



ulm university universität
uulm

Ulm University | 89069 Ulm | Germany

**Faculty of Engineering
and Computer Science**
Institute of Databases and
Information Systems

Design and Implementation of a Runtime Environment of an Object-Aware Process Management System

Master's Thesis at Ulm University

Submitted by:

Sebastian Steinau

sebastian.steinau@uni-ulm.de

Reviewer:

Prof. Dr. Manfred Reichert

Dr. Vera Künzle

Supervisor:

Dr. Vera Künzle

2015

Version February 27, 2015

© 2015 Sebastian Steinau

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/de/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Satz: PDF-L^AT_EX 2_ε

Abstract

Contemporary business process management systems manage their processes based on the *activity-centric paradigm*. However, many business processes are *un- or semi-structured* and cannot be represented adequately by the activity-centric paradigm. The fundamental reason for this insufficiency is the lack of cohesion between processes on one hand and data on the other.

Object-aware process management provides a solution for this issue by tightly integrating processes with data. Data is represented as structured object types with complex data dependencies. This is mirrored in the concepts of *object behavior* and *object interactions*. Object behavior is captured in *micro processes*, whereas object interactions are represented by *macro processes*. Both micro and macro processes are integral parts of the runtime of an object-aware process management system. Furthermore, the execution of micro and macro processes is governed by *process rules*. The process rules advance the process state depending on the available data, react to user interactions or perform error handling.

PHILharmonicFlows is a framework that aims at the proper support of object-aware processes. A prototype implementing the framework is currently under development. This thesis contributes to the prototype by describing concepts concerning process rules and by providing implementations as well.

In detail, the contributions are as follows.

The ongoing development of the prototype requires process rules to be easily created and altered, i.e. to enable a high maintainability. Therefore, a high level of abstraction for the process rule definitions is necessary. The *Process Rule Framework* enables the creation and alteration of process rules and satisfies the high maintainability requirement by leveraging functional programming devices. Using the Process Rule Framework, a high productivity can be achieved when handling process rules. Moreover, the Process Rule Framework has the possibility to be extended to a general-purpose rule engine in which rules may be created, compiled and executed during runtime.

At runtime, process rules provide one of the cornerstones to object-aware process execution. This requires complex interactions between process rules, as they can trigger each other and therefore chain together. The challenge is to enable these interactions while keeping the individual process rules independent from each other and only loosely coupled. The *Process Rule Manager* enables the complex interactions between process rules by providing the means to coordinate and control process rule interactions. Additionally, the Process Rule Manager abstracts from the inner workings of process execution and provides a well-defined interface to interact with a micro process.

Contents

1. Introduction	1
1.1. Motivation	1
1.2. Contribution	2
1.3. Outline	3
2. The PHILharmonicFlows Framework	5
2.1. Object-Aware Processes	5
2.1.1. Definitions	5
2.1.2. Requirement 1: Object Behavior	6
2.1.3. Requirement 2: Object Interactions	6
2.1.4. Requirement 3: Data-Driven Execution	7
2.1.5. Requirement 4: Variable Activity Granularity	7
2.1.6. Requirement 5: Integrated Access to Business Processes and Objects	7
2.2. Micro Processes	8
2.2.1. Forms	9
2.2.2. Markings	11
2.2.3. Process Rules	11
2.3. Micro Process Execution	12
2.4. Prototype Architecture	18
2.5. Summary	20
3. Programming Concepts	21
3.1. C# in general	21
3.2. Windows Presentation Foundation and Windows Communication Foun- dation	22
3.3. Namespaces	22
3.4. Partial Classes	22
3.5. Interfaces	23
3.6. Properties	23
3.7. Collections	24
3.8. LINQ and Fluent Interfaces	25
3.9. Lambdas	26
3.10. Delegates and Events	26
3.11. Extension Methods	28
3.12. Expressions and Expression Trees	28
3.13. Summary	29
4. The Process Rule Framework	31
4.1. Runtime Model	31

Contents

4.2. Implementation	34
4.2.1. Process Rule Analysis	34
4.2.2. Goals	36
4.2.3. Example Rule	37
4.2.4. Preconditions	38
4.2.5. Preconditions for Collections	41
4.2.6. Effects	42
4.2.7. Effects on Collections	45
4.2.8. Effects for Execution Rules	47
4.2.9. Restrictions on Effects	47
4.2.10. Building and Using a Process Rule	47
4.2.11. Applications of the Process Rule Framework	47
4.3. Summary	48
5. The Process Rule Manager	49
5.1. Event-driven Rule Application	49
5.2. Process Rule Manager Context and Example	50
5.3. Process Rule Manager Implementation	52
5.3.1. Reaction Rule Mapper	53
5.3.2. Process Rule Repository	53
5.3.3. Handling a Process Rule Event	54
5.3.4. Type Switch	55
5.4. Initialization	57
5.5. Summary	57
6. Related Work	59
6.1. QuestionSys	59
6.2. Rete Algorithm	59
6.3. Microsoft BizTalk Server Business Rule Engine	60
7. Summary and Outlook	61
7.1. Summary	61
7.2. Outlook	61
A. Source	69

1

Introduction

1.1. Motivation

Every business on the market has at least one process to manufacture a product or deliver a service. Additionally, many more processes exist to deal with other vital requirements of a business, for example acquiring new personnel, ordering raw materials or shipping the goods to the customers. In order to stay competitive, a company has to find ways to improve itself. Processes are an obvious choice to look for improvements, leading to an edge over the competitors [13].

Since its beginning, information technology has been able to improve and streamline processes. However, the approach was mostly to digitize and improve a single aspect or a small part of a process with specialized software. To improve and support processes as a whole, computer science has dedicated a branch to researching, describing and optimizing business processes. This branch is called *Business Process Management*. So far, the research resulted in various paradigms for describing, modeling, executing, monitoring and optimizing processes. This led to software called *process management systems* to manage a company's processes according to those paradigms [39].

The most common way of modeling processes is the *activity-centric paradigm* [39]. Examples include the AristaFlow BPM Suite [15, 24], YAWL [50, 48] and the Activiti BPM Platform [38]. According to this paradigm, a process can be split into *activities*. An example for an activity would be writing a letter or interviewing an applicant, which also have a certain order, modeled by so called sequence flow. Domain languages like the *business process model and notation (BPMN)* [36] have been developed to describe activity-centric processes adequately. The most successful commercial process management systems are also based on an activity-centric process model. However, using these process management systems in practice has revealed serious drawbacks, like the rigidity of activity-centric processes [52]. Deviating from the predefined execution path is at least difficult and often impossible. To provide the necessary flexibility for real-world processes, concepts like ADEPT [40] have been built on top of the activity-centric paradigm. Others have developed new paradigms like case handling [43, 51], declarative activity-centric processes [37, 53, 49] and artifact-centric processes [9, 2] to increase flexibility.

1. Introduction

Another major drawback of current process management systems is the insufficient integration of data into processes [21, 19]. On the one hand, this comprises the lack of structured data types and the lack of an explicit view on the data involved in process execution. For example *person* involves a name, his or her age, an address and many data types, depending on the context the person data type is used in. On the other, relations of objects can at most be expressed inadequately, if at all. For example, a company can have multiple applications for a job position. In order for an applicant to get the job, his or her application must receive at least three positive reviews and at least one interview with him or her must be conducted [18]. For most process management systems, the relation “at least three reviews” between applicant and reviews is impossible to express. Many of the new approaches like case handling or the artifact-centric paradigm in addition to increasing process flexibility also enable a tighter integration of data. However, these concepts are still unsatisfactory for some knowledge-intensive applications.

PHILharmonicFlows [3, 6, 18, 16, 17, 42] addresses and solves issues like modeling object relations by providing a well defined framework for integrating processes and data. It defines the characteristics of an *object-aware process* [19] and creates tools and specifications to model, execute and monitor such object-aware processes. Since object-aware processes are fundamentally different from activity-centric processes, a new way of of business process modeling is introduced [4]. PHILharmonicFlows also provides a runtime component for executing object-aware processes [17].

This master’s thesis describes the part of the prototypical runtime of PHILharmonicFlows which executes object-aware processes, in particular micro processes [20]. Most of the execution logic of micro processes relies on process rules. A framework for creating process rules is presented, as well as a concept to establish the complex interactions between individual process rules. For all concepts, a corresponding implementation is provided as well.

1.2. Contribution

The concepts presented in this thesis form a significant contribution to the runtime of PHILharmonicFlows. In detail, the contributions are as follows

- Process rules can be created and altered with a concise and high-level syntax. With the design of the Process Rule Framework, special focus was placed on the maintainability of the rules.
- A precise coordination of process rules is provided by the Process Rule Manager. Supported by the Process Rule Framework, clear separation and independence of individual process rules is achieved.
- A testing suite consisting of exemplary micro process execution templates, individual process rule tests and process model verifications to verify the functionality

of process rules, the proper coordination of the process rules and the structural soundness of new process models. The testing suite is not described in this thesis.

All concepts have been prototypically implemented and tested. Also, all process rules concerning micro processes have been implemented. As of Christmas 2014, a micro process with backwards transitions was for the first time successfully executed, satisfying all requirements detailed in Chapter 8 of [17]. As an additional feature, the Process Rule Framework laid the basis for a business rule engine which is able to define, compile and execute business rules entirely at runtime.

1.3. Outline

The remainder of this thesis is structured as follows: Chapter 2 gives an overview of the PHILharmonicFlows framework, defines object-aware processes and introduces the necessary concepts for understanding the thesis. Of particular interest are *micro processes* and *process rules*. These concepts form the base for the implementation described in Chapters 4 and 5. Chapter 3 explains the programming fundamentals used in the implementation of the PHILharmonicFlows runtime environment. These fundamentals are necessary for understanding the *Process Rule Framework* and the *Process Rule Manager*. Chapter 4 presents the requirements for a Process Rule Framework, followed by a discussion on how these requirements are achieved. The Process Rule Manager enables micro process execution by applying process rules in an event-driven manner. The details are explained in Chapter 5. Chapter 6 describes other approaches to implement rules. Chapter 7 concludes the thesis with a summary and an outlook.

2

The PHILharmonicFlows Framework

*PHILharmonicFlows*¹ is a framework for modeling, executing and monitoring object-aware business processes [17, 19, 16]. Among the features of PHILharmonicFlows are a modeling tool for graphically modeling processes, automatic work list generation and generic form generation based on the underlying process data. It also supports an authorization policy that may be adjusted at a fine level of granularity.

2.1. Object-Aware Processes

In order for a process or framework to be classified as object-aware, a set of requirements must be met. These requirements are described in [17, 22, 21, 41]. First, definitions of terms used in the description of PHILharmonicFlows and object-aware processes are given. Afterwards, the five requirements are presented for classifying an object-aware process.

2.1.1. Definitions

In an object-aware process, data is represented as *object types*. An object type consists of a set of *object attributes* and a set of *object relations* to other object types [22, 16]. Object attributes are atomic data, e.g. a name or an account balance. Object relations define the relationship of an object type to other object types. For example, a company worker must plan and coordinate tasks for the company. For this purpose, he or she has a task planning sheet that comprises task name, duration and assigned workers. “Task name” and “duration” are both object attributes. “Assigned workers” is an object relation, as “worker” is another object type. The empty task planning sheet is the object type. For each task, a new task planning sheet needs to be filled out, which are instances of the object type “task planning sheet”. For each task, a minimum of three workers and a maximum of seven need to be assigned, which is a cardinality constraint on the object relation. Otherwise, a task can not be worked efficiently. For example, task “A”, with a duration of five hours, is assigned four people. For another task “B”, which lasts a whole week, there might be five people assigned. The actual number of assigned workers can

¹ “Process, Humans and Information Linkage for harmonic Business Flows”

2. The PHILharmonicFlows Framework

differ from task to task, respectively the planning sheet for the task. But for each task, the number of workers must be within the cardinality constraints. Object attributes and object relations directly relate to different levels of process granularity: *object behavior* and *object interactions*.

An *activity* in PHILharmonicFlows is the provision of values to object attributes or the creation of object instances [4]. This is done with a *form* that comprises *input fields* for writing object attributes or *data fields* for displaying object attribute values.

2.1.2. Requirement 1: Object Behavior

Object behavior is the first level of granularity and represents processing a single object instance [20]. Basically, it is about providing values to the object attributes and subsequently reading those values. However, a certain processing order of the object attributes may be required. In addition, not every worker may have reading or writing permissions for an object attribute, so authorizations must be taken into account. For this purpose, for every object instance that is created, a corresponding *micro process* [20] is instantiated as well that ensures a correct processing order of attribute and the validity of reading and writing permissions.

2.1.3. Requirement 2: Object Interactions

It must be possible to create and delete object instances at any point in time during process execution. Each of these object instances may have relations to other object instances, with different cardinality constraints. To complicate matters, object instances may be in different processing states, which can affect other instances, e.g. if certain data is not (yet) present.

This leads to the emergence of a *complex process structure*² at runtime [16, 17]. By principle, an object instance is independent from another instance and can be processed concurrently. However, at certain points they may interact with other instances and thus need to be synchronized. These *object interactions* form the second level of granularity. For example, a company can only hire an applicant if he or she has been recommended by at least two people who have reviewed the application. Unless at least two reviews are present, the application process cannot continue. To manage all interactions, a *macro process* is instantiated together with the object type.

Object interactions and macro processes are not the focus of this thesis. Thereby, an in-depth explanation of the challenges and the solution concepts involved with macro processes can be found in [17].

²Müller et al. introduced this notion in the COREPRO framework [34, 35], which allows for the data-driven creation, enactment and monitoring of large process structures that may consist of hundreds of object life cycles at runtime. As opposed to PHILharmonicFlows, COREPRO does not consider the cardinality constraints of object relations.

2.1.4. Requirement 3: Data-Driven Execution

A *data-driven process* is a type of process where the state of a process execution relies solely on the available data. A process execution progresses further when particular data becomes available. To ensure that certain required object attribute values are set, they can be flagged as mandatorily required. If an attribute is mandatorily required, process execution halts until a proper value is available. Flexibility is possible by providing values up front or changing values after process execution. Changing values requires a (partial) re-execution of the process and explicit user commitments for approving object attribute values, e.g. when validity is in question due to a changed context. In context of object interactions, “coordination of multiple micro process instances should be supported in a data-driven way” [41, 22], as progression may depend on data values from other objects instances.

2.1.5. Requirement 4: Variable Activity Granularity

Activities should accommodate users with their preference in work practice [22]. In *Instance-specific activities*, all input and data fields refer to the same object instance. *Context-sensitive activities* have fields that refer to semantically related object instances. For example, the task planning sheet is related to the task object instance itself. When editing object attributes of the task planning sheet, it is possible to alter attribute values of the task instance as well. *Batch activities* allow writing one object attribute value to the attributes of multiple object instances of the same type. For example, several tasks last two hours, so a batch activity can assign each task instance the value in one go, so the task instances do not need to be changed individually. Activity granularity is not to be confused with process granularity, i.e., object behavior and object interactions.

2.1.6. Requirement 5: Integrated Access to Business Processes and Objects

A process-oriented view allows providing object attribute values in a structured, pre-defined way. However, this view lacks flexibility. For users to be able to add object instances, delete object instances or manipulate object attributes at any point in time, a *data-oriented view* is required. This view should also allow access to selected activities for usually uninvolved users. For this purpose, a permission system is required to grant respective authorizations for creating, manipulating and deleting object instances and to ensure user without these permissions are denied access [18]. Permissions are not limited to an object type, but can be granted in respect to a specific instance of an object type. For example, a user might only execute process instances which he or she created. When providing data with the data-oriented view, it must be ensured that the specified object behavior is not violated.

2.2. Micro Processes

The topic of this thesis is centered on object behavior, in particular the execution of the *micro processes* specifying object behavior [20, 17]. The following gives an explanation of micro processes and how such a process is executed. A graphical representation of a micro process is given in Figure 2.1.

Micro processes determine the order in which object attributes need to be provided and check if any of the attributes are mandatory. It is also involved in specifying which user can access a specific attribute at which point in time, however this has been omitted for simplicity reasons. Micro processes are represented as a graph. Object attributes are represented as *micro steps* and are connected by *micro transitions* which determine the order of the micro steps. Micro steps are logically grouped into *states*. A state shows the progress of a micro process execution. A micro process instance has a *start state* that determines where process execution begins. Several *end states* terminate micro process execution when reached. A *backward transition* enables a user to redo previous work, e.g. for correcting errors. A backward transition originates from a state and targets a state, which means a redo operation involves a state as a whole, even if only one object attribute's value needs to be changed. Unchanged values must have an approving commitment by the user to ensure they are still correct.

There are different variants of micro steps, each for a different purpose. *Empty micro steps* do not correspond to an object attribute. Empty micro steps are used among other things to denote the start step and the end steps of a process. Normal micro steps are used to write an object attribute with an unlimited range of values (only restricted by the data type). To impose restrictions on values for object attributes a user can provide, *value-specific micro steps* have been introduced. Its predefined, finite set of attribute values is mapped to a set of *value steps*. A decision, for example a Yes/No-Question, is modeled a value-specific micro step with one value step corresponding to 'Yes', another value step to 'No'. Another example can be found in Figure 2.1, where a value-specific micro step represents a start date. It has the value steps "later" and "now". Mapping predefined values is one application of a value step's *predicate*. A predicate is an expression that has a Boolean value as a result. It can use comparative operators (e.g. $>$, $=$, $<$), use system resources like the current date and allows using previously provided attribute values for complex decision making.

Figure 2.1 shows a micro process for a very simplistic job application object. Micro steps are rectangles with rounded corners, states are rectangles with sharp corners. Transitions are represented as arrows, for simplicity the micro process does not have backward transitions. The job application consists of first name and surname of the applicant. Then, the applicant has to make a decision when he or she wants to start the job. If the option "later" is chosen, the applicant must provide a date when he or she wants to start instead.

The micro process comprises three states: "personal data", "job beginning" and an end state to terminate the process. The start state is "personal data". The state consists of

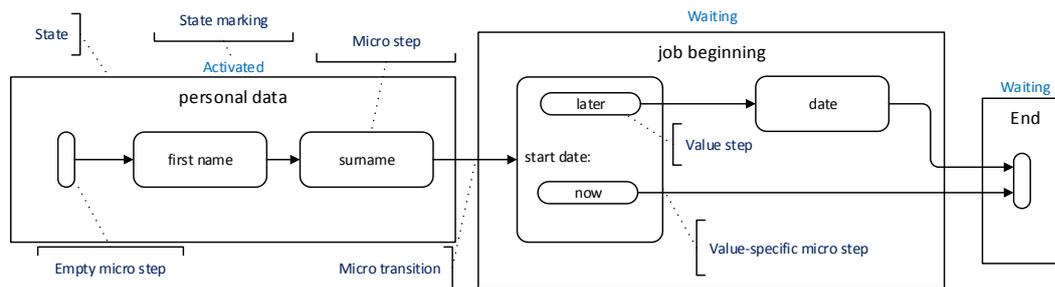


Figure 2.1.: Example Micro Process

an empty micro step that serves as start step. A start step represents the entry point to begin micro process execution. Each micro process can only have one start step. Micro transitions connect the start step to the micro steps for writing the object attributes “first name” and “surname”. A transition connects micro step “surname” with value-specific micro step “start date”.

Value-specific micro steps can be recognized by looking at the value steps inside the micro step. The value steps represent a decision option. In the example, option one is starting to work in the job now, option two for the applicant is to start the job later. He or she then has to provide a date when he or she is able to start. Therefore, a value step “later” is connected by a transition to micro step “date”, which in turn concludes the process by connecting to an end step in an end state. The value step “now” connects directly to the end step as no additional information is required.

By the looks of the process graph, one might get the impression that the execution of a micro process is a rigid and inflexible endeavor. However, PHILharmonicFlows allows writing every object attribute at any point in time. For this purpose, every element of a micro process graph is enhanced by a *Marking*. Markings provide additional information about the current process state and are an essential part to provide flexibility. Based on process structure and markings, *process rules* govern the execution flow of a micro process and provide support for every conceivable user interaction [20, 17]. Both markings and process rules are introduced in detail in Section 2.2.2.

2.2.1. Forms

The states of a micro process instance directly relate to *user forms* in a graphical user interface. Each object attribute and therefore micro step is related to a respective form field. Depending on the object attribute type and the micro step type, a different graphical input element is displayed (e.g., text box for String, check box for Booleans, drop-down menu for value-specific decision making, etc). These form are automatically generated

2. The PHILharmonicFlows Framework

Personal data:

First name

Surname

Job beginning:

Start date

Date

Ok

Figure 2.2.: Example of a Form Representation

based on the micro process. Figure 2.2 shows a form representation of the micro process depicted in Figure 2.1. Although the micro process comprises two states, which would normally translate to two different forms, both states are depicted in one form, since both states only contain two micro steps each.

There exist two possibilities for providing data to an object instance. In the *process-oriented view* data is entered sequentially into forms according to the flow of the micro process. It represents a traditional approach, similar to current workflow system. The *data-oriented view* allows users to enter data at any point in time (cf. Section 2.1.6). It gives a complete overview over all object attributes and relations, which can be manipulated directly. This flexibility is supported by the underlying micro process, as it can react accordingly to data values which were already provided.

A detailed example from the healthcare domain, specifying a data model, related micro processes and user forms can be found in [5].

2.2.2. Markings

Markings convey state information about the process instance during execution. They determine if and which user actions are required. Changes in Markings represent process progression. Each Marking has a specified semantics, which in turn determines what actions are available to progress further in process execution. Table 2.1 contains all Markings for a micro step with a simplified explanation of the Marking semantics. An example execution with Markings displayed can be found in Section 2.3.

Marking	Description
Waiting	The step has not yet been processed, but may be processed later
Ready	The step belongs to the currently activated state and is ready to be processed. Opposite to bypassed.
Enabled	The object attribute value is required and must be mandatorily written.
Activated	An attribute value has been provided.
Blocked	For value specific micro steps: A value step has been provided that does not correspond to any value step predicate. Therefore process execution is halted.
Unconfirmed	The micro step has been provided with a value, but the state it is in is still being processed.
Confirmed	The micro step has been provided with a value and the processing of the state is concluded.
Bypassed	The micro step is in a branch that was not taken during a previous decision. The state, in which the micro step is in, is still processed.
Skipped	Same as bypassed, but state processing is concluded.

Table 2.1.: Markings of a Micro Step

Micro transitions, micro states, backward transitions, value steps and the micro process instance each have their own set of Markings with their own semantics. Since the list of Markings is substantial, Markings which do not belong to micro steps are not displayed in a table. If such a Marking is needed, an explanation will be given when it is first used.

2.2.3. Process Rules

The execution of a micro process instance is driven by process rules. Process rules are responsible for reacting to changes and assigning appropriate Markings to elements of the process graph. Some process rules send messages to the user, who can or must perform an action. Another sort of process rules react to the user input and set appropriate Markings. These three rule types are called *Marking Rules*, *Execution Rules* and *Reaction Rules*. Marking Rules are for internal processing of the Markings and

2. The PHILharmonicFlows Framework

do not affect the outside. Execution Rules trigger on specific Markings, which means the user has to perform an action, e.g., provide a value for an object attribute or make a commitment. Once such action has been performed, a *Reaction Rule* applies the corresponding Markings to the process instances so that execution can continue. A tabular summary of the process rule types can be found in Table 2.2.

	Abbreviation	Input	Output	Color
Marking Rule	MR	markings	markings	
Execution Rule	ER	markings	user input requests	
Reaction Rule	RR	user input	markings	

Table 2.2.: Process Rule Types, taken from [17]

Process rules are not isolated from each other, but work in concert. Process rules can trigger each other iteratively in a cascading fashion. The interactions are diverse and complex, for a graphical representation of all interactions refer to [17].

2.3. Micro Process Execution

To illustrate the use of Markings and process rules, the execution of the process shown in Figure 2.1 is used as an example. For reasons of clarity, micro transition Markings have been omitted unless relevant to the current processing step. Also, some aspects of process execution have been simplified or left out in order to concentrate on the essentials. First, a normal, sequential process execution is demonstrated. Afterward, it is demonstrated how values can be provided up front at any point in time for any object attribute. PHILharmonic Flows provides this flexibility and considers these values during sequential process execution.

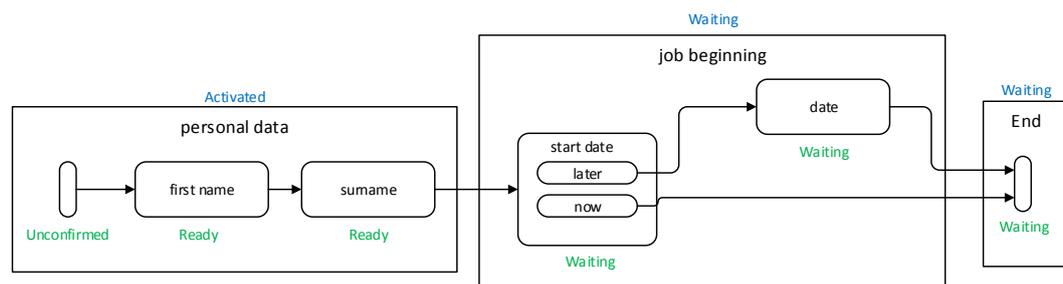


Figure 2.3.: Initialization of a Micro Process Instance

When an object instance is created, a corresponding micro process is instantiated as well. Initially, all object attributes have not been assigned a value and the micro process instance has no Markings. To prepare the process instance for execution, *Reaction Rule RR01* sets initial Markings. Initially all micro steps, value steps and states are marked as Waiting. An exception is the *start state*, which is marked as Activated, which means the state is currently processed. All micro steps in the start state receive the Marking Ready instead of Waiting. The *start step* is marked unconfirmed. The Markings of the micro process instance after the application of Reaction Rule RR01 is depicted in Figure 2.3. Micro step Markings are depicted below the corresponding step, state Markings above the corresponding state. The rule also marks the process instance itself as Running to indicate it is being processed. The process Marking is not displayed.

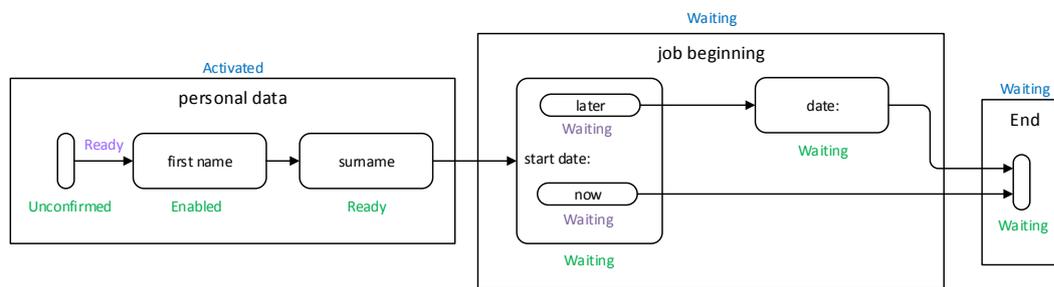


Figure 2.4.: Starting Micro Process Execution

Marking the start step as Unconfirmed triggers *Marking Rule MR01*, which states when a step is marked as Unconfirmed, its outgoing transitions are marked as Ready. The Marking Ready expresses that a transition has been reached and the target steps of the transition can be processed. Therefore, *Marking Rule MR02* marks the target steps of transitions that are Ready as *Enabled*, as illustrated in Figure 2.4, where micro step “first name” is now marked as Enabled. The Marking Enabled for micro steps means that a value for the corresponding attribute is now mandatorily required. Without the value, process execution cannot continue. Once micro step “first name” is marked as Enabled, *Execution Rule ER02* is applied. It tells the user interface that attribute “first name” is mandatorily required. As soon as the user supplies the object attribute with a value, *Reaction Rule RR02* is triggered.

Reaction Rule RR02 states that when a micro step marked as Enabled is supplied a value for its corresponding attribute, it becomes marked as Activated. The user now fills in the corresponding form field with the value “John” and the micro process instance is notified. Since a value is now available, the Reaction Rule triggers and marks the micro step as Activated. The current processing state is presented in Figure 2.5.

2. The PHILharmonicFlows Framework

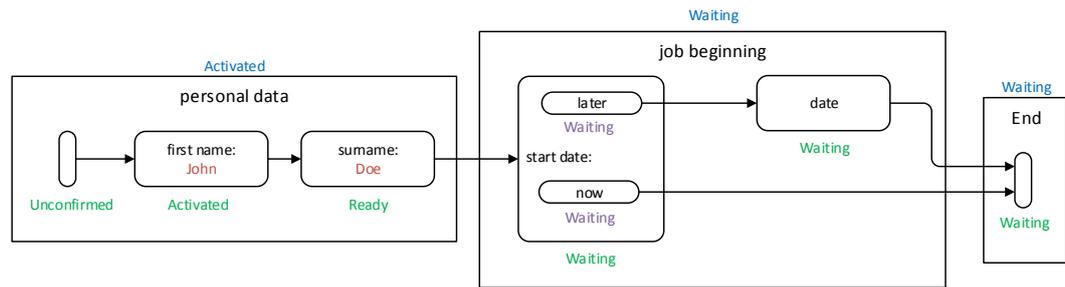


Figure 2.5.: Supplying a Value to an Attribute

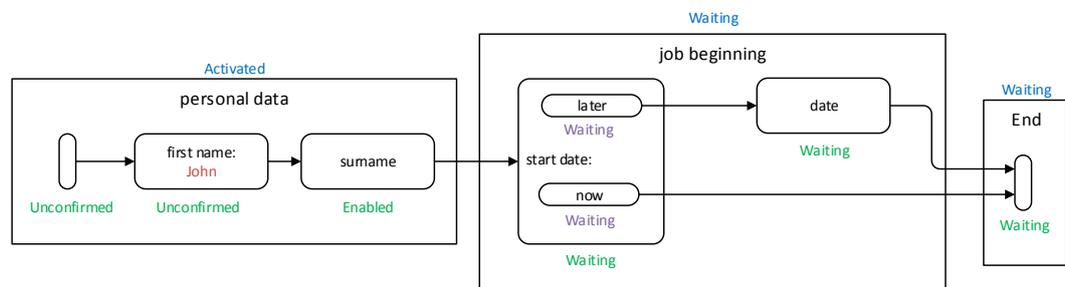


Figure 2.6.: Advancing to the Next Micro Step

After the step has been marked as Activated, the incoming transition is processed. The new Marking for the transition signifies that process execution has progressed past the transition. After the incoming transition has been remarked, the step currently marked as Activated is marked as Unconfirmed by Marking Rule MR05 (cf. Figure 2.6). In turn, this triggers Marking Rules MR01 and MR02 again. As a result, the next micro step “surname” is marked as Enabled. Once the user supplies a value for surname (e.g. “Doe”), the same rules as in micro step “first name” are executed.

At the end of this chain, the micro step “surname” is marked as Unconfirmed and Marking Rule MR01 is triggered, setting the the Marking of the outgoing transitions of the step “surname” to Ready. The transition between step “surname” and value-specific micro step “start date” is external, which means its source and target micro steps belong to different states. In the example, “surname” belongs to state “personal data”, value-specific micro step “start date” to state “job beginning”. According to Marking Rule MR01, the external transition is now marked as Ready (cf. Figure 2.7).

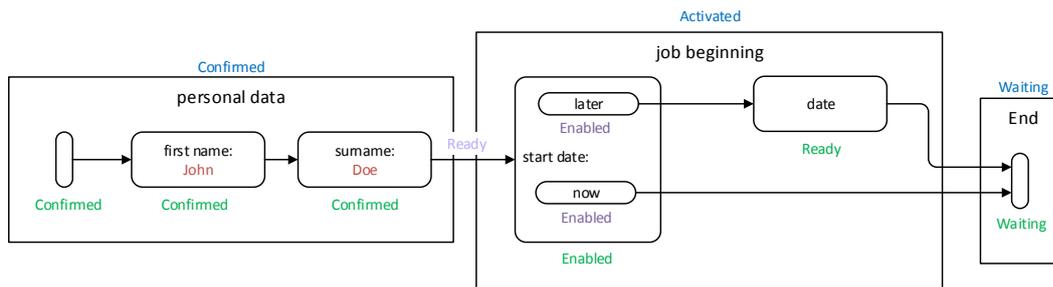


Figure 2.7.: State Change

Since the transition is external, a state change is triggered. The old state “personal data”, which was previously marked as Activated, is now marked as Confirmed. The target state “job beginning” becomes marked as Activated instead. Marking a state as Confirmed triggers additional Marking Rules. Each micro step in the Confirmed state, which has the Marking Unconfirmed, is remarked as Confirmed. And as before, Marking Rule MR02 marks the target step of the external transition, value-specific micro step “start date”, as Enabled. The current state is presented in Figure 2.7.

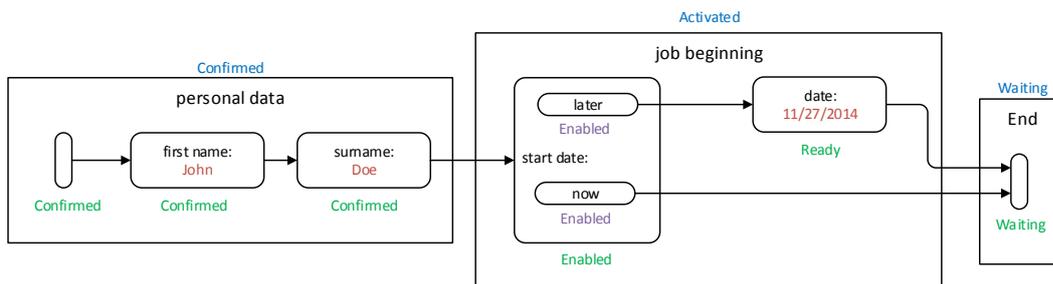


Figure 2.8.: Providing Data to Attribute “date” up front

For demonstration purposes, the sequential execution of the micro process is stopped for the moment. PHILharmonicFlows allows providing data at any point in time while ensuring a correct process execution. To demonstrate this flexibility, a value is provided up front for a step that is not currently processed. The micro step “date” in Figure 2.7 is marked as Ready and has not yet been processed. It also currently has no value. The user now provides a value to the date attribute, e.g. “11/27/2014”. This can be seen in Figure 2.8. Since the Marking of the step is Ready, Reaction Rule RR02 is *not* triggered and the step is not marked Activated. As of this moment, the value is present but not

2. The PHILharmonicFlows Framework

relevant to the current state of process execution. Once process execution progresses to micro step date, the value will be considered accordingly. The details are explained later.

The only remaining attribute that has not yet received a value is “start date”, which is the next processing step for sequential execution. Therefore, sequential process execution resumes with the value-specific micro step “start date”. The difference between a normal and a value-specific micro step is that an attribute value represented by a value-specific micro step is restricted to a predefined, finite set of values. A normal micro step’s attribute value is not restricted. The predefined values are represented as value steps, e.g. “later” and “now” in Figure 2.8. For value-specific micro steps, there exists a set of rules governing the relationship between value steps and their parent value-specific micro step.

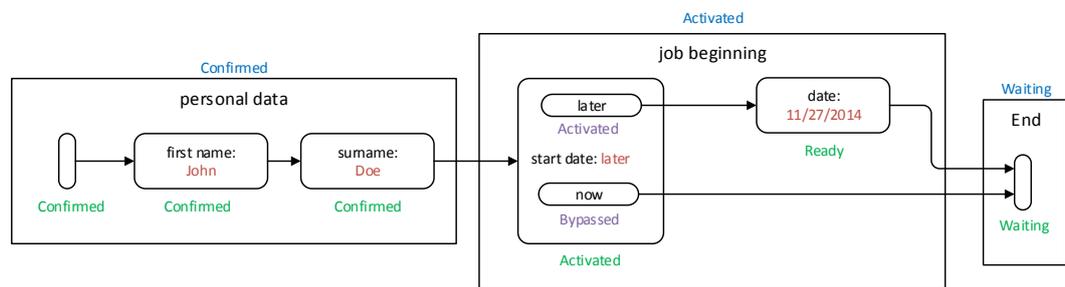


Figure 2.9.: Example Micro Process Execution

One of these rules states that when a value-specific micro step becomes enabled, its value steps become enabled as well. Then, Execution Rule ER02 sends a message to the user interface to request a value for the attribute. However, once a value is set, the value-specific micro step is not immediately marked as Activated. The supplied value must correspond to at least one of the value steps. In the example, the value must either be “later” or “now”. In Figure 2.9, the user has supplied value “later”, which matches to the top value step named “later”. In this case, the matching value step is marked as Activated, which in turn marks the parent value-specific micro step as Activated as well. The other, non-matching value steps are marked as Bypassed. The current status of the process can be seen in Figure 2.9.

The value-specific micro step is re-marked from Activated to Unconfirmed by the same mechanism as for micro steps. A value-specific rule then marks the values steps which are Activated also as Unconfirmed. Marking Rules MR01 and MR02 are also used with value steps, so the outgoing transition from value step “later” is marked as Ready and then micro step “date” as Enabled. However, as the value was provided before, no Execution Rule requesting a value is executed. Instead, a Marking Rule immediately marks micro step “date” as Activated. This is seen in Figure 2.10.

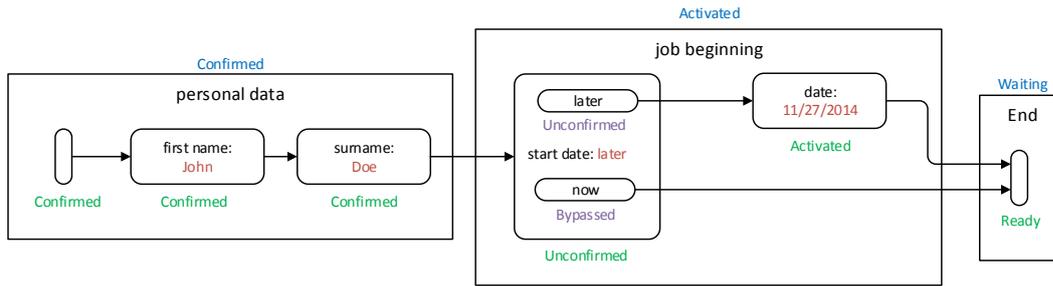


Figure 2.10.: Example Micro Process Execution

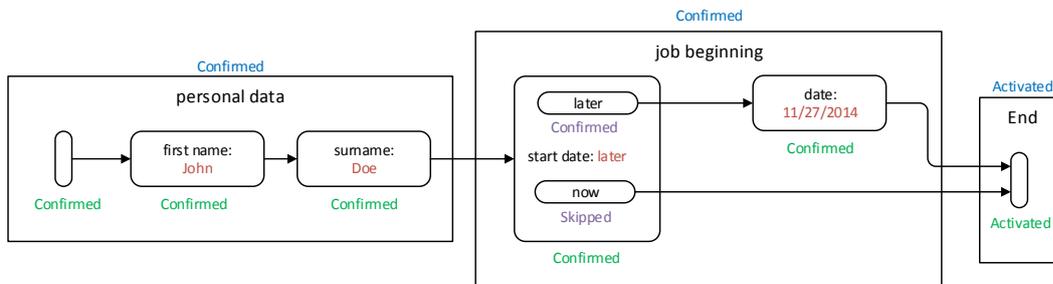


Figure 2.11.: Example Micro Process Execution

The micro step “date” being marked as Activated triggers the rule chain that marks “date” as Unconfirmed. This, in turn, marks the external transition between “date” and the end micro step as Ready. It causes state “job beginning” to be marked as Confirmed, switching the Marking of micro steps in the state from Unconfirmed to Confirmed. All micro steps marked as Bypassed are remarked as Skipped. The “End” state is marked as Activated, which concludes the entire process execution. Therefore, the process instance is marked as Finished. The final markings can be observed in Figure 2.11. As a final remark: Normally, a micro step with an incoming transition marked as Ready should be marked as Enabled. However, the end step is an empty micro step, which causes a rule to remark it immediately as Activated.

2.4. Prototype Architecture

PHILharmonicFlows' prototype implementation [3] consists of three parts, depicted in Figure 2.12. The *Modeling Environment* allows creating object types and model their behavior and interactions with micro and macro processes. The objects and process models are stored in a modeling database. The *Runtime Environment* consists of a *runtime server* that handles all incoming or outgoing communication and the actual *runtime*.

The runtime itself executes micro and macro processes and handles user authorizations and permissions. Process models created in the Modeling Environment need to be deployed to the runtime in order to make them executable. The runtime shares a common database with the *Runtime User Interface*, which is responsible for automatic form generation and displaying work lists.

All three parts can be hosted independently from each other, as all communication is managed by web services (cf. Section 3.2). Connections to the database are managed by Entity Framework, which is an object-relational mapping framework for .NET.

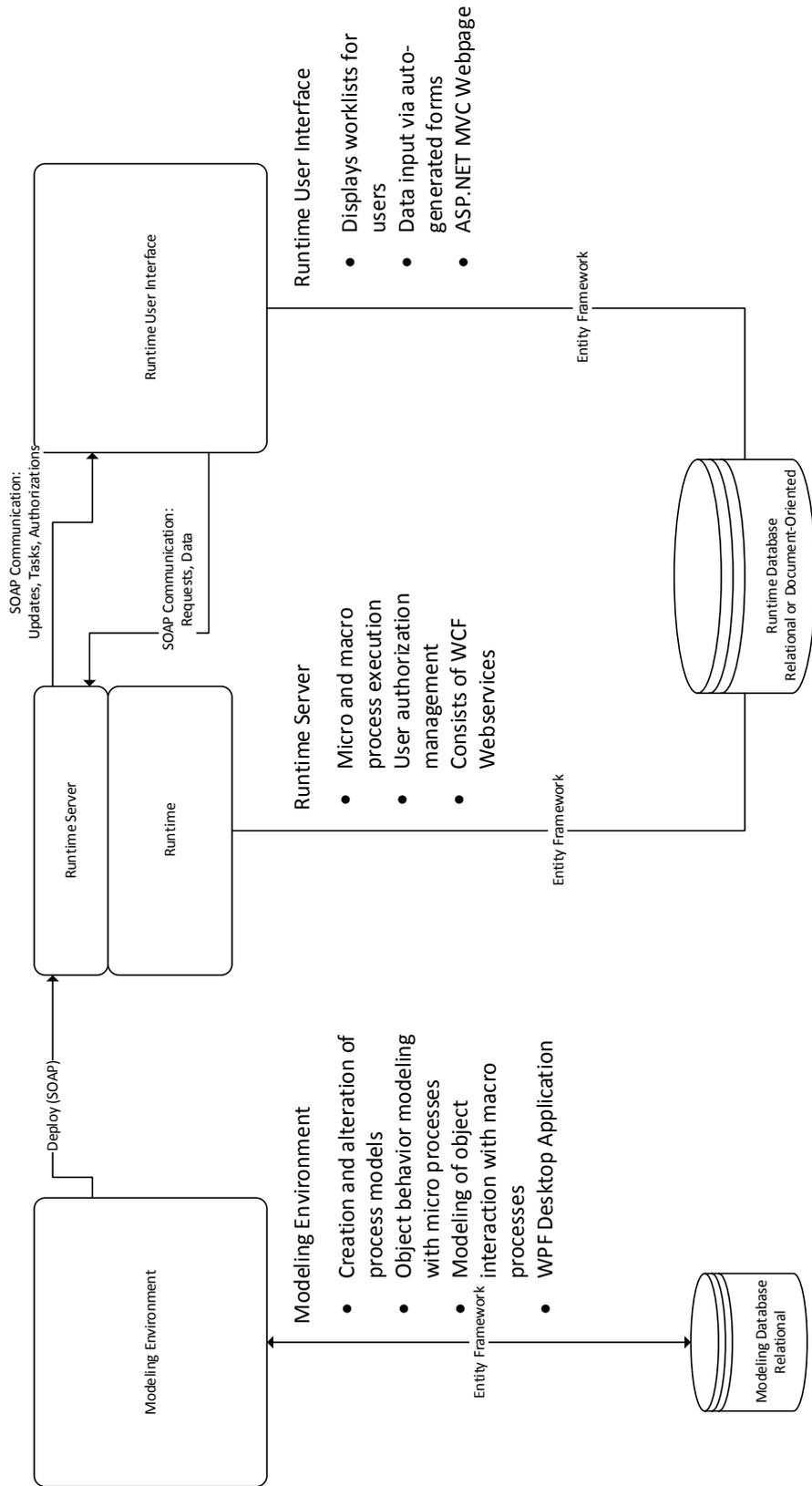


Figure 2.12.: PHILharmonicFlows Architecture

2.5. Summary

This Chapter has presented the concepts of an object-aware process management system. Of particular interest in the context of this thesis is object behavior, which is captured by micro processes. An example provided insight into object behavior at runtime by executing such a micro process. The driving force behind the execution are process rules and Markings, which let process execution progress. Process rules and micro process execution are essential parts of the runtime of PHILharmonicFlows. To provide context, a complete overview over the architecture of the prototype implementation was given in Section 2.4.

3

Programming Concepts

This chapter gives a basic explanation of advanced programming devices as well as C# [28] specific concepts used in the implementation of PHILharmonicFlows. It is assumed that the reader has an understanding of object-oriented programming and the related concepts of information hiding, encapsulation, inheritance and parametric polymorphism (commonly known as *generics* in object-oriented languages). As C# allows for a more functional programming style, readers with a basic knowledge of functional programming are at an advantage. However, functional concepts are explained as it is necessary for the understanding of the code. Throughout this thesis, programming related identifiers like class names are written in *CamelCase*, also when object instances are meant (plural-s).

3.1. C# in general

Microsoft's C# [28] is a multi-paradigm programming language with full object-orientation, meaning everything is an object, including primitives like int/float and functions, with strong typing. It includes the functional paradigm, supporting higher-order functions and closures. Its syntax resembles that of the C programming language with curly braces to group statements and semicolons to denote the end of a statement. Objects are declared in a class and instantiated by calling a constructor function. Objects can inherit from other objects, although multiple inheritance is not supported. Instead, interfaces are used to offer some of the functionality of true multiple inheritance.

As part of the .NET-Framework [26], C# is compiled into *Common Intermediate Language(CIL)*, a language that is the foundation of all .NET languages and enables language interoperability. The CIL code together with the required resources is stored in an assembly. Upon execution, an assembly is loaded into the *Common Language Runtime(CLR)* and is then just-in-time-compiled to the native machine code of the operating system. The implementation of the PHILharmonicFlows runtime uses version 4.5 of the .NET Framework. Unfortunately this prohibits the use of *Mono*, an open source .NET Framework implementation on Linux, to run PHILharmonicFlows on a Linux operating system. The reason is that Mono only supports .NET features up to Version 4.0, excluding WPF and with incomplete support for WCF (cf. Section 3.2). On November 12, 2014, Microsoft has announced that they will publish the .NET Framework under an

3. Programming Concepts

open source license and will work with the Mono project to port .NET to Linux and Mac OS [23]. Hence, it may become possible in the near future to host PHILharmonicFlows on a Linux OS.

The development environment used for this thesis is Visual Studio 2013 with Team Foundation Server (TFS) as a version control system. Visual Studio provides intelligent code completion with *IntelliSense* [27] and other useful features for a .NET/C# Developer.

3.2. Windows Presentation Foundation and Windows Communication Foundation

Windows Presentation Foundation (WPF) [31] is a Graphical User Interface framework for Windows using an XML-based language called XAML [29]. PHILharmonicFlows' modeling environment utilizes WPF as graphical front end. The communication among the different modules of PHILharmonicFlows (modeling, runtime and user interface) is realized with the Windows Communications Foundation (WCF) [30]. It enables creating Webservices specified by WSDL [7]. This allows establishing loosely coupled clients and servers. Loose coupling allows deploying all components of PHILharmonicFlows flexibly each on a different machine or all on the same machine or any combination in between. Developers can also rewrite the user interface in other, non .NET languages as long as these languages or frameworks understand WSDL.

Both WPF and WCF are only of minor importance to the topic of the thesis directly, but are needed to understand the big picture of PHILharmonicFlows. The interface of the runtime to the modeling environment and to the UI is implemented as a WCF service.

3.3. Namespaces

Namespaces are used to group code with similar functionality. For example, everything related to the String-class is found in namespace *System.String*. It is also used to distinguish classes and methods with identical identifiers. Namespaces can be hierarchically organized, for example the namespace *Runtime.ProcessRules* contains the children *Runtime.ProcessRules.ExecutionRules* and *Runtime.ProcessRules.MarkingRules*.

3.4. Partial Classes

Partial classes can be used to distribute the source code of a single class across multiple files. This is indicated with the keyword *partial* in front of the class name. This feature is particularly useful for large classes with hundreds of lines of code or to combine auto-generated code with custom modifications. The auto-generated part can be replaced or updated without losing the custom part. Files containing the partial class definitions can

be named independently from the actual class name. For example, there is a class for unit testing another class with a large number of different testing functions incorporated. Since there exist numerous methods that must be tested and multiple tests that cover all relevant cases for each of them, every group of tests can be put into its own file. The file can be named accordingly without having to create separate test classes for each group or a single file with hundreds of test methods.

3.5. Interfaces

Interfaces work similarly to interfaces in other object-oriented languages. However, in C#, the programmer can also declare properties as part of the interface. By convention, interfaces are prefixed with a capital i, for example *IInterface*.

3.6. Properties

Properties are members of a class that expose private fields for reading and writing their values. Syntactically they are used like public fields, but in fact they are syntactic sugar for *getter* and *setter* accessor functions. For simple properties, get and set can be auto-implemented by the compiler. Properties with such get and set accessors are called *autoproperties*. For more complex requirements the get and set accessors can be customized. The customizations can include complex verification code, raising of events and computations. However, to have customized setters and getters the property must have a private *backing field* for storing the value.

The code in Listing 3.1 is taken from the `MicroProcessInstance.cs` file and shows the declaration of an autoproperty and a custom property. Line 5 is a backing field called `_marking` belonging to the custom property declared in Line 8. The get accessor is default for custom properties, whereas set has custom code that raises an event if the `Marking` is set to `Waiting`. The variable `value` in this context denotes the parameter value that will be assigned to `_marking`. Together with methods, fields, constants and others, properties are *members* of a class.

3. Programming Concepts

```
1 //Autoproperty
2 public int Id { get; set; }
3
4 //backing field
5 private MicroProcessMarkings _marking = MicroProcessMarkings.None;
6
7 //custom Property, raises an event if the marking changes
8 public MicroProcessMarkings Marking
9 {
10     get { return _marking; }
11     set {
12         if (_marking == value) return;
13         _marking = value;
14         switch (value)
15         {
16             case MicroProcessMarkings.Waiting:
17                 RaiseProcessRuleEvent(ProcessRuleType.ProcessInitiatedRr01);
18                 break;
19         }
20     }
21 }
```

Listing 3.1: Autoproperty and custom property

3.7. Collections

Collections are used to store a group of objects which size shrinks or grows dynamically. They reside in the *System.Collections* namespace. For groups of objects that all have the same type, *System.Collections.Generic* provides strongly typed collections that are type-safe, most notably the *List<T>* collection. The type parameter *T* identifies the type of the objects stored in the collection. Listing 3.2 shows how to use an *object initializer* to create a *List<string>* and one possibility to iterate over its elements.

```
1 // Create a list of strings
2 var animals = new List<string>() { "monkey", "elefant" };
3 animals.Add("cow");
4 // Iterate through the list.
5 foreach (var animal in animals)
6 {
7     Console.Write(animal + " ");
8 } // Output: monkey elefant cow
```

Listing 3.2: Collection Example

The *var* keyword denotes an implicitly typed variable declaration, which aides in conciseness since there is no need to repeat the type *List<string>* that is already declared on the right side.

3.8. LINQ and Fluent Interfaces

Language-Integrated Query (LINQ) is a feature, introduced in .NET 3.5, that extends C# with query expressions. These queries can be used on objects, object collections, XML documents, relational databases and other sources, providing a concise and efficient way to retrieve data. It was influenced by SQL and the functional programming language *Haskell*. LINQ has a query syntax similar to that of a standard SQL statement. The change from “SELECT FROM WHERE” (SQL) to “FROM WHERE SELECT”(LINQ) was done to provide better IntelliSense support in Visual Studio. Additionally, LINQ also defines methods to filter, aggregate or project data, which are called *standard query operators*. These methods can also be used in a fluent style to achieve the same result as with a normal LINQ-Query. Most prominent among these methods are *Where* (filter) and *Select* (projection) which are also used extensively in the Process Rule Framework (cf. Section 4.2) and the Definition of the Process Rules (cf. Section 4.2.1).

Fluent Interfaces [12] are an implementation of an API that aims to provide more readable code. Usually, this is achieved by chaining methods together. The example in Listing 3.3 presents a LINQ query performed twice in both SQL style (Line 1) and fluent style (Line 5) on an undefined collection of objects named *SomeCollection*.

```

1 var results = from c in SomeCollection
2               where c.SomeProperty < 10
3               select new {c.SomeProperty, c.OtherProperty};
4
5 var results = SomeCollection.Where(c => c.SomeProperty < 10).Select(c
  => new {c.SomeProperty, c.OtherProperty});

```

Listing 3.3: LINQ Queries

In the fluent-style query, the higher-order function *Where* first filters the collection based on a predicate and then *Select* takes the values of *SomeProperty* and *OtherProperty* and wraps them into a new object which is stored into the collection *results*. The designation fluent interface is based on the fact that the query can be read much like a sentence in a natural language. The Process Rule Framework uses a fluent interface to provide a fast and easy way to create process rules. The specifics on how the fluent interface is implemented and how a process rule works can be found in Section 4.2.

3.9. Lambdas

Lambda functions (named for the λ -calculus [8] by Alonzo Church), also called anonymous functions or just lambdas) are function definitions that do not have an identifier. In C# they are declared as $x \Rightarrow x * x$. The \Rightarrow indicates the lambda function, x is an input variable and $x * x$ is the function body. Lambda functions can have an arbitrary number of input variables and no or one return value. When used in context, the type inferencer of .NET can usually infer the types of input and output. However, lambda function declarations cannot be bound to an implicitly typed variable using *var*. Instead, the function $x \Rightarrow x * x$ is bound to the identifier f by the following declaration: $Func<int, int> f = x \Rightarrow x * x$. The first type parameter of *Func* defines the type of the only input parameter whereas the second one is the return type of the function.

Anonymous functions that do not have a return value (void or ()) are *Actions*. Their declaration only incorporates the types of the input parameters. Lambda functions are no different from named functions in functionality, since each lambda can be replaced with an equal named function or method. Usually, anonymous functions are more convenient to use, especially if the function declaration is very small.

In Listing 3.3, the arguments of *Where* and *Select* are lambda functions which either define a predicate for the filtering operation or a projection to a new object. Lambdas are very versatile and can even reference identifiers from the enclosing scope, for example a *List* or a complex object, which makes a lambda function a *closure*. The body of a lambda function is not limited to a single expression, multiple statements can be grouped together with curly braces, similar to named functions.

3.10. Delegates and Events

Events in programming are actions or occurrences that are caused outside of the current scope by an asynchronous external activity and may be handled by the program. Typical examples for events are keystrokes or the click of a mouse button. In C#, an event is nothing more than the invocation of a function, called *event handler*, which has a specified signature.

In order to understand how events work, it is necessary to understand the concept of *delegates*. Basically, a delegate is a *pointer (or reference) to a function*. A delegate definition is introduced by using the keyword *delegate* and then defining a signature with input types and return type that a function must have in order to be referenced by the delegate.

Listing 3.4 shows an excerpt from the *MicroProcessInstance.cs* class that shows the usage of events. Line 2 shows the signature that a function has to fulfill in order to serve as a *ProcessRuleEventHandler*. Event handling functions always have return type *void*. By contrast, input parameters may be defined freely. The Microsoft Developer Network

(MSDN) [30], however, recommends always providing the sender as a first parameter. If the event wants to pass additional data to the event handler, it encapsulates them in an object derived from the *EventArgs* class. In the example, the *ProcessRuleEventType* class is derived from *EventArgs*. The event is declared in Line 5 with the keyword *event*, followed by the signature definition for the handlers and then the identifier. An event can only be raised inside the class where it was declared, called a *Publisher class*. The event is raised by calling *RaiseProcessRuleEvent* with appropriate parameters. In particular, each handler is invoked with a multicast and executes its code for handling the event.

```

1 //Event contract definition
2 public delegate void ProcessRuleEventHandler(object sender,
    ProcessRuleEventType type);
3
4
5 public event ProcessRuleEventHandler ProcessRuleEvent;
6
7 //used to dispatch an event to the event handlers (basically invoke
    them methods
8 protected virtual void RaiseProcessRuleEvent(ProcessRuleType type)
9 {
10     var handler = ProcessRuleEvent;
11     if (handler != null) handler(this, new ProcessRuleEventType(type));
12 }

```

Listing 3.4: Delegates and Events

In Listing 3.5, Line 8 shows a subscription to an event of a *MicroProcessInstance*. The operator += adds a new delegate of type *ProcessRuleEventHandler* to the event that points to the handling function *HandleProcessRuleEvent*.

```

1 void HandleProcessRuleEvent(object sender, ProcessRuleEventType type)
    )
2 {
3     //Do something
4 }
5
6 //Creating a delegate(pointer) to HandleProcessruleEvent
7 //and add it to ProcessruleEvent's list of "Event Handlers".
8 processInstance.ProcessRuleEvent += new ProcessruleEventHandler(
    HandleProcessRuleEvent);

```

Listing 3.5: Subscribing Event Handlers

Essentially, an event in C# is a list of methods with the same signature. The list is stored in the class which declares the event. When an event is raised, the list is iterated and each method is called with the current parameter values. Assigning an event handler is an easier, more elegant way of adding a method to that list, with delegates being the glue to hold event and methods together.

3.11. Extension Methods

Extension methods enable the programmer to add functionality to an existing type without modifications; i.e. deriving a new subtype. This is especially useful for types that have been sealed so they cannot be subclassed. Extension methods are static methods, but a special syntax enables them to be called like normal instance methods. The standard LINQ query operators (*Select*, *Where* etc.) are extension methods, extending the functionality of *System.Collections.IEnumerable* and the generic variant, *System.Collections.IEnumerable<T>*. Various extension methods take lambdas as a parameter, although this is not mandatory. Extension methods are used in the Process Rule Framework of PHILharmonicFlows to provide easy extensibility. For further details please refer to Section 4.2.4

3.12. Expressions and Expression Trees

Expressions are sequences which can be evaluated to a single value. Expressions consist of one or more operands and zero or more operators. For example, lambda functions in C# are expressions, hence they are mostly referred to as lambda expressions and not lambda functions. Expressions can range from very simple to very complex. Complex expressions are mainly produced by the nesting of other expressions. Evaluation of such expressions is governed by associativity and operator precedence. For example the arithmetic expression $4 * 5 + 3$ evaluates to 23 since $*$ takes precedence over $+$. But $4 * (5 + 3)$ evaluates to 32, because the parentheses change the precedence of the enclosed expression.

```
1 // The parameter for the lambda expression.
2 ParameterExpression parameterExpression = Expression.Parameter(typeof
   (int), "x");
3
4 // This expression represents a lambda expression
5 //That multiplies its parameter with itself
6 . LambdaExpression lambdaExpression = Expression.Lambda(
7     Expression.Mult(
8         parameterExpression,
9         parameterExpression
10    ),
11    new List<ParameterExpression>() { parameterExpression}
12 );
```

Listing 3.6: Creating a Lambda Expression with Expression Trees

Expressions can be represented using a tree data structure, in which each node is an expression and edges represent the relationships between them. There are various types of nodes, a non exhaustive example list would be *ParameterExpression* for representing a function parameter, *BinaryExpression* for expressions with a binary operator and

LoopExpression for creating loops. These *expression trees* can be compiled and the resulting code executed at runtime. This allows the creation of dynamic queries by the user to get data from queryable data structures. An expression tree also allows for code to be persisted in a database. Such expressions can also be manipulated dynamically by methods provided by the *ExpressionTree* class. Listing 3.6 shows the creation of the lambda function $x \Rightarrow x * x$ explicitly with expression trees.

Expression trees are a very powerful and versatile programming tool and are used in the Process Rule Framework together with a fluent interface to provide a consistent and easy-to-use way to create process rules.

3.13. Summary

This chapter explained the programming concepts and devices necessary to understand the code-heavy chapters 4 and 5. Delegates and events are necessary to understand the event-driven application of process rules with the Process Rule Manager, explained in Chapter 5. Of particular importance are expressions and expressions trees, which form the basis for the high-level abstraction of process rule definitions presented in Chapter 4. These are provided by the Process Rule Framework. Both Process Rule Framework and Process Rule Manager use lambdas to allow a concise and easy-to-read programming style.

4

The Process Rule Framework

This section analyzes the process rules presented in [17] and derives a common template to all process rules. A process rule template consists of a target object, a set of preconditions and a set of effects. Section 4.2.4 explains how a precondition is defined and the inner workings to evaluate a precondition. The definition of effects is explained in Section 4.2.6. The section also explains the technical details how an effect definition is translated to a change of the state of a micro process. The process rule template and the definition methods for preconditions and effects form the Process Rule Framework.

4.1. Runtime Model

The PHILharmonicFlows Model Editor [6] creates micro process models based on design-time model classes. However, these micro process models can not directly be executed. Execution requires an object instance, based on runtime model classes. Deployment makes the micro process model known to the runtime server. Once a model has been deployed, users can create object instances, and the server instantiates a corresponding micro process instance. Micro process instances are composed of runtime classes which resemble their design-time counterparts but contain additional properties and methods necessary for the the process instance to be executable. Most prominent among those additional properties are Markings [17]. Figure 4.1 shows a class diagram of the most important runtime classes referring to micro processes along with their essential properties. For reasons of succinctness and clarity, properties of minor importance and methods altogether have been excluded, as well as UML associations between the classes. The associations can be inferred fairly easily by looking at the class properties.

These classes are used to represent the micro process graph which is used to execute a micro process instance. The dotted horizontal line separates *value properties* from *reference properties*. Reference properties are used to navigate the graph. For example, from a micro step A it is possible to navigate to its successor micro step B via the transition linking A and B. It is also possible to first navigate to the micro process instance and then look for B in the *MicrostepSet* of the process instance. Some of the references are mutual and form a reference cycle in graph. An example for a mutual reference is in the transition which links A and B. The transition references its target step,

4. The Process Rule Framework

which is B, and in turn micro step B references the transition as an incoming transition (cf. classes *MicroStep* and *MicroTransition* in Figure 4.1) . The event-driven execution model for the process rules requires that from every point in the graph, every other point can be reached via at least one path (refer to Chapter 5 for details). To keep process rule definitions succinct, in general a process rule developer has multiple options on how to traverse the graph and can choose the shortest or clearest path.

In the source code, runtime classes differentiate themselves from design-time classes by replacing the suffix *-Type* with *-Instance* in its class name. So the runtime class for *MicroProcess* is actually called *MicroProcessInstance*, the design-time class is called *MicroProcessType* . However, this thesis focuses on the runtime of PHILharmonicFlows, which naturally uses the runtime classes. So for reasons of brevity the *-Instance* is omitted. Unless stated otherwise, throughout this thesis any class name mentioned in Figure 4.1 always identifies a runtime class. Design-time classes are identified explicitly by its suffix *-Type* .

4.1. Runtime Model

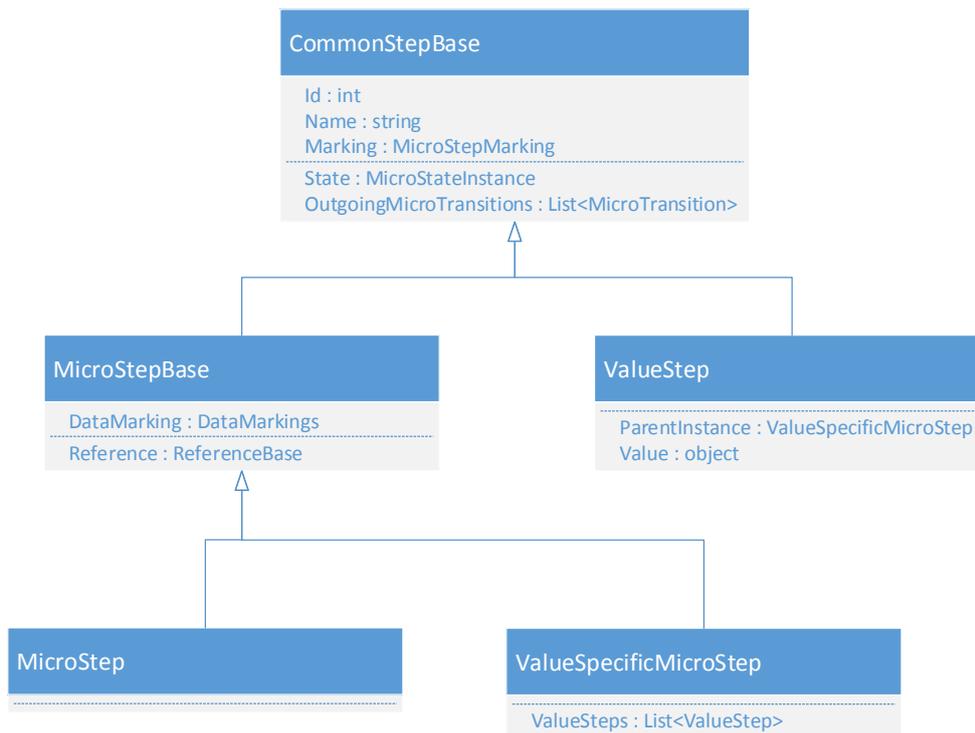
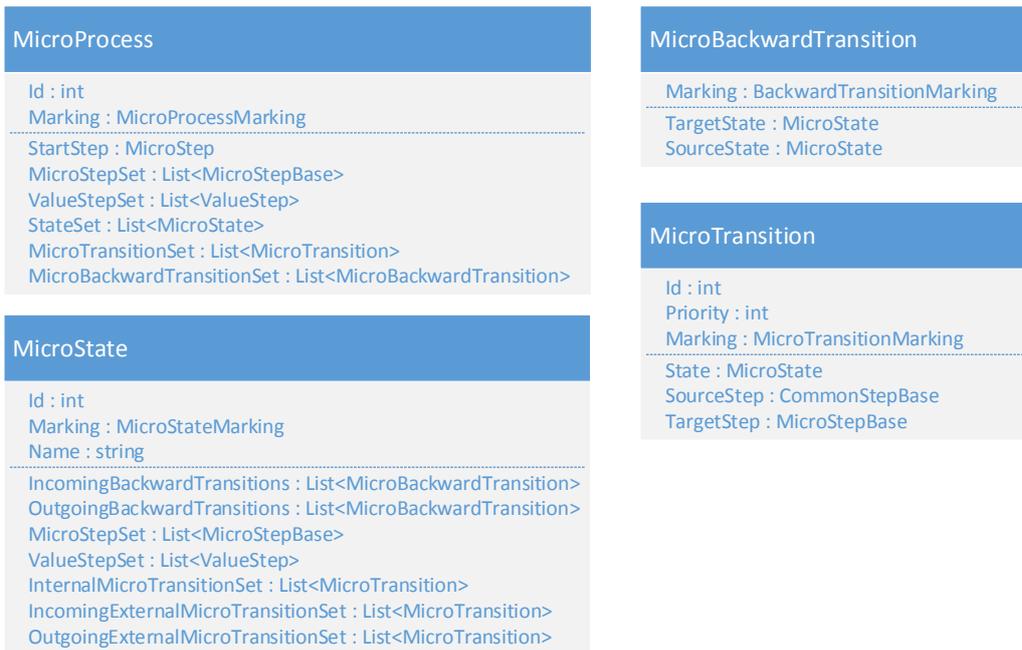


Figure 4.1.: PHILharmonicFlows Runtime Classes, -Instance Omitted

4. The Process Rule Framework

4.2. Implementation

Process Rules are the core part of micro and macro process execution. The *Process Rule Framework* allows to easily create process rules, hiding away all common code so the rule creator can focus on essential tasks while creating rules. Therefore a set of specific goals for the framework was formulated.

4.2.1. Process Rule Analysis

A thorough analysis revealed that for the most part, the process rules formulated in Chapter 8 of [17] had a common template to them. Figure 4.2 shows a Marking Rule as presented in [17] which will serve as an example for extracting the template. Initially the process instance, its properties and sets are defined. Then a statement is made, defining to which part of the process instance the rule refers (e.g., state, transition or micro step). In this case it is a micro step. Usually, a condition follows narrowing the scope even more, in this case, for instance, the micro step must not belong to the currently activated state. This condition applies to all parts of the rule defined in the next part. The next part consists of one or more statements that describe the effects the process rule has on elements of the process graph. In this case it affects *Micro Transitions* and *ValueSteps*.

Marking Rule (MR17: Resetting micro steps and value steps):

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

$\forall \text{micStep} = (\text{ref}, \text{ValueSteps}) \in \text{MicStepSet}$ with $\text{state} \in \text{StateSet} \wedge M_{\text{State}}(\text{state}) \neq \text{ACTIVATED}$:

1. $\forall \text{micTrans} \in \text{intrans}(\text{micStep})$ with $M_{\text{MicTrans}}(\text{micTrans}) = \text{WAITING}$; $M_{\text{MicStep}}(\text{micStep}) := \text{WAITING}$;
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 marked as WAITING.
2. $\forall \text{valueStep} \in \text{ValueSteps}$ with $M_{\text{MicTrans}}(\text{micTrans}) = \text{WAITING}$; $M_{\text{MicStep}}(\text{micStep}) := \text{WAITING}$;
i.e., if a micro step is marked as WAITING, all corresponding value steps will also be marked as WAITING.

Figure 4.2.: Marking Rule (adapted from [17])

The pattern is that a certain type (in the example, *Micro Transition* and *ValueStep*) has to meet a set of preconditions in order for a set of effects to be applied. A typical precondition is that something must have a specific Marking. A typical effect is changing that Marking. Please note that preconditions and effects are not restricted to the initial type, but can be for any type referenced by the initial type, which will be detailed later on. In statement 1) of Marking Rule MR17 the type *Micro Transition* must meet the preconditions “Is marked as Waiting” and “does not belong to the state marked as Activated”. Thereby, an effect on its target *MicroStep* can be applied. The effect is that the target micro step will be marked Waiting if all of its incoming transitions are marked as Waiting. The effect description contains another precondition indirectly related to the *Micro Transition*. In order for the target micro step to be marked as Waiting, all

4.2. Implementation

three preconditions must evaluate to true at runtime. A similar analysis is valid also for statement 2), but will not be discussed here, as no new information is gained from it.

Based on these observations a generic rule template for a Marking Rule, that captures the observed pattern, is as follows: An implicit precondition for any rule would be that it is only applicable to a certain type (i.e., *MicroTransition* in the example case). In order for a set of effects to be applied, a set of preconditions must be met, including the type restriction. Figure 4.3 illustrates the structure of the process rule template.

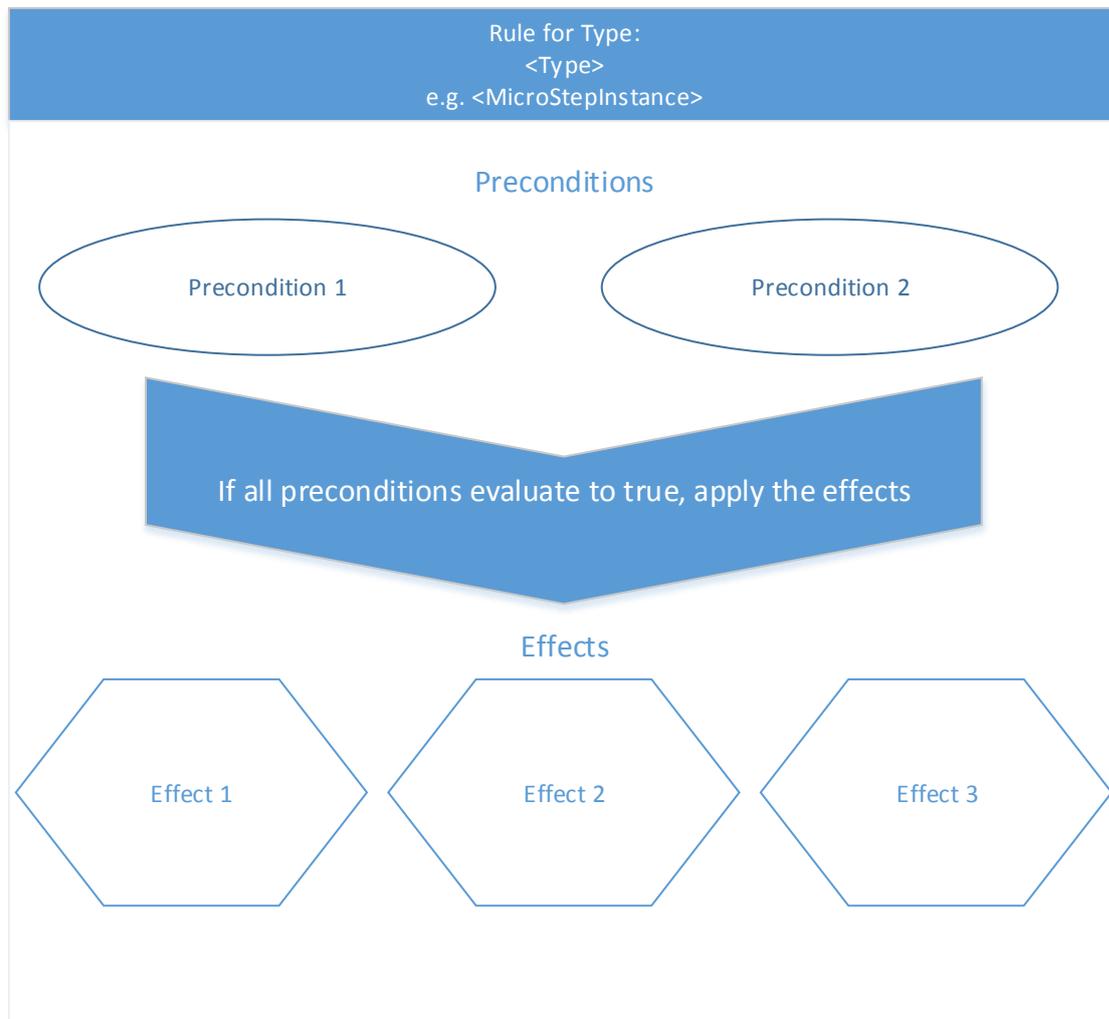


Figure 4.3.: Process Rule Template

4. The Process Rule Framework

This template is valid for Reaction Rules and Execution Rules as well. However, the nature of the precondition is different with Reaction Rules, as is the nature of the effect with Execution Rules. A precondition for a Reaction Rule is an event raised by the process instance when data or a user commitment is available. The effect for an Execution Rule is an event that requests data from the user interface.

Both effects and preconditions need not to point directly to the type, but can also point to other parts of the process instance. Regarding the example, this can be observed in "the micro step does not belong to the state marked as Activated" and "target step of the transition is marked as Waiting", where the preconditions refer to a *MicroState* and a *MicroStep* respectively.

As an entry point into the process graph, the template receives only a reference to an instance of its type, i.e., *MicroTransition* in the example. In order to navigate to the state to evaluate if the precondition are true, there are two options (cf. Figure 4.1). First, the process graph must have a path from micro step to state via reference properties. The second option is to take the *StateSet* of the micro process and search it for the appropriate state. Searching requires iterating over all states and possibly over all steps in the respective states, to determine if the step belongs to the state. Both options are also valid for the general case, not just for the given example.

Out of these two possibilities, the former one presents more advantages. Having a lot of rules that constantly iterate over every set of a micro process instance is not a good idea, considering that the set size and the amount of micro process instances can go up into the hundreds or even more. Furthermore, choosing the first option allows for a very efficient event-driven rule application. If, for example, the Marking of a *MicroTransition* changes, an event is raised and the Process Rule Manager can look up the appropriate rule and apply it. Details on event-driven rule application and the Process Rule Manager are explained in Chapter 5.

4.2.2. Goals

For the Process Rule Framework, the following design goals have been formulated.

- **Maintainability:** Process rules should be easy to create and to alter. The addition of new features to PHILharmonicFlows might require new rules and during the current development practical problems might arise that make a rule alteration necessary.
- **High Abstraction Level:** Process rules should be definable on a fairly high abstraction level, with scarce to no use of low-level code. It should be obvious at a glance what a rule does and what its purpose is, even without documentation.
- **Extensibility:** New process rules might require new features currently not implemented in the framework. It should be possible to easily modify the framework to provide those new features.

- **Good Object-Oriented Design:** Process rules should be objects, not functions. This helps to avoid redundant code and increases abstraction. A single method per rule with a lot of if-then-elses repeated for every rule would be a maintainability nightmare.
- **Consistency:** Designing rules with the framework should be consistent in syntax and behavior of the tools.

4.2.3. Example Rule

The Marking Rule depicted in Listing 4.1 shows two possibilities for defining either a precondition or an effect. A Marking Rule is a class that inherits from *AbstractEffectRule<T>* (Line 1). The type parameter *T*, called the *target type*, specifies the type the Marking Rule refers to, a *MicroStep* in case of the example. Generics ensure that everything is type-safe and additionally provide IntelliSense support when programming in Visual Studio. Reaction and Marking Rules both inherit from *AbstractEffectRule<T>*, though an Execution Rule inherits from *AbstractExecutionRule<T>* due to the different nature of effects. A *target object* is an object of type *T* to which the rule is applied in context of this explanation.

```

1 public class ExampleMarkingRule : AbstractEffectRule<
    MicroStepInstance>
2 {
3     public ExampleMarkingRule()
4     {
5         PreconditionFor(step => step.Marking).IsMarked(
            MicroStepMarkings.Activated);
6
7         PreconditionForEach(step =>
8             step.OutgoingTransitions.Select(x => x.Marking))
9             .IsMarked(MicroTransitionMarkings.Activated);
10
11        EffectFor(step => step.Marking).AssignMarking(MicroStepMarkings.
12            Confirmed);
13
14        EffectForEach<MicroTransitionInstance, MicroTransitionMarkings>(
15            step =>
16            step.OutgoingTransitions.Select(trans=>trans.Marking))
17            .AssignMarking(MicroTransitionMarkings.Confirmable);
18    }
19 }

```

Listing 4.1: Example Marking Rule

There are four different creation expressions, two for each precondition and effect. Due to technical reasons preconditions and effects are split into a version for single value and one for collections. In Line 5 and 7, where the *Each*-suffix identifies the expression

4. The Process Rule Framework

for collections. In the following section, precondition and effect creation is explained in detail.

4.2.4. Preconditions

When discussing preconditions (and effects) in the context of the Process Rule Framework, it is important to distinguish between the front end and the back end of a process rule. The front end consists of creation methods like *PreconditionFor* and *EffectFor*. A creation method is the high level abstraction of a precondition or effect definition. Upon instantiation of a process rule, the creation methods are evaluated and the defined preconditions and effects are converted to objects which provide the actual functionality. These objects form the back end of the process rule. At first the syntax and semantics of a creation expression are explained, then the workings in the back end. Figure 4.4 shows an overview of all classes used for the Preconditions.

The Process Rule Framework differentiates between a front end and a back end, and most front end methods correspond to an object in the back end. Table 4.1 shows a mapping of front end to back end. The conversion happens when a process rule is instantiated.

Front end	corresponds to	Back end
<i>PreconditionFor</i>	->	<i>Precondition</i>
<i>PropertyFunction</i>	==	<i>PropertyFunction</i>
Predicate (e.g. <i>IsMarked</i>)	->	<i>IConditionItem</i>
-		<i>PreconditionContext</i>

Table 4.1.: Front-End to Back End Mapping

PreconditionFor is a static method that initiates a declaration of a precondition (cf. Listing 4.1). The method takes one parameter which is a lambda expression. The lambda expression is called the *PropertyFunction*, since it specifies the property that should satisfy a predicate. The predicate (e.g., *IsMarked*) is defined after the creation method. The *PropertyFunction* takes the target object as argument and navigates the graph via a path of reference properties and returns a property of type *TProperty*. For example, the *PropertyFunction* in Line 5 of Listing 4.1 goes directly to the Marking property of the step itself, where *TProperty* is the type *MicroStepMarkings*.

Once a property has been selected, it can be checked via predefined predicate expressions, e.g., *IsMarked*. *IsMarked* is realized with an extension method that takes one argument of type *TProperty* and checks whether the argument is equal to previously selected property. Although *TProperty* is generic, the name *IsMarked* suggests it is only for comparisons with Markings. The reason for the generics here is that Markings have different categories. There are separate Marking enums for steps, transitions and other

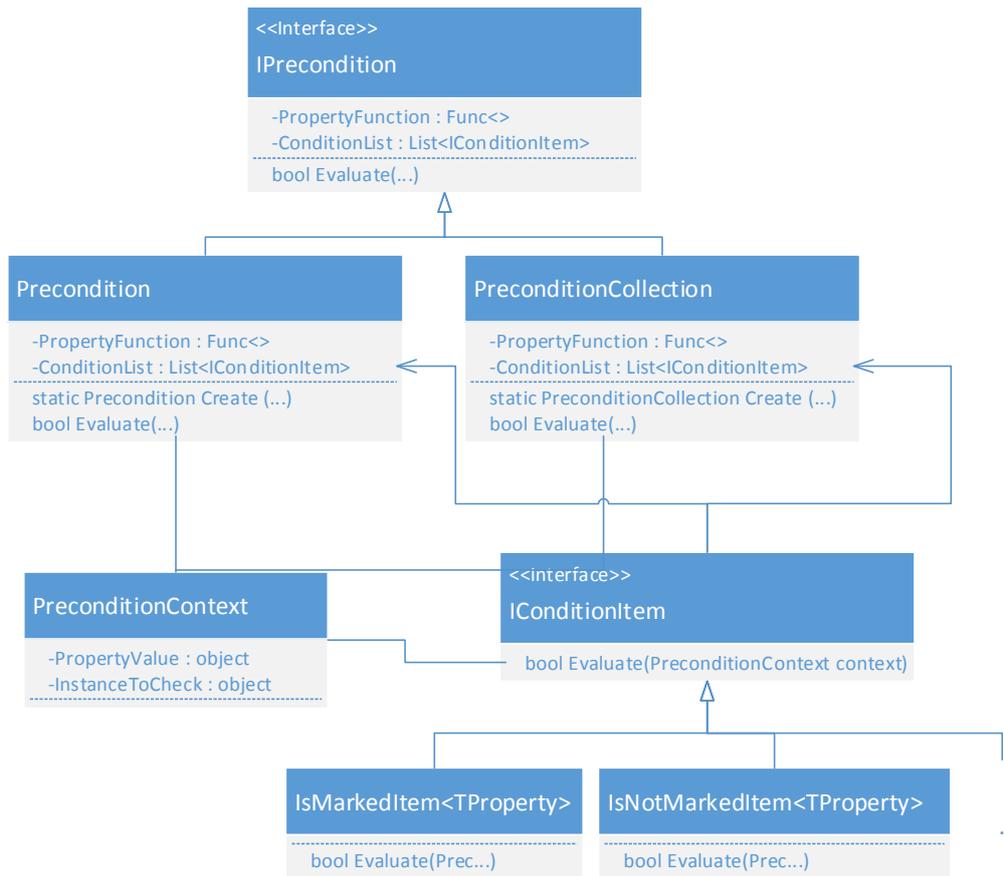


Figure 4.4.: Class Diagram of the Precondition Environment

4. The Process Rule Framework

elements which all need to be compared. An extra method for each of them would make the interface less clear.

The circumstance that it is named “*IsMarked*”, although it is capable of a generic *Equals*, is accounted for by usability. *IsMarked* resembles natural language very closely and specifies exactly what is intended. A more generic name like “Equals” might be interpreted ambiguously, especially when used in conjunction with properties that are not Markings. Also it prevents the framework to be used in the wrong way on a non-code level, although deliberate sabotage by a malevolent user is not prevented by it.

As this is a fluent interface, predicate expressions can be chained together to provide complex checks. For example, if a person needs to be between age 18 and 25 to qualify for, e.g., a scholarship, a possible precondition expression could be *PreconditionFor*(*person* => *person.Age*).*OlderThan*(18).*YoungerThan*(25). The predicates chained together have an implicit AND semantic. Chaining is useful to check complex objects (e.g., person) for different properties (e.g., age and height) with appropriate predicates. Another advantage of chaining predicates is to avoid defining a new predicate like “between”, since it can be expressed with already existent ones.

The predicate *IsMarked* and others as the fictive “OlderThan” and “YoungerThan” predicates are implemented using extension methods that reside in a single static class. This allows for a very easy extensibility, as any required functionality can be added without affecting anything else.

The expressions *PreconditionFor* and *IsMarked* create respective objects *Precondition* and *IsMarkedItem* in the back end providing the functionality upon evaluation. The objects created by the predicate expressions are called *IConditionItem* (because they have to implement an interface of the same name) which represent a predicate for the property specified in the *PreconditionFor* lambda expression argument. A *Precondition* object can have multiple *IConditionItems*.

In Listing 4.2, Line 1 shows the return type of the extension method, *IPreconditionBuilder*. It is an interface that enables the fluent declaration style. It is also used as an argument in Line 2, together with the *this* keyword. This is the special syntax for extension methods, which are static, enabling them to be used like normal methods using the dot operator. The *IPreconditionBuilder* argument is the same object that is returned, which allows the chaining of extension methods with the same properties.

```
1 public static IPreconditionBuilder<T, TProperty>
2   IsMarked<T, TProperty>(this IPreconditionBuilder<T, TProperty>
3     preconditionBuilder, TProperty marking)
4   {
5     return preconditionBuilder.Add(new IsMarkedItem<TProperty>(marking
6       ));
7   }
```

Listing 4.2: Implementation of *IsMarked*

In Line 4, the *IConditionItem* for the *IsMarked*-extension method is instantiated and added to the precondition via the *IPreconditionBuilder preconditionBuilder*. The instantiated *IConditionItem* stores the Marking in a property to be used when the *Precondition* is evaluated.

The *Precondition* itself is created in the back end by the *PreconditionFor* expression evaluation. The precondition obtains the lambda expression argument *PropertyFunction* as a property and is added the *IConditionItems* defined by the predicates. A full, operational precondition exists now on the front end.

When the rule is instantiated, the front end expressions create the necessary precondition objects in the back end (cf. Table 4.1). The objects created are the *Precondition* itself, the *PropertyFunction* and the predicates in form of *IConditionItems*. However, at the moment the rule is instantiated, the *PropertyFunction* and the *IConditionItems* are independent from each other. To be able to evaluate the whole precondition these parts need to be brought together. Evaluation happens when a target object is passed to the process rule.

To bring both parts together, a *PreconditionContext* is created with the target object and the *PropertyFunction* as arguments. The context extracts the property from the target object with the *PropertyFunction* and performs some basic validity checks. Each *IConditionItem* takes the context and evaluates its condition, and then passes the result to the *Precondition*. Using a context that performs the property extraction only once instead of executing it for each *IConditionItem* individually benefits performance and the encapsulation principle. The context allows the *IConditionItems*, where the actual evaluation takes place, to be completely agnostic to any specific details about the target object or the property, i.e., only the property type *TProperty* is known.

When an evaluation of the precondition is requested, the context is passed to all *IConditionItems*. They perform an internal evaluation and return true or false based on the result. The individual results are aggregated. If any *IConditionItem* returns false, the whole precondition returns false (AND-semantic).

4.2.5. Preconditions for Collections

The preconditions presented in Section 4.2.4 are only for properties that are not collections (cf. Section 3.7). Collections need to be handled slightly differently, using the *PreconditionForEach* expression. The apparent difference in the declaration - apart from the *Each*-suffix - is that the *PropertyFunction* requires a projection (*Select*, cf. Line 7 from Listing 4.1) for the desired property. In the *PreconditionForEach* expression at Line 7, the *PropertyFunction* navigates to the Marking property of the outgoing transitions of the step. The *Select* extension method is necessary because *OutgoingTransitions* is a collection of *MicroTransitions*. From each of the transitions from the *Marking* property is the desired property. The *Select* extension method of the *PropertyFunction* is used to obtain a collection *IEnumerable<TProperty>*.

4. The Process Rule Framework

Each element of type *TProperty* is treated the same as with a non-collection property. More specifically, for each element a context is created and passed to each of the *IConditionItems* (cf. Figure 4.4). The results are taken and aggregated to obtain a single true or false. A definitive advantage of this approach is that every extension method can be reused for collections.

4.2.6. Effects

Effects represent the changes a process rule applies to a process instance. For example, in Listing 4.1, Line 11, the Marking of the target object is altered to Confirmed. Technically, an effect is the assignment of a value to the property by means of lambda expressions. An overview in form of a class diagram of every class concerning effects can be found in Figure 4.5.

Declaration-wise effects are basically the same as preconditions. There is again a front end and back end, the associations are analogous to preconditions. The creation expression starts with *EffectFor*, which takes a lambda expression, again called *PropertyExpression* as an argument to select the property to have the effect for. What is assigned is determined by an extension method which takes the value to be assigned as an argument. Since the only effect currently required by the PHILharmonicFlows runtime is the assignment of new Markings, the only extension methods implemented is *AssignMarking*. However, using extension methods provides extensibility of effects if needed and keeps consistency with preconditions. It is also possible to chain assignments together, which will possibly be useful for future extensions or applications outside the scope of the process rules of PHILharmonicFlows (cf. Section 4.2.11).

Unfortunately, doing assignments with lambda expressions is complicated, especially if the arguments are dynamically determined during runtime and the assigning function must be assembled from parts. A process rule is applied to different target objects as the process execution progresses, so a function is needed to combine the assignment with the property selection and with a target object of appropriate type as a dynamic parameter. Considering the effect from Listing 4.1, Line 11, the resulting assigning function must provide the same functionality of the function $instance \Rightarrow instance.Marking = Confirmed$, with *step* being the dynamic parameter: At this point, the dynamic argument is called *instance* since such effects are not limited to steps, but the target objects supplied as the dynamic parameter can be *MicroTransitions* and *MicroStates*.

The Marking value Confirmed is provided by the *AssignMarking* extension method and known at compile time, same goes for the property expression that selects the Marking in the example. The final building block for the assigning function is a placeholder serving as the dynamic parameter. The "glue" that brings these three parts together and enables doing assignments in a LINQ-like style is *Expression.Assign*. It is a binary expression taking two expression arguments, named *left* and *right*, and assigns expression *right* to expression *left*. In terms of the assigning function, the property function and the placeholder need to be combined to form expression *left*, whereas expression *right* is

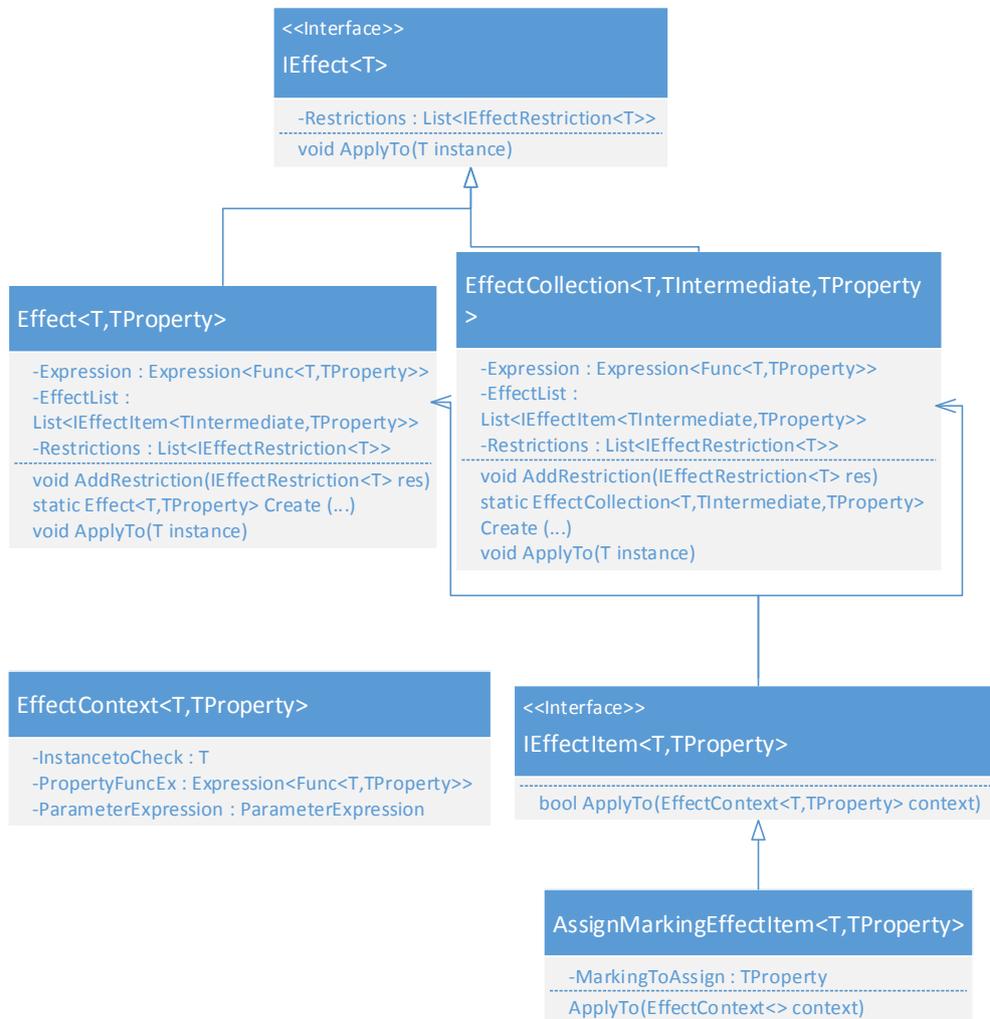


Figure 4.5.: Class Diagram of the Effect Environment

4. The Process Rule Framework

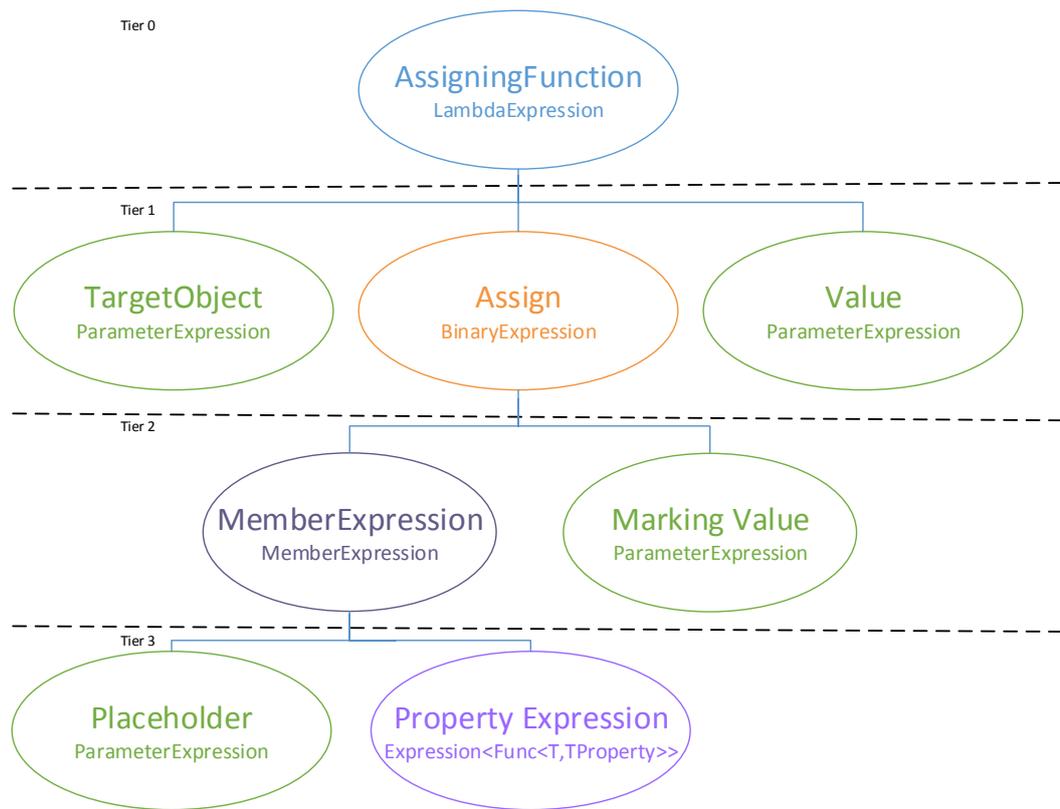


Figure 4.6.: Simplified Expression Tree of the target function

the Marking value. All expressions are represented as expression trees (cf. Section 3.12) and in the end form a single, large expression tree that represents the assigning function. The resulting expression tree is depicted in Figure 4.6 in a simplified version, the actual tree is far too complex to be displayed here entirely.

In the following, the assigning function is assembled from parts. This means the expression tree in Figure 4.6 must be constructed with these parts. The expression tree is conceptually constructed from bottom to top. The parts which are already present from the effect are *PropertyExpression* and the Marking value to assign.

The nodes in the tree contain an identifier for the expression and below, in a slightly smaller font, the expression type. At Tier 3, the expression tree has the two parameters for the *MemberExpression*. The *PropertyExpression* is obtained from the *EffectFor* declaration, the placeholder must be created manually. It is a *ParameterExpression* from which the constructor takes two arguments: First and most important, the type of the parameter, which is the generic *T*, and second a string naming the parameter. In code this may be expressed as `ParameterExpression p = Expression.Parameter(typeof(T), "Placeholder");`

Placeholder and PropertyExpression are then combined into a new lambda expression, which is then cast to a *MemberExpression* (cf. Section 3.6). The explicit cast to *MemberExpression* is necessary because *Assign* expects something on the left side that an assignment can be made to. Not all expression types have this attribute. Also this attribute cannot be inferred from an ordinary lambda expression. For the right side, it is advantageous to have a second, variable parameter to *Assign* to represent the value instead of hard-coding the constant value of the Marking. Therefore, the Marking value to assign is another *ParameterExpression*, with the difference that this time the type is *TProperty*. Type-wise, this works out because the *PropertyExpression* converts from type *T* to type *TProperty*, which means the result value of the *MemberExpression* is of type *TProperty*, too.

Now that both parameters for *Assign* are present and accounted for, a *BinaryExpression* making the assignment is created, which is located on Tier 1 of the expression tree in Figure 4.6. However, the binary expression can not be compiled into an executable function yet. Hence another conversion takes place. The parameter expressions were lost when creating the binary expression, so new *ParameterExpressions* replacing placeholder and the Marking value are needed (named “TargetObject” and “Value” respectively), to convert the assign expression into another lambda expression. The lambda expression can now be compiled to obtain the assigning function with two parameters for the instance and the value to be assigned.

4.2.7. Effects on Collections

Like the preconditions, effects also need a different approach to deal with collections. In Listing 4.1, Line 13 shows a declaration for an effect on collections. The first difference is the addition of the *Each*-suffix, so an *EffectFor* expression becomes an *EffectForEach* expression. The second is a new syntactical difference: The generic type parameters in angle brackets following *EffectForEach*. Their significance will be explained later on. After the angle brackets follows the property expression, the same as with a *PreconditionForEach* expression, with *Select* being the prominent element. The *AssignMarking* extension method is conceptually the same as the one with effects, as explained in Section 4.2.6.

The challenge for implementing effects on collections lies with the property expression. In the preconditions on collections it was sufficient to take the result of the expression (i.e., *IEnumerable<TProperty>*) and treat each element as a regular property. However, in order to keep syntax consistent, Effects on collections are not as easy to accomplish. In order for an assignment to be made by *Expression.Assign*, it needs a proper *MemberExpression* as a left parameter. Unfortunately, there is no straightforward way to achieve this with a collection of objects. The assignment must be made to the same property of each of these objects. This requires a separate *MemberExpression* for each property, with the same target object as initial argument. In Line 13 of Listing 4.1 the *PropertyFunction* navigates to a collection of properties, which are the Markings of the

4. The Process Rule Framework

outgoing *MicroTransitions*. The collection of properties cannot easily be converted to a “collection of *MemberExpressions*”. The reason is the extension method *Select* in the *PropertyFunction*. As an extension method, it interrupts the path of reference properties necessary for a valid *MemberExpression*.

The *Select* separates the *PropertyFunction* in two parts. The first part is called *collection identifier* and it navigates to the collection *OutgoingTransitions* and stands before *Select*. The second part is called *property identifier* and is the argument expression of *Select*, *trans => trans.Marking*, which navigates from a transition to a the property. To obtain a valid member expression that can be used for an assignment to one property, the *Select* must be removed and the collection identifier and property identifier need to be extracted. Luckily, the property expression is represented as an expression tree with *Select* as its root node, which makes it easy to extract the collection and property identifier from the tree. In terms of the example in Listing 4.1, Line 13, the collection identifier is *step => step.OutgoingTransitions*, which navigates from a target object called *step* to a collection of unknown type, and the property identifier is *trans => trans.Marking*, which selects the *Marking* property from an element *trans* of unknown type from the collection.

After the extraction, the property identifier *trans => trans.Marking* is already a valid *MemberExpression* and can be used for assignment. It is not unlike the *PropertyFunction* as in Section 4.2.6 and can be regarded as such. The extension method *AssignMarking* is the same as the extension method in Section 4.2.6. However, the type of input parameter *trans* and the type of the collection *OutgoingTransitions* is not known to the compiler. If the property identifier is to serve as a *MemberExpression*, the type of *trans* and the collection must be known. This unknown type, denoted *TIntermediate*, is in fact masked inside the original expression tree of the *PropertyFunction* of *EffectForEach* and cannot be automatically inferred by the type inferencer. In order to extract it from the expression tree manually, a lot of work would have to be done, possibly negatively impacting performance. To avoid that possibility, the responsibility of providing *TIntermediate* has been shifted to the rule creator, who can easily provide it by specifying the type parameters of *EffectForEach*. The full specification is actually *EffectForEach<TIntermediate, TProperty>*. As a side effect of having to provide *TIntermediate*, the type of *TProperty*, although already known, must be stated explicitly for technical reasons regarding generics. As a drawback, the declaration *EffectForEach* is less consistent with the other declarations, but it may be argued that it is warranted for not affecting performance of rule execution.

Now that all required information is available, the collection identifier is applied onto the target object of type *T* to obtain a collection of type *IEnumerable<TIntermediate>*. Each of its elements is treated like an instance of a normal *Effect*, with the property identifier being a proper *MemberExpression* for use with *Expression.Assign*.

4.2.8. Effects for Execution Rules

Execution Rules do not alter the micro process state directly as Marking- and Reaction Rules do. Instead, an event is raised informing the user interface that certain data is required or a commitment is needed. The necessary logic to raise an event is provided within the *AbstractExecutionRule<T>* class, which is limited to a single effect: *RaiseProcessRuleEvent(ProcessRuleType type)*, with *type* being an identifier for the Execution Rule (e.g., *UserCommitmentEr3*). Preconditions are specified in the same way as other rule types.

4.2.9. Restrictions on Effects

Since there are classes that are derived from other classes and rules can be for the base classes, it may be necessary to restrict single effects to one of the derived classes. In the rules for PHILharmonicFlows, this scenario is most common for *MicroStepBase* and the derived classes *MicroStep* and *ValueSpecificMicroStep*. A *ValueSpecificMicroStep* often has effects for its *ValueSteps*, which obviously do not apply to a normal *MicroStep* (c.f. .). To avoid defining an extra rule to deal with this case, *Restrictions* have been introduced. They can be considered additional preconditions specifically for a single effect. They are declared by using the *When* extension method that has a predicate expression as a parameter.

4.2.10. Building and Using a Process Rule

Each process rule has its own class, derived either from *AbstractExecutionRule<T>* if it is an Execution Rule or from *AbstractEffectRule<T>* if it is a Reaction- or Marking Rule. Usually a process rule class only contains a constructor without arguments in which the Preconditions and Effects are defined. However, the process rule class can be enhanced with additional custom functionality, in case special circumstances require it.

Since Preconditions and Effects are defined in the constructor, it is ensured that upon instantiation of a rule, their declaration functions are executed to create the necessary classes to provide concrete rule functionality. The abstract rules provide a method *ApplyTo(T instance)* that is used to execute a rule on an instance. For details on how these rules are employed in PHILharmonicFlows, please refer to Chapter 5.

4.2.11. Applications of the Process Rule Framework

The Process Rule Framework and the Process Rule Manager not only work with micro processes, but can also be used for process rules of macro processes and any other process rules within PHILharmonicFlows. Most of these rules do not differ much from the process rules for micro processes. The built-in extensibility possibilities should be

4. The Process Rule Framework

able to provide all additionally needed functionality, so no fundamental change in the Process Rule Framework should be necessary.

Although the Process Rule Framework currently is tailored to the specific needs of PHILharmonicFlows, it is possible to adapt it to applications outside the scope of object-aware process management systems. The underlying concept is sound and has proven to be viable in testing of the PHILharmonicFlows runtime. For a general-purpose rule framework, however, the concept would have to be extended and generalized. With proper adaptation, rules could even be created and executed at runtime, using the power and versatility of expression trees.

4.3. Summary

This chapter introduced the fundamental programming constructs that were used in the Process Rule Framework. The Process Rule Framework was designed to simplify the creation of process rules. The ultimate requirement was to make the runtime easily maintainable, which has been achieved. An analysis of the process rules revealed a pattern which could be used create a template for process rules with high abstraction level. Hereby, the Framework implementation hides the complexity of process rules. The Process Rule Framework enables the creation of rules with an easy and concise syntax at a very high abstraction level.

5

The Process Rule Manager

The Process Rule Manager is an object that manages the application of process rules to a given process instance. Upon a change in a process instance, the Process Rule Manager looks for appropriate rules, applies them if all preconditions are met, and in case of an Execution Rule, propagates the request for data or a commitment to the WCF Service and thereby the UI. While process execution is rule-driven, rules themselves rely on events to trigger them. This mechanism is explained in section 5.1. While process rules are conceptually independent from each other and do not require a certain ordering, it is unavoidable in practice to have certain implicit dependencies between rules. The process rule manager must account for these implicit dependencies by imposing that certain rules are applied in a certain order.

5.1. Event-driven Rule Application

Actively looking for instances on which a rule can be applied by iterating over all sets of the process instance has been found to be inefficient. For each element of the process graph, any rule of appropriate type must be checked for matching preconditions. Since most of the time changes in the process graph only happen in a small area (the currently activated state, with a few exceptions), most of the checks are unnecessary and do not yield new results. Tracking changes to the process graph to narrow down the point where a possible rule application is possible consumes additional memory and impacts performance. Additionally, while iterating over each element and applying rules, subsequent necessary rule applications might not trigger immediately, but are delayed in the worst case until the next pass. This is likely to have an impact on the responsiveness of the User Interface, since between two execution events normally as many as four or five Marking Rule applications occur. This is under the assumption that iterating over all sets takes a noticeable amount of time. So possibly the user has to wait half a second or more before another action can be taken. This can be worsened by putting the runtime under significant load.

A simple observation allows for a much more efficient execution: New process rules only need to be applied if there has been a change in the process instance beforehand. Take, for example, a Marking change by another rule, the user provided a data value or has made a commitment. Instead of actively searching for these changes, the process

5. The Process Rule Manager

instance can notify a system which then can specifically check rules and apply them without any unnecessary overhead. This system is called the *Process Rule Manager*. The notification is an event (cf. Section 3.10) raised by the elements of the process instance where the change occurred. To keep the pool of possible rules as small as possible, the instance needs to inform the Process Rule Manager about the kind of change. For example, Marking changes trigger Marking Rules, while a commitment triggers a Reaction Rule. Trying to apply Marking Rules to some event where a Reaction Rule is the appropriate response, is a waste of resources and must be avoided.

The event-driven paradigm solves the problem of delayed rule triggering, the problem of appropriate rule selection and the problem of rules triggering cascades. Once a rule is applied, the changes it applies causes the instance to raise new events that trigger the application of subsequent rules immediately. This creates a cascade of rule applications until stopped by an Execution Rule event. Continuing process execution requires interaction with the user. An in detail example of how event-driven rules are applied with Process Rule Manager is presented in section 5.2.

5.2. Process Rule Manager Context and Example

The Process Rule Manager is one of the essential parts of the PHILharmonicFlows runtime. It is responsible for advancing process state and is the interface between the process instances and the WCF Service. The relationship between a micro process instance, the Process Rule Manager and the WCF Service is depicted in Figure 5.1. The Process Rule Manager conceptually resides between the WCF Service and the process instance. It receives events from the instance, chooses appropriate rules and the rules alter the Markings of the process instance elements.

In the following, an exemplary execution of a micro process is conducted. It focuses on the qualitative aspects and therefore omits certain detail. The numbers in the circles in Figure 5.1 are referenced by a letter inside normal parentheses and indicate where a certain event happens. The numbering does not necessarily induce an ordering to the events. The target object is the process instance element that raised the event and to which an appropriate rule is applied. In the beginning, the micro process instance has not yet been started.

1. The micro process instance is started (A), which triggers an event for Reaction Rule RR01 (B) . The event is acknowledged by the Process Rule Manager, which selects the Reaction Rule from its rule repository and applies it to the target object. This updates the Markings (C) of several elements of the graph like states, which in turn trigger additional events (B).
2. These events raised are now Marking Rule events. Depending on the type of target object that raised the event, the Process Rule Manager looks for appropriate rules and applies them (C).

5.2. Process Rule Manager Context and Example

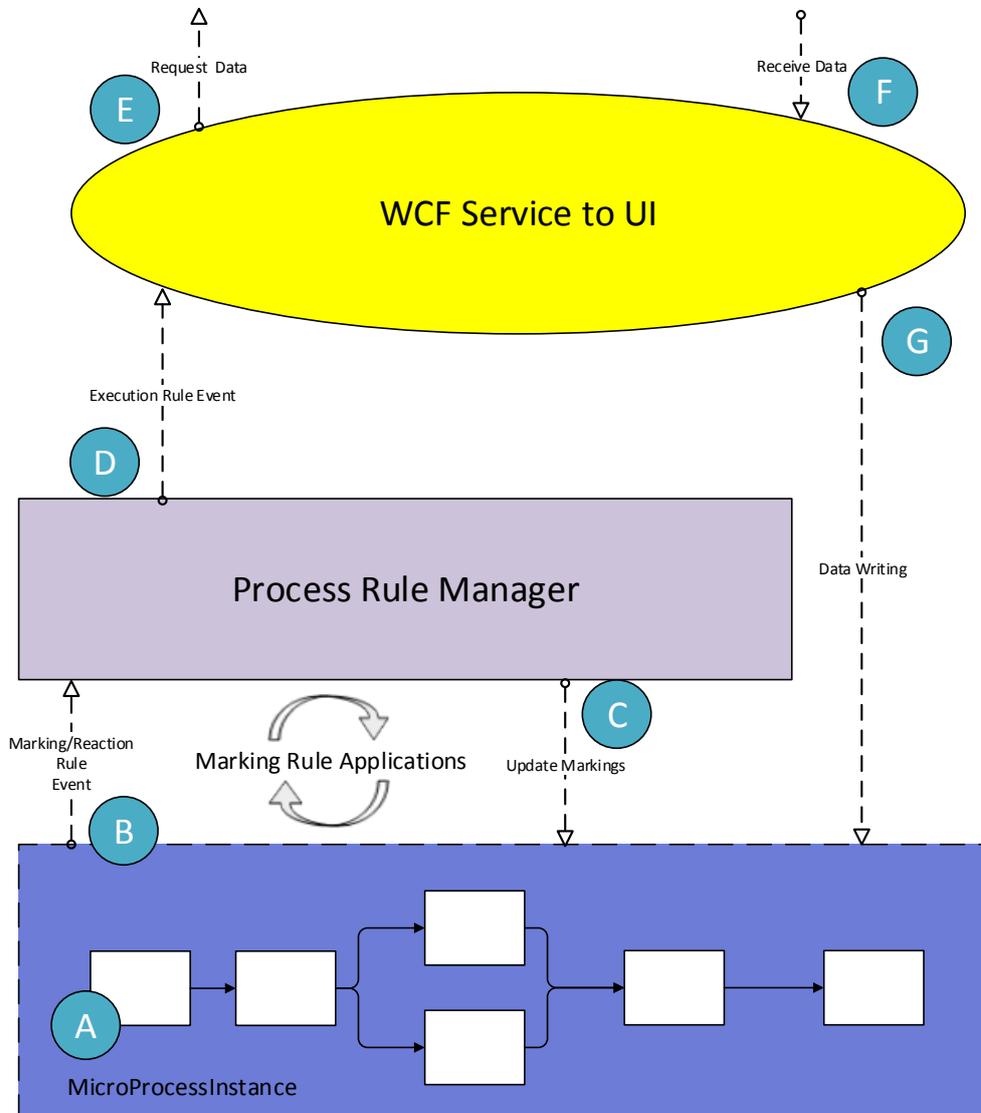


Figure 5.1.: Runtime Architecture and Template of Event-Driven Process Rule Application

5. The Process Rule Manager

3. The cycle between (B) and (C) represents a cascading application of rules and repeats until the preconditions for the raising of an Execution Rule event is met. An Execution Rule does not affect a process instance directly. It is instead passed to the WCF Service (D)
4. The WCF service analyzes the incoming event and extracts a reference to the target object to which the Execution Rule event belongs. The WCF Service checks if an attribute value has been provided for the instance by the user. It communicates with the UI to determine if a value is present. If not, it reacts by sending a request to the UI (E) to acquire the data. If data is already present, it continues at (G).
5. The UI waits until the user has filled out the corresponding form field. Once this has happened, the form field is parsed and the resulting value is sent back to the WCF Service (F)
6. The service matches the incoming data to the corresponding target object. Since it obtained a reference, it can write the data directly into the instance attribute (G), which in turn raises a Reaction Rule event (B)
7. The cycles repeat themselves until the process instance execution is terminated.

The direct write back to the model is currently needed to trigger the Reaction Rule event. This may change as soon as the WCF Service is specified entirely. The current status of the WCF Service is *not implemented* and only rudimentary components have been devised for the testing purposes of other parts, for example the Process Rule Manager.

The Process Rule Manager requires that process instances register themselves with the Process Rule Manager. Otherwise the event mechanism which drives rules execution is not able to work. Ideally, registering takes place directly after deployment so a process instance can be executed right away. However, it is possible to register or unregister process instances at any point in time, which might be useful in the future, for example when performing a schema evolution [3] for running instances.

While an event-driven rule execution has many advantages for performance and simplicity, it also has disadvantages. The most significant disadvantage is, that event raising logic and -conditions have to be hard coded into the runtime model, which makes changing events or even adding new ones significantly harder. Therefore runtime model maintainability suffers.

5.3. Process Rule Manager Implementation

The Process Rule Manager is responsible for finding all appropriate rules for a given target object to test whether they are applicable or not. This consists of two separate requirements. First, if a Reaction Rule event is raised, the appropriate Reaction Rule must be found and applied to the target object. Second, if the event is a Marking Rule event, all Marking Rules for the appropriate type must be found. The standard case

is to find a rule *AbstractEffectRule<T>* or *AbstractExecutionRule<T>* for a target object of type *T*. But if *T* is a subtype of *U* (for example, *MicroStep* and *MicroStepBase*), an *AbstractEffectRule<U>* might also be applicable. A more detailed problem statement and the solution to the problem is presented in Sections 5.3.3 and 5.3.4.

5.3.1. Reaction Rule Mapper

Each Reaction Rule event has a parameter that identifies the Reaction Rule it is supposed to trigger, the so called *reaction rule type*. To apply the Reaction Rule to the target object that raised the event, the Process Rule Manager needs to retrieve the corresponding rule from a collection of Reaction Rules. The Reaction Rule Mapper associates reaction rule types with their corresponding Reaction Rule of type *T*. To retrieve a Reaction Rule by its corresponding reaction rule type, the reactions rules need to be stored in a key-value pair with the reaction rule type as the key. This is done by a *Dictionary<ProcessRuleType, object>*. Dictionary is a generic collection therefore can only hold objects of the same type. A Reaction Rule however is parametrized by a type parameter *T*, which changes depending on the type of target object the Reaction Rule is for. This prevents an *AbstractEffectRule<T>* to be stored in the same generic collection as an *AbstractEffectRule<U>*, $T \neq U$ since they are considered different types by the CLR. In order to store every Reaction Rule in the dictionary as a value, the value must have the a common non generic super class of *AbstractEffectRule<T>*, which is *object*. Storing an *AbstractEffectRule<T>* as object hides its actual type information. When a rule is retrieved from the Reaction Rule Mapper, a cast to the appropriate type of the rule is necessary. In fact, a retrieval from the Reaction Rule Mapper is unnecessary, as the Reaction Rule Mapper can also handles the ensuing application of the Reaction Rule, after the rule has been retrieved from the dictionary and cast to its original type.

5.3.2. Process Rule Repository

The key-value mechanic of the Reaction Rule Mapper in principle also applies to the Marking- and Execution Rules. For an target object of type *T* several process rules for *T* might be applicable, so it would be advantageous if it were possible to retrieve this list based on the type as the key, again using a *Dictionary* to provide the basic functionality. A *Type* is, like everything else, an object in C# and can be used as key to a dictionary. The problem is again that generic collections can not hold objects of different types, but for one bucket of the dictionary, execution and Marking Rules have the common ancestor *AbstractRule<T>*. And since there is possibly more than one rule for a given type *T*, a *List<AbstractRule<T>>* is stored in a bucket, hidden under a non-generic interface *IEnumerable*. Like with the Reaction Rule Mapper, the Process Rule Manager delegates the task of applying rules to the Process Rule Repository. The mechanism for applying the rules is as follows. An target object of type *T* is passed to the Process Rule Repository and based on *T*, an *IEnumerable* is retrieved from the internal dictionary. Since the target object is of type *T*, the *IEnumerable* retrieved from the corresponding

5. The Process Rule Manager

bucket must be a *List<AbstractRule<T>>*. The *IEnumerable* can now be cast and each rule in the list is applied to the target object.

5.3.3. Handling a Process Rule Event

It was shown in section 5.3.1 and 5.3.2 how a strongly typed target object of a type is used to search for applicable rules and apply them. However, beforehand the Process Rule Manager needs to solve the following two problems:

1. A process rule event passes its source, the target object, not as an object of type *T*, but as an object of type *object*. This was made to be able to use the same event for all target object types, e.g. *MicroStep*, *MicroTransition* and so on. However, once the target object reaches the Process Rule Manager, it needs to be cast back from *object* to its original type *T* to be compatible to the Reaction Rule Mapper and the Process Rule Repository, which rely on knowing *T*.
2. The step types (*MicroStep*, *ValueStep* and *ValueSpecificMicroStep*) are derived from the abstract classes *MicroStepBase* and *CommonStepBase* (see Figure 4.1). Also, there are rules that have an abstract type as its target type. The problem is that generics forbid to apply a *Rule<U>* to an target object of type *T* when *T* is derived from *U*. Also, casting a *Rule<U>* to a *Rule<T>* is not possible, the only remaining solution is to cast an target object of type *T* to all of its superclasses (e.g., *U*) and for each try to apply rules.

For problem a), there are no information from the event about which actual type the sender object has. Fortunately, the pool of possible solutions is finite and also small. It can be tested in a trial-and-error fashion. The possible types an target object can be cast to consist of the following seven types: *MicroProcess*, *MicroState*, *MicroTransition*, *MicroBackwardTransition*, *ValueSpecificMicrostep*, *MicroStep*, and *ValueStep*. The trial and error is simply casting the target object to one of these types, if it does not fail with an exception, the target object is of type *T* and can be passed to Reaction Rule Mapper or Process Rule Repository, depending on the event type. Afterward it terminates execution, since an target object can only be of one of the types.

Basically the same functionality as problem a) is needed with problem b), however an target object can belong to several types, namely the target object's actual type and the supertypes. Therefore, if a cast was successful, it must not terminate but instead continue until all types are tested.

Combining both solution requirements, a pattern was created that resembles that of a switch statement with fall-through behavior. The switch parameter is the target object and the cases are all possible types. A normal switch statement of C# however is unable to switch on types. For this reason, a custom switch on types with the required features and behavior has been designed (cf. Section 5.3.4).

5.3.4. Type Switch

The *TypeSwitch* class allows to execute code based on the type of an object. It uses a fluent style syntax and safe type casting to enable the core functionality of the Process Rule Manager. It mimics syntax and behavior of a standard switch statement in C#. The *TypeSwitch* can be configured to either fall through the cases like a normal switch without break, or simulate the break statement by aborting the function chain after a *Case* method was successful.

Listing 8 shows an excerpt from *ProcessRuleManager.cs* that uses a *TypeSwitch* to match target objects to process rules. If a Marking Rule event is triggered, a new *TypeSwitch* is instantiated with a constructor that takes the object on which the switch will be performed as an argument. Next, an arbitrary number of methods with signature *Case<TPossibly>(Action<TPossibly> action)* follow. The type parameter *TPossibly* represents a type to which the object may be cast. For solving problem one and two, the type switch remains in its default fall-through behavior.

Inside the *Case*-method, the object is safely cast to *TPossibly* using the operator *as*. Instead of throwing an exception when a cast fails, the operator returns *null*. If the object is not *null* after casting, it was of type *TPossibly* and the *Action* is executed (for *Action* c.f. Section 3.9). The action is a lambda expression and acts as the case statement body. In Line 7 of Listing 5.1, the action takes the object, now cast to type *MicroProcessInstance*, and passes it as an argument to the *ApplyRulesTo* method of the Process Rule Repository.

Whether or not a cast is successful and the action executes, the fall-through behavior does not break the chain but takes the next *Case* and evaluates it. This continues until the last *Case* in the method chain, where either a *Default* method can execute additional code or the *Default* is missing and the type switch simply terminates.

5. The Process Rule Manager

```
1 switch (processRuleType)
2 {
3     case ProcessRuleType.MarkingRuleEvent:
4         new TypeSwitch(sender)
5             .Case<MicroProcessInstance>(process =>
6                 {
7                     _processRuleