existing business application. In particular, we identified the major characteristics of object-aware processes in this context. Second, we developed the *PHILharmonicFlows framework*, which targets at a comprehensive support of object-aware processes.

15.1.1 Object-awareness

As shown in this thesis, the relationships between the different business perspectives have not been well understood so far. Hence, as a first major contribution of this thesis, we conducted natural research in order to identify the mutual relationships that exists between these perspectives. For this purpose, we elaborated the key challenges for advanced process management, which we denote as *object-aware process management*. By introducing the notion of object-awareness, a comprehensive understanding of the relationships that exist between business processes, data, functions, and users is provided.

In summary, object-awareness can be described by the following five fundamental characteristics:

1. *Object behavior:* The behavior of the business objects involved in a business process must be taken into account during process execution.
2. *Object interactions:* Interactions between business objects must be adequately handled; i.e., the behavior of individual objects must be coordinated with the one of related objects.
3. *Data-driven execution:* Since the progress of a business process mainly depends on available business objects and their attribute values, process execution should be accomplished in a data-driven manner.
4. *Integrated access:* Authorized users must be able to access and manage process-related business objects at any point in time (presuming they have proper authorizations for this).
5. *Flexible activity execution:* Activities should be executable at different levels of granularity. While a particular user may want to work on a certain object instance, another one may desire to process a number of related object instances in one go.

Our analyzes revealed that contemporary PrMS have not achieved the technological maturity yet for adequately supporting these fundamental characteristics in an integrated and consistent way. Instead, a *more advanced process modeling paradigm and methodology* is required which enables a tight integration of all business perspectives. Furthermore, process execution should no longer be solely activity-driven. Instead, a *data-driven execution paradigm*, combined with activity-oriented aspects, is required. Finally, since activities may have flexible granularity, individual activities must no longer coincide with particular process steps like in traditional PrMS.

15.1.2 The PHILharmonicFlows framework

As second major contribution of this thesis, we developed the PHILharmonicFlows framework. This framework provides a comprehensive approach for supporting unstructured, data-driven processes, as they can be found in many contemporary business applications. The fundamental idea of the framework is to provide a generic component that allows realizing similar features as can be found in hard-coded business applications in a much more effective and efficient manner on one hand, and which benefits from the approach taken by PrMS on the other. The framework enables data- as well as process-oriented views in an integrated and consistent way. Using generic components, these views as well as form-based activities can be automatically generated at run-time. Opposed to existing approaches targeting at an improved integration of processes and data, PHILharmonicFlows covers all characteristics of object-awareness in an integrated and comprehensive way. Further, it enables process modelling and provides a precise and well-defined operational semantics for executing business processes at different levels of granularity. Based on the formal semantics provided, in addition, end-user components can be automatically generated at run-time. Besides worklists and overview tables, PHILharmonicFlows considers the high number of individual user forms. This way, activity execution is decoupled from particular process steps and process instances respectively. In turn, an implementation is required for black-box activities, which allow for more complex computations or the integration of legacy applications.

As a prerequisite for any integrated access, a *data model* must be provided. With PHILharmonicFlows it becomes possible to define *object types*, *object attributes*, and *object relations*. Regarding the latter, minimum and maximum *cardinalities* may be assigned. To enable process support, PHILharmonicFlows enforces a well-defined modeling methodology, which differentiates between *micro and macro processes* to cover both object behavior and object interactions. Regarding *object behavior*, existing approaches can be divided into two groups. The first one allows for a state-based modeling, combined with an activity-driven execution paradigm. Opposed to this, the second group follows a data-driven execution paradigm, but without considering states. The PHILharmonicFlows approach applies the well established concept of modeling object behavior in terms of *states* and *state transitions*. Opposed to existing work, however, a *mapping between attribute values and objects states* is established. This ensures compliance between them and enables a data-driven process execution at run-time. Thus, for the first time, a *state-based modeling approach* is combined with a *data-driven execution paradigm*.
In particular, for each object type a corresponding *micro process type* comprising a number of states needs to be defined. Each state comprises a number of *micro steps*. In turn, a micro step refers to an attribute and describes an atomic action for writing it. By connecting micro steps with *micro transitions*, we obtain their default execution order. At run-time, a micro step is completed when a value becomes available for the corresponding attribute. While states are used to coordinate the processing of an object instance between different users, micro steps capture the internal logic of activities.
To enable *optional activities*, a sophisticated *authorization table* is automatically generated for each object type. Based on it, different permissions for reading and writing attribute values as well as for creating and deleting object instances are granted to user roles. Thereby, PHILharmonicFlows considers the different states as well. Overall, object attributes, user permissions, and the micro logic defined, build the foundation for automatically generating user forms at run-time.
Whether or not subsequent states may be reached also depends on the execution of other process instances; i.e., micro process instances of the same and of different type must be synchronized with each other. Regarding process synchronization mechanisms, PHILharmonicFlows provides ground-braking concepts. First of all, process coordination is based on states rather implemented based on message exchanges. On one hand, this allows for the *asynchronous execution of individual process instances*, on the other, *sophisticated aggregation concepts*, which consider the cardinalities existing between object instances, are applied. Moreover,

PHILharmonicFlows hides the complexity of large process structures from modelers as well as from end-users to a large extend. For this purpose, *macro processes* can be modeled in a flat and compact way, comprising process steps and process transitions as known from existing process modeling paradigms. As opposed to traditional process modeling approaches, where process steps are defined in terms of activities, a *macro step* always refers to an object type together with a corresponding state. To take the dynamically evolving number of object instances as well as their asynchronous execution into account, for each *macro transition*, a corresponding *coordination component* needs to be defined. Based on coordination components, aggregation conditions as well as asynchronous process execution can be realized.

15.2 Benefit

PHILharmonicFlows enables a very flexible integration of business data, functions, and processes, and thus overcomes many of the limitations known from activity-centered PrMS. Logically, the latter rely on a number of pre-defined processing states. Thereby, the defined control flow and the activities executed determine which of these states may be reached. In turn, all other processing states are prohibited by the defined process model (cf. Fig. 15.1a).

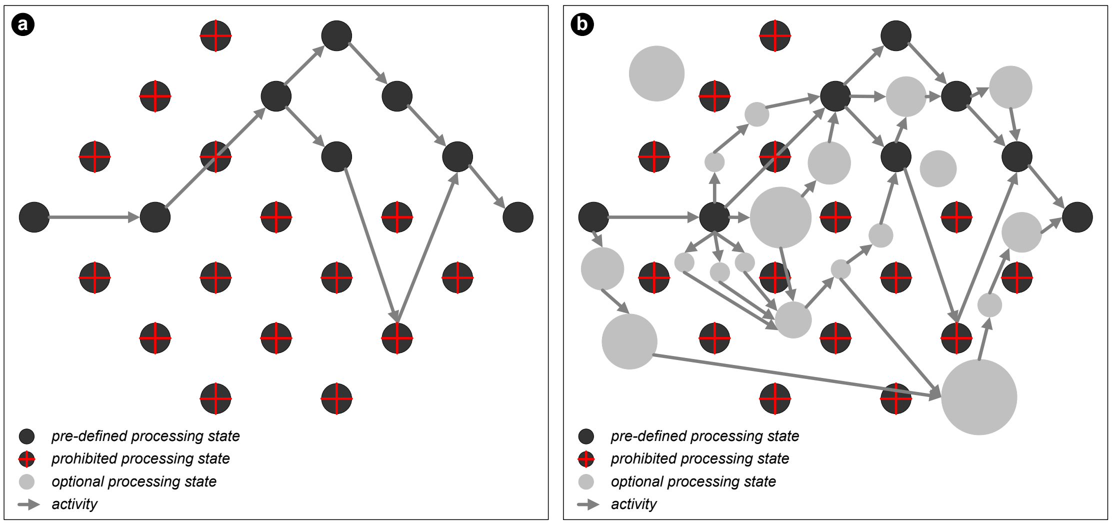

a
b
pre-defined processing state
prohibited processing state
optional processing state
activity

Figure 15.1: Process execution in (a) traditional and (b) object-aware PrMS

Opposed to this rigid behavior, object-aware processes, as supported by PHILharmonicFlows, allow for optional processing states, which enable users to access and manage business data independent from actual process execution (cf. Fig. 15.1b). In this context, it becomes possible to apply different ways of working as well as activities of different granularity to reach the overall process goal. Thus, users may freely choose their preferred work practice. Particularly, a knowledge-driven process execution enabling better user assistance is provided; i.e., while a certain user may prefer doing a lot of work in the context of a particular activity (cf. Fig. 15.2a), another one may prefer performing several activities for achieving the same results (cf. Fig. 15.2b).

Accordingly, the activities actually performed may vary from process instance to process instance (cf. Fig. 15.3). Particularly, this fosters the execution of unstructured or semi-structured

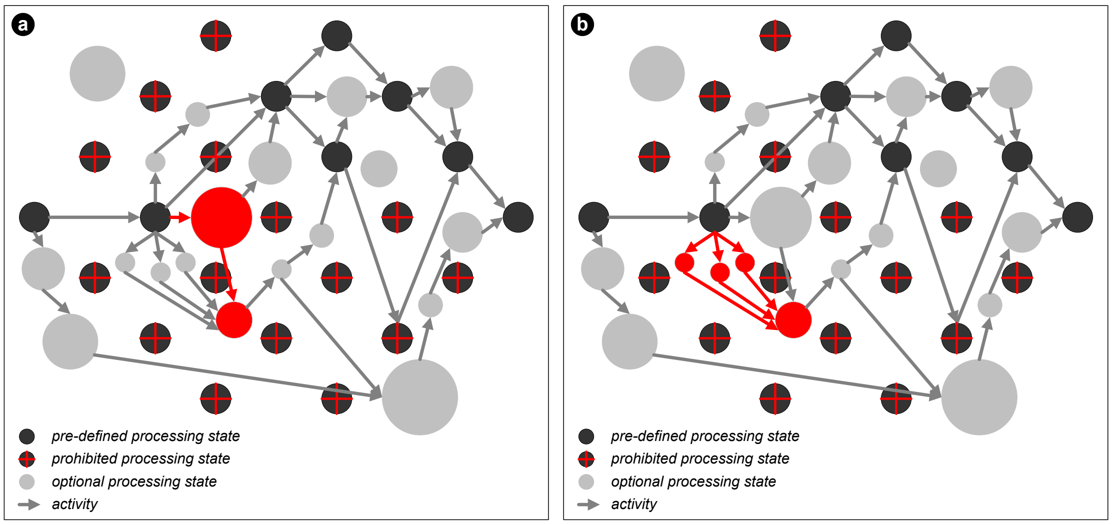

a
pre-defined processing state
prohibited processing state
optional processing state
activity
b
pre-defined processing state
prohibited processing state
optional processing state
activity

Figure 15.2: Choosing preferred work practices

process instances. The latter is achieved through a precise and intuitive methodology, which allows for the harmonized modelling of processes. The tight integration with corresponding business data leads to more comparable process models. Finally, the tight integration of business data and the data-driven execution paradigm allows ensuring compliance between the state of data objects and the progress of the process. Since it can be controlled, which context information is available for a particular user, the "context tunnelling" problem can be avoided.

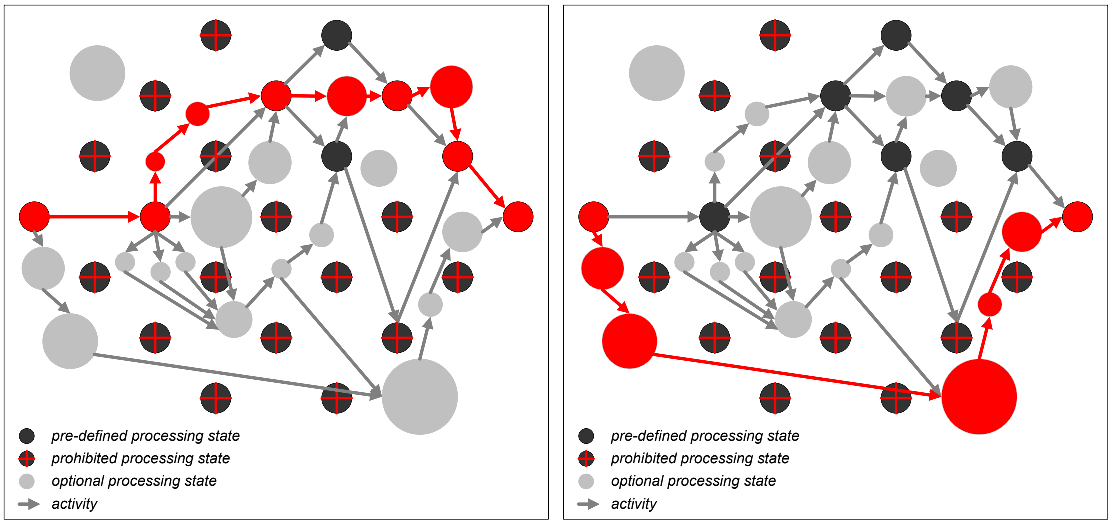

pre-defined processing state
prohibited processing state
optional processing state
activity
pre-defined processing state
prohibited processing state
optional processing state
activity

Figure 15.3: Varying execution of process instances

15.3 Outlook

This thesis has covered the basic relationships between business processes, business functions, and business data. Thereby, it has focused on the main stages of the process lifecycle; i.e., process modeling and execution. In future work, we will extend our framework by addressing other advanced issues related to object-aware process management (cf. Fig. 15.4); e.g., historization, traceability, process variability, and process flexibility (including ad-hoc changes). Another challenging issue is to generate different process perspectives based on one and the same process by taking the current context of the user into account as well. In addition, it should be satisfied that process execution is compliant with business rules as discussed in the context of imperative approaches [LKRM$^+$10, LRMD10]. Finally, we are currently working on the controlled evolution of data and process models [CKR12b, CKR12a].

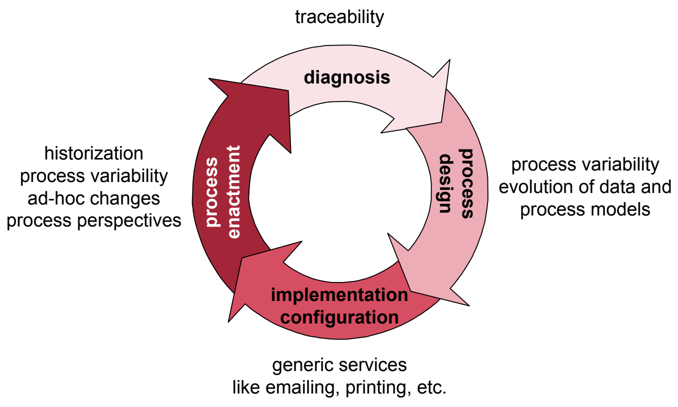


Figure 15.4: Further issues

Overall, we will continue working on the PHILharmonicFlows framework to realize a more flexible process management technology that allows performing daily work in a more natural and intuitive way.

Bibliography

[Ath02] Pallas Athena. *Flower User Manual*. Pallas Athena, 2002.

[BBU99] J. Barkley, K. Beznosov, and J. Uppal. Supporting Relationships in Access Control Using Role Based Access Control. In *Proceedings of the fourth ACM Workshop on Role-based Access Control (RBAC '99)*, pages 55–65. ACM, 1999.

[Bec12] H. Beck. *Implementierung einer Komponente zur Modellierung von Mikro-Prozessen in einem datenorientierten Prozess- Management-System*. Diploma thesis, Ulm University, 2012.

[Ber98] E. Bertino. Data Security. *Data and Knowledge Engineering*, 25(1-2):199–216, 1998.

[BFA99] E. Bertino, E. Ferrari, and V. Atluri. The Specification and Enforcement of Authorization Constraints in Workflow Management Systems. *ACM Transactions on Information and System Security*, 2(1):65–104, 1999.

[BGH$^+$07] K. Bhattacharya, C. Gerede, R. Hull, R. Liu, and J. Su. Towards Formal Analysis of Artifact-centric Business Process Models. In *Proceedings of the 5th international Conference on Business Process Management (BPM'07)*, pages 288–304. Springer-Verlag, 2007.

[BHM01] L. Brehm, A. Heinzl, and M. Markus. Tailoring ERP Systems: A Spectrum of Choices and their Implications. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS '01)*, pages 3–6. IEEE Computer Society, 2001.

[BHS09] K. Bhattacharya, R. Hull, and J. Su. *A Data-Centric Design Methodology for Business Processes*, pages 503–531. IGI Global, 2009.

[Boe86] B. Boehm. A Spiral Model of Software Development and Enhancement. *SIGSOFT Software Engineering Notes*, 11(4):14–24, 1986.

[Bot02] R. Botha. *CoSAWoE - A Model for Context-sensitive Access Control in Workflow Environments*. PhD thesis, Rand Afrikaans University, 2002.

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Longman Publishing Co., Inc., 1998.

[CH09] D. Cohn and R. Hull. Business Artifacts : A Data-centric Approach to Modeling Business Operations and Processes. *IEEE Data Engineering Bulletin*, pages 3–9, 2009.

[CKLY98] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri Nets from Finite Transition Systems. *IEEE Transactions on Computers*, 47(8):859–882, 1998.

[CKR12a] C. M. Chiao, V. Künzle, and M. Reichert. Schema Evolution in Object and Process-Aware Information Systems: Issues and Challenges. In *1st Int. Workshop on Data- and Artifact-centric BPM (DAB'12), BPM'12 Workshops*, volume 132 of *LNBIP*, pages 328–339. Springer, 2012.

[CKR12b] C. M. Chiao, V. Künzle, and M. Reichert. Towards Object-Aware Process Support in Healthcare Information Systems. In *4th International Conference on eHealth, Telemedicine, and Social Medicine (eTELEMED'12)*, pages 227–236. IARIA, 2012.

[Cod90] E. F. Codd. *The Relational Model for Database Management*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[CV11] M. Comuzzi and I. Vanderfeesten. Product-Based Workflow Design for Monitoring of Collaborative Business Processes. In *Proceedings of the 23rd International Conference on Advanced Information Systems Engineering (CAiSE 2011)*, volume 6741 of *LNCS*, pages 154–168. Springer Berlin, 2011.

[DD97] C. J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[DH12] U. Döbrich and R. Heidel. Datengetriebene Programmsysteme. *Informatik Spektrum*, 35(3):190–203, 2012.

[DHV11] E. Damaggio, R. Hull, and R. Vaculín. On the Equivalence of Incremental and Fixpoint Semantics for Business Artifacts with Guard-Stage-Milestone Lifecycles. In *Proceedings of the 9th International Conference on Business Process Management (BPM'11)*, pages 396–412. Springer-Verlag, 2011.

[DHV13] E. Damaggio, R. Hull, and R. Vaculín. On the Equivalence of Incremental and Fixpoint Semantics for Business Artifacts with Guard-Stage-Milestone Lifecycles. *Information Systems*, 38(4):561–584, 2013.

[Dij76] E. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, 1st edition, 1976.

[Dor02] D. Dori. *Object-Process Methodology*. Springer, 2002.

[DR09] P. Dadam and M. Reichert. The ADEPT Project: A Decade of Research and Development for Robust and Flexible Process Support - Challenges and Achievements. *Computer Science - Research and Development*, 23(2):81–97, 2009.

[DRRM11] P. Dadam, M. Reichert, and S. Rinderle-Ma. Prozessmanagementsysteme: Nur ein wenig Flexibilität wird nicht reichen. *Informatik-Spektrum*, 34(4):364–376, 2011.

[Dru67] F. Drucker. *The Effective Executive*. Harper Collins, London, 1967.

[Dvt05] M. Dumas, W. M. P. van der Aalst, and A. H. M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software through Process Technology*. Wiley, 2005.

[ER89] A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures - Part 1 and Part 2. *Acta Informatica*, 27(4):315–368, 1989.

[Esh02] R. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modeling*. PhD thesis, University of Twente, 2002.

[FK92] D. Ferraiolo and R. Kuhn. Role-based Access Control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[FLM$^+$09] D. Fahland, D. Lübke, J. Mendling, H. Reijers, B. Weber, M. Weidlich, and S. Zugal. Declarative versus Imperative Process Modeling Languages: The Issue of Understandability. In *Enterprise, Business-Process and Information Systems Modeling*, volume 29 of *LNBIP*, pages 353–366. Springer Berlin Heidelberg, 2009.

[FMR$^+$10] D. Fahland, J. Mendling, H. Reijers, B. Weber, M. Weidlich, and S. Zugal. Declarative versus Imperative Process Modeling Languages: The Issue of Maintainability. In *Business Process Management Workshops*, volume 43 of *LNBIP*, pages 477–488. Springer Berlin Heidelberg, 2010.

[Fra10] D. S. Frankel. *Model Driven Architecture (OMG): Applying MDA to Enterprise Computing*. Wiley, 2010.

[GBS07] C. E. Gerede, K. Bhattacharya, and J. Su. Static Analysis of Business Artifact-centric Operational Models. In *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA '07)*, pages 133–140. IEEE Computer Society, 2007.

[Ges10] D. Gessenharter. Extending The UML Semantics For A Better Support of Model Driven Software Development. In *Proceedings of the 2010 International Conference on Software Engineering Research & Practice (SERP'10)*, pages 45–51. CSREA Press, 2010.

[GR92] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992.

[GRv08] C. W. Guenther, M. Reichert, and W. M. P. van der Aalst. Supporting Flexible Processes with Adaptive Workflow and Case Handling. In *3rd IEEE Workshop on Agile Cooperative Process-aware Information Systems (ProGility'08)*, pages 229–234. IEEE Computer Society Press, 2008.

[GS07] C. E. Gerede and J. Su. Specification and Verification of Artifact Behaviors in Business Process Models. In *Proceedings of the 5th International Conference on Service-Oriented Computing (ICSOC '07)*, pages 181–192. Springer-Verlag, 2007.

[HBR10a] A. Hallerbach, T. Bauer, and M. Reichert. Capturing Variability in Business Process Models: The Provop Approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(6-7):519–546, 2010.

[HBR10b] Alena Hallerbach, Thomas Bauer, and Manfred Reichert. *Configuration and Management of Process Variants*, pages 237–255. Springer, 2010.

[HDD$^+$11] R. Hull, E. Damaggio, R. De Masellis, F. Fournier, M. Gupta, F. T. Heath, S. Hobson, M. Linehan, S. Maradugu, A. Nigam, P. N. Sukaviriya, and R. Vaculin. Business Artifacts with Guard-Stage-Milestone Lifecycles: Managing Artifact Interactions with Conditions and Events. In *Proceedings of the 5th ACM International Conference on Distributed Event-based System (DEBS '11)*, pages 51–62. ACM, 2011.

[HL99] C. P. Holland and B. Light. A Critical Success Factors Model for ERP Implementation. *IEEE Software*, 16(3):30–36, 1999.

[HMPR04] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105, 2004.

[HTG12] N. Haddar, M. Tmar, and F. Gargouri. Implementation of a Data-driven Workflow Management System. In *IEEE 15th International Conference on Computational Science and Engineering (CSE'12)*, pages 111–118, 2012.

[Hul08] R. Hull. Artifact-Centric Business Process Models: Brief Survey of Research Results and Challenges. In *Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part II on On the Move to Meaningful Internet Systems (OTM '08)*, pages 1152–1163. Springer-Verlag, 2008.

[HW04] J. Hu and A. C. Weaver. A Dynamic, Context-Aware Security Infrastructure for Distributed Healthcare Applications. In *Proc. 1st Workshop on Pervasive Privacy Security, Privacy, and Trust (PSPT'04)*, 2004.

[Kï1] V. Künzle. Towards a Framework for Object-Aware Process Management. In *1st Int'l Symposium on Data-driven Process Discovery and Analysis (SIMPDA'11), PhD Seminar*, 2011.

[KB11] V. Künzle and S. Buchhester. Routenplanung für Unternehmen. *HR Performance*, 5:57–63, 2011.

[KE11] A. Kemper and A. Eickler. *Datenbanksysteme: Eine Einführung*. Oldenbourg Wissenschaftsverlag, 2011.

[Kel02] W. Keller. *Enterprise Application Integration*. dpunkt Verlag, 2002.

[KG07] K. Küster, J. Ryndina and H. Gall. Generation of Business Process Models for Object Life Cycle Compliance. In *Business Process Management (BPM'07)*, volume 4714 of *LNCS*, pages 165–181. Springer Berlin Heidelberg, 2007.

[KKC02] A. Kumar, N. Karnik, and G. Chafle. Context Sensitivity in Role-based Access Control. *SIGOPS Operating Systems Review*, 36(3):53–66, 2002.

[KLW08] S. Kumaran, R. Liu, and F. Y. Wu. On the Duality of Information-Centric and Activity-Centric Models of Business Processes. In *Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE '08)*, pages 32–47. Springer-Verlag, 2008.

[KR05] P. Kleinschmidt and C. Rank. *Relationale Datenbanksysteme*. Springer, 2005.

[KR09a] V. Künzle and M. Reichert. Herausforderungen auf dem Weg zu datenorientierten Prozess-Management-Systemen. *EMISA Forum*, 29(2):9–24, 2009.

[KR09b] V. Künzle and M. Reichert. Integrating Users in Object-Aware Process Management Systems: Issues and Challenges. In *Business Process Management Workshops (Proc. BPD'09)*, volume 43 of *LNBIP*, pages 29–41. Springer Berlin Heidelberg, 2009.

[KR09c] V. Künzle and M. Reichert. Towards Object-aware Process Management Systems: Issues, Challenges, Benefits. In *Enterprise, Business-Process and Information Systems Modeling (BPMDS'09)*, volume 29 of *LNBIP*, pages 197–210. Springer Berlin Heidelberg, 2009.

[KR10] V. Künzle and M. Reichert. Herausforderungen bei der Integration von Benutzern in Datenorientierten Prozess-Management-Systemen. *EMISA Forum*, 30(1):11–28, 2010.

[KR11a] V. Künzle and M. Reichert. A Modeling Paradigm for Integrating Processes and Data at the Micro Level. In *Enterprise, Business-Process and Information Systems Modeling (BPMDS'11)*, volume 81 of *LNBIP*, pages 201–215. Springer, 2011.

[KR11b] V. Künzle and M. Reichert. PHILharmonicFlows : Towards a Framework for Object-Aware Process Management. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(4):205–244, 2011.

[KR11c] V. Künzle and M. Reichert. PHILharmonicFlows: Research an Design Methodology. *Technical Report*, 2011.

[KR11d] V. Künzle and M. Reichert. Striving for Object-Aware Process Support: How Existing Approaches Fit Together. In *1st Int'l Symposium on Data-driven Process Discovery and Analysis (SIMPDA'11)*, 2011.

[KRG00] H. Klaus, M. Rosemann, and G. G. Gable. What is ERP? *Information Systems Frontiers*, 2(2):141–162, 2000.

[KS91] G. Kappel and M. Schrefl. Object / Behavior Diagrams. In *7th International Conference on Data Engineering (ICDE)*, pages 530–539, 1991.

[KWR10a] V. Künzle, B. Weber, and M. Reichert. Object-aware Business Processes: Fundamental Requirements and their Support in Existing Approaches. *International Journal of Information System Modeling and Design (IJISM)*, 2(2):9–46, 2010.

[KWR10b] V. Künzle, B. Weber, and M. Reichert. Object-aware Business Processes: Properties, Requirements, Existing Approaches. *Technical Report*, 2010.

[LBW07] R. Liu, K. Bhattacharya, and F. Y. Wu. Modeling Business Contexture and Behavior Using Business Artifacts. *Advanced Information Systems Engineering*, 4495:324–339, 2007.

[LKRD10] A. Lanz, U. Kreher, M. Reichert, and P. Dadam. Enabling process support for advanced applications with the aristaflow bpm suite. In *Proc. of the Business Process Management 2010 Demonstration Track*, number 615 in CEUR Workshop Proceedings, 2010.

[LKRM$^+$10] L. T. Ly, D. Knuplesch, S. Rinderle-Ma, K. Goeser, H. Pfeifer, M. Reichert, and P. Dadam. SeaFlows Toolset - Compliance Verification Made Easy for Process-aware Information Systems. In *Proc. CAiSE'10 Forum - Information Systems Evolution*, number 72 in LNBIP, pages 76–91. Springer, 2010.

[LR00] F. Leymann and D. Roller. *Production Workflow*. Prentice-Hall, 2000.

[LR12] M. Lohrmann and M. Reichert. Efficacy-aware Business Process Modeling. In *On the Move to Meaningful Internet Systems (OTM 2012), 20th International Conference on Cooperative Information Systems*, volume 7565 of *LNCS*, pages 38–55. Springer Berlin Heidelberg, 2012.

[LRMD10] L. T. Ly, S. Rinderle-Ma, and P. Dadam. Design and Verification of Instantiable Compliance Rule Graphs in Process-Aware Information Systems. In *The 22nd International Conference on Advanced Information Systems Engineering (CAiSE'10)*, number 6051 in LNCS, pages 9–23. Springer, 2010.

[LS97] E Lupu and M. Sloman. A Policy Based Role Object Model. In *Enterprise Distributed Object Computing Workshop (EDOC '97)*, pages 36–47, 1997.

[Mö9] D. Müller. *Management datengetriebener Prozessstrukturen*. PhD thesis, Ulm University, 2009.

[MHHR06] D. Müller, J. Herbst, M. Hammori, and M. Reichert. IT Support for Release Management Processes in the Automotive Industry. In *4th Int'l Conf. on Business Process Management (BPM'06)*, number 4102 in LNCS, pages 368–377. Springer, 2006.

[MKR12] N. Mundbrod, J. Kolb, and M. Reichert. Towards a System Support of Collaborative Knowledge Work. In *1st Int'l Workshop on Adaptive Case Management (ACM'12), BPM'12 Workshops*, LNBIP. Springer, 2012.

[MRB08] B. Mutschler, M. Reichert, and J. Bumiller. Unleashing the Effectiveness of Process-Oriented Information Systems: Problem Analysis, Critical Success Factors, and Implications. *IEEE Transactions on Systems, Man, and Cybernetics*, 38(3):280–291, 2008.

[MRH07] D. Müller, M. Reichert, and J. Herbst. Data-Driven Modeling and Coordination of Large Process Structures. In *Proceedings of the 2007 OTM Confederated International Conference on On the move to Meaningful Internet Systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I*, number 4803 in LNCS, pages 131–149. Springer, 2007.

[MRH08a] D. Müller, M. Reichert, and J. Herbst. A New Paradigm for the Enactment and Dynamic Adaptation of Data-Driven Process Structures. In *Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE'08)*, number 5074 in LNCS, pages 48–63. Springer, 2008.

[MRH$^+$08b] D. Müller, M. Reichert, J. Herbst, D. Köntges, and A. Neubert. COREPRO-Sim: A Tool for Modeling, Simulating and Adapting Data-driven Process Structures. In *6th International Conference on Business Process Management (BPM'08 Demonstrations)*, volume 5240 of *LNCS*, pages 394–397. Springer, 2008.

[MRv+10] R. S. Mans, N. C. Russell, W. M. P. van der Aalst, A. J. Moleman, P. J. M. Bakker, and M. Jaspers. Proclets in Healthcare. *Journal of Biomedical Informatics*, 43(4):632–649, 2010.

[MS95] S. T. March and G. F. Smith. Design and Natural Science Research on Information Technology. *Decision Support Systems*, 15(4):251–266, 1995.

[MWR08] B. Mutschler, B. Weber, and M. Reichert. Workflow Management versus Case Handling: Results from a Controlled Software Experiment. In *Proceedings of the 2008 ACM symposium on Applied computing (SAC'08)*, pages 82–89. ACM, 2008.

[NC03] A. Nigam and N. S. Caswell. Business Artifacts: An approach to Operational Specification. *IBM Systems Journal*, 42(3):428–445, 2003.

[PDG+11] H. Partsch, M. Dausend, D. Gessenharter, J. Kohlmeyer, and A. Raschke. From Formal Semantics to Executable Models: A Pragmatic Approach to Model-Driven Development. *International Journal of Software and Informatics*, 5(1-2):291–312, 2011.

[Pes08] M. Pesic. *Constraint-Based Workflow Management Systems: Shifting Control to Users*. PhD thesis, Eindhoven, University of Technology, 2008.

[Pfe05] V. Pfeiffer. *A Framework for Evaluating Access Control Concepts in Workflow Management Systems*. Diploma thesis, Ulm University, 2005.

[Prö11] A. Pröbstle. *Technische Konzeption und Realisierung der Modellierungskomponente für ein daten-orientiertes Prozess-Management-System*. Diploma thesis, Ulm University, 2011.

[PS98] G. Preuner and M. Schrefl. Observation Consistent Integration of Views of Object Life-Cycles. In *Proceedings of the 16th British National Conference on Databases: Advances in Databases (BNCOD 16)*, pages 32–48. Springer-Verlag, 1998.

[PWZ+12] P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, and H. Reijers. Imperative versus Declarative Process Modeling Languages: An Empirical Investigation. In *Proc. Business Process Management Workshops ER-BPM '11*, volume 99 of *LNBIP*, pages 383–394. Springer Berlin Heidelberg, 2012.

[RD98] M. Reichert and P. Dadam. Supporting Dynamic Changes of Workflows without Losing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.

[RD09] M. Reichert and P. Dadam. Enabling Adaptive Process-aware Information Systems with ADEPT2. In *Handbook of Research on Business Process Modeling*, pages 173–203. Information Science Reference, 2009.

[RDtI07] G. Redding, M. Dumas, A. H. M. ter Hofstede, and A. Iordachescu. Transforming Object-oriented Models to Process-oriented Models. In *Proceedings of the 2007 International Conference on Business Process Management (BPM'07)*, number 4928 in LNCS, pages 132–143. Springer, 2007.

[RDtI09a] G. M. Redding, M. Dumas, A. H. M. ter Hofstede, and A. Iordachescu. A Flexible, Object-centric Approach for Business Process Modelling. *Service Oriented Computing and Applications*, 4(3):191–201, 2009.

[RDtI09b] G. M. Redding, M. Dumas, A. H. M. ter Hofstede, and A. Iordachescu. Modelling Flexible Processes with Business Objects. *Seventh IEEE International Conference on E-Commerce Technology (CEC'05)*, 0:41–48, 2009.

[Red09] G. M. Redding. *Object-centric Process Models and the Design of Flexible Processes*. PhD thesis, Queensland University of Technology, 2009.

[Rei00] M. Reichert. *Dynamische Ablaufänderungen in Workflow-Management-Systemen*. PhD thesis, Ulm University, Germany, 2000.

[Rei02] H. A. Reijers. Product-Based Design of Business Processes Applied within the Financial Services. *Journal of Research and Practice in Information Technology*, 34(2):34–46, 2002.

[RHD98] M. Reichert, C. Hensinger, and P. Dadam. Supporting Adaptive Workflows in Advanced Application Environments. In *EDBT Workshop on Workflow Management Systems*, pages 100–109, 1998.

[RKG06] K. Ryndina, J. Küster, and H. Gall. Consistency of Business Process Models and Object Life Cycles. In *Proceedings of the 2006 International Conference on Models in Software Engineering (MoDELS'06)*, pages 80–90. Springer, 2006.

[RL03] H. A. Reijers and W. M. P. Liman, S. van der Aalst. Product-Based Workflow Design. *Management Information Systems*, 20(1):229–262, 2003.

[RM09] S. Rinderle-Ma. Data Flow Correctness in Adaptive Workflow Systems. *EMISA Forum*, 29(2):25–35, 2009.

[RMR07] S. Rinderle-Ma and M. Reichert. A Formal Framework for Adaptive Access Control Models. *Journal on Data Semantics IX*, pages 82–112, 2007.

[RMR08] S. Rinderle-Ma and M. Reichert. Managing the Life Cycle of Access Rules in CEOSIS. In *Proceedings of the 12th IEEE International Enterprise Computing Conference (EDOC'08)*, pages 257–266. IEEE Computer Society Press, 2008.

[RMR09] S. Rinderle-Ma and M. Reichert. Comprehensive Life Cycle Support for Access Rules in Information Systems: The CEOSIS Project. *Enterprise Information Systems*, 3(3):219–251, 2009.

[RMRD04] S. Rinderle-Ma, M. Reichert, and P. Dadam. Correctness Criteria for Dynamic Changes in Workflow Systems: A Survey. *Data & Knowledge Engineering*, 50(1):9–34, 2004.

[RR06] S. Rinderle and M. Reichert. Data-driven Process Control and Exception Handling in Process Management Systems. In *Proceedings of the 18th International Conference on Advanced Information Systems Engineering (CAiSE'06)*, pages 273–287. Springer, 2006.

[RRD09] M. Reichert, S. Rinderle, and P. Dadam. Flexibility in Process-aware Information Systems. *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, 2(5460):115–135, 2009.

[RRKD05] M. Reichert, S. Rinderle, U. Kreher, and P. Dadam. Adaptive Process Management with ADEPT2. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 1113–1114. IEEE Computer Society, 2005.

[RRvdA03] H. A. Reijers, J. Rigter, and W. M. P. van der Aalst. The Case Handling Case. *International Journal of Cooperative Information Systems*, 12(3):365–391, 2003.

[RtE05] N. C. Russell, A. H. M. ter Hofstede, and D. Edmond. Workflow Resource Patterns. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE'05)*, pages 216–232. Springer, 2005.

[RVV10] H. A. Reijers, J. Vogelaar, and I. Vanderfeesten. Changing Products, Changing Processes: Dealing with Small Updates in Product-Based Design. In *Proceedings of the 2nd International Conference on Information, Process, and Knowledge Management (eKNOW 2010)*, pages 56–61, 2010.

[RW12] M. Reichert and B. Weber. *Enabling Flexibility in Process-Aware Information Systems*. Springer, 2012.

[RzM98] M. Rosemann and M. zur Mühlen. Modellierung der Aufbauorganisation in Workflow-Management-Systemen: Kritische Bestandsaufnahme und Gestaltungsvorschläge. *EMISA-Forum*, 3(1):78–86, 1998.

[RzM04] M. Rosemann and M. zur Mühlen. Organizational Management in Workflow Applications: Issues and Perspectives. *Information Technology and Management*, 5(3-4):271–291, 2004.

[Sch10] C. Scheb. *Entwicklung eines Usability-Konzepts für die Laufzeitumgebung eines datenorientierten Prozess-Management-Systems*. Diploma thesis, Ulm University, 2010.

[Sch12] S. Schultz. *Implementierung einer Komponente zur Ausführung von Mikro-Prozessen in einem datenorientierten Prozess-Management System*. Diploma thesis, Ulm University, 2012.

[Sil09] B. Silver. Case Management: Addressing unique BPM Requirements. *BPMS Watch*, pages 1–12, 2009.

[Sim96] H. A. Simon. *The Sciences of the Artificial (3rd ed.)*. MIT Press Cambridge USA, 1996.

[SOSS05] S. W. Sadiq, M. E. Orlowska, W. Sadiq, and K. Schulz. When Workflows will not deliver: The Case of contradicting Work Practice. In *Proceedings of the 8th International Conference on Business Information Systems (BIS'05)*, volume 1, pages 69–84. Wydawnictwo Akademii Ekonomicznej w Poznaniu, 2005.

[Spi13] T. Spindler. *Int. der Modellierungs- und Laufzeitumgebung eines datenorientierten Prozess-Management-Systems*. Master thesis, Ulm University, 2013.

[SSB09] N. Schroeder, U. Spinola, and J. Becker. SAP Records Management. *SAP PRESS*, 2009.

[SSO05] S. Sadiq, W. Sadiq, and M. E. Orlowska. Specification and Validation of Process Constraints for Flexible Workflows. *Information Systems*, 30(5):349–378, 2005.

[ST97] R. S. Sandhu and R. K. Thomas. Task-based Authorization Controls (TBAC): A Family of Models for Active and Enterprise-oriented Authorization Management. In *Proceedings of the IFIP TC11 WG11.3 Eleventh International Conference on Database Security XI: Status and Prospects (IFIP'97)*, pages 166–181. Chapman & Hall, Ltd., 1997.

[SV01] P. Samarati and S. Vimercati. Access Control: Policies, Models and Mechanisms. In *Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design: Tutorial Lectures (FOSAD)*, pages 137–196. Springer, 2001.

[TDGS07] I. J. Tylor, E. Deelman, D. B. Gannon, and M. Shields. *Workflows for E-Science*. Springer, 2007.

[Tho97] R. K. Thomas. Team-based Access Control (TMAC): A Primitive for Applying Role-based Access Controls in Collaborative Environments. In *Proceedings of the Second ACM Workshop on Role-based Access Control (RBAC '97)*, pages 13–19. ACM, 1997.

[TRI09] L. Thom, M. Reichert, and C. Iochpe. Activity Patterns in Process-aware Information Systems: Basic Concepts and Empirical Evidence. *International Journal of Business Process Integration and Management (IJBPIM)*, 4(2):93–110, 2009.

[van97] P. J. van Strien. Towards a Methodology of Psychological Practice: The Regulative Cycle. *Theory & Psychology*, 7(5):683–700, 1997.

[Van09] I. Vanderfeesten. *Product-Based Design and Support of Workflow Processes*. Phd thesis, Eindhoven University of Technology, 2009.

[vB01] W. M. P. van Der Aalst and P.J.S. Berens. Beyond Workflow Management : Product-Driven Case Handling. In S. Ellis, T. Rodden, and I. Zigurs, editors, *International ACM SIGGROUP Conference on Supporting Group Work (GROUP 2001)*, pages 42–51. ACM Press, 2001.

[vBEW00] W. M. P. van der Aalst, P. Barthelmess, C. A. Ellis, and J. Wainer. Workflow Modeling using Proclets. In *Proceedings of the 7th International Conference on Cooperative Information Systems (CoopIS '02)*, pages 198–209. Springer, 2000.

[vBEW01] W. M. P. van der Aalst, P. Barthelmess, C. A. Ellis, and J. Wainer. Proclets: A Framework for Lightweight Interacting Workflow Processes. *International Journal of Cooperative Information Systems*, 10(4):443–482, 2001.

[vdA97] W. M. P. van der Aalst. Designing Workflows Based on Product Structures. In *Proceedings of the ninth IASTED International Conference on Parallel and Distributed Computing Systems*, pages 337–342. IASTED/Acta Press, 1997.

[vH04] W. M. P. van Der Aalst and K. Hees. *Workflow-Management: Models, Methods and Systems*. MIT Press, Cambridge, 2004.

[VHH$^+$11] R. Vaculin, R. Hull, T. Heath, C. Cochran, A. Nigam, and P. Sukaviriya. Declarative Business Artifact Centric Modeling of Decision and Knowledge Intensive

Business Processes. In *Proceedings of the 2011 IEEE 15th International Enterprise Distributed Object Computing Conference (EDOC '11)*, pages 151–160. IEEE Computer Society, 2011.

[vMR09] W. M. P. van der Aalst, R. S. Mans, and N. C. Russell. Workflow Support Using Proclets: Divide, Interact, and Conquer. *IEEE Bulletin of the Technical Committee on Data Engineering*, 32(3):16–22, 2009.

[vP06] W. M. P. van der Aalst and M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In *Web Services and Formal Methods*, volume 4184 of *LNCS*, pages 1–23. Springer, 2006.

[vPS09] W. M. P. van der Aalst, M. Pesic, and H. Schonenberg. Declarative Workflows: Balancing between Flexibility and Support. *Computer Science - Research and Development*, 23:99–113, 2009.

[VRv08] I. Vanderfeesten, H. A. Reijers, and W. M. P. van der Aalst. Product-Based Workflow Support: Dynamic Workflow Execution. In *Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE'08)*, number 5074 in LNCS, pages 571–574. Springer, 2008.

[VRv11] I. Vanderfeesten, H. A. Reijers, and W. M. P. van der Aalst. Product-based Workflow Support. *Information Systems*, 36(2):517–535, 2011.

[VRvdA08] I. Vanderfeesten, H. A. Reijers, and W. M. P. van der Aalst. Case Handling Systems as Product Based Workflow Design Support. In *Enterprise Information Systems*, pages 187–198. Springer Berlin, 2008.

[vtKB03] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[vtW03] W. M. P. van der Aalst, A. H. M. ter Hofstede, and M. Weske. Business Process Management: A Survey. In *Proceedings of the 2003 International Conference on Business process management (BPM'03)*, number 2678 in LNCS, pages 1–12. Springer, 2003.

[vWG05] W. M. P. van der Aalst, M. Weske, and D. Grünbauer. Case Handling: A new Paradigm for Business Process Support. *Data Knowledge Engineering*, 53(2):129–162, 2005.

[Wag10] N. Wagner. *Entwicklung eines Usability-Konzepts für die Modellierungsumgebung eines datenorientierten Prozess-Management-Systems.* Diploma thesis, Ulm University, 2010.

[WBK03] J. Wainer, P. Barthelmess, and A. Kumar. W-RBAC - A Workflow Security Model Incorporating Controlled Overriding of Constraints. *International Journal of Cooperative Information Systems (IJCIS)*, 12, 2003.

[Wes07] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007.

[Wie09] R. J. Wieringa. Design Science as Nested Problem Solving. In *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology*, DESRIST '09, pages 1–12. ACM, 2009.

[WMR10] B. Weber, B. Mutschler, and M. Reichert. Investigating the Effort of Using Business Process Management Technology: Results from a Controlled Experiment. *Science of Computer Programming*, 75(5):292–310, 2010.

[WRRM08] B. Weber, M. Reichert, and S. Rinderle-Ma. Change Patterns and Change Support Features - Enhancing Flexibility in Process-Aware Information Systems. *Data and Knowledge Engineering*, 66(3):438–466, 2008.

[WRWR05] B. Weber, M. Reichert, W. Wild, and S. Rinderle. Balancing Flexibility and Security in Adaptive Process Management Systems. In *Proceedings of the 2005 Confederated International Conference on On the Move to Meaningful Internet Systems (OTM'05)*. Springer, 2005.

[WRWRM09] B. Weber, M. Reichert, W. Wild, and S. Rinderle-Ma. Providing Integrated Life Cycle Support in Process-Aware Information Systems. *International Journal of Cooperative Information Systems (IJCIS)*, 18(1):115–165, 2009.

[WRZW09] B. Weber, H. A. Reijers, S. Zugal, and W. Wild. The Declarative Approach to Business Process Execution: An Empirical Test. In *Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE '09)*, pages 470–485. Springer-Verlag, 2009.

[WSML02] S. Wu, A. Sheth, J. Miller, and Z. Luo. Authorization and Access Control of Application Data in Workflow Systems. *Journal of Intelligent Information Systems*, 18(1):71–94, 2002.

[WSR09] B. Weber, S. Sadiq, and M. Reichert. Beyond Rigidity - Dynamic Process Lifecycle Support: A Survey on Dynamic Changes in Process-aware Information Systems. *Computer Science - Research and Development*, 23(2):47–65, 2009.

[ZSPW12] S. Zugal, P. Soffer, J. Pinggera, and B. Weber. Expressiveness and Understandability Considerations of Hierarchy in Declarative Business Process Models. In *Enterprise, Business-Process and Information Systems Modeling*, volume 113 of *LNBIP*, pages 167–181. Springer Berlin Heidelberg, 2012.

List of Figures

List of Tables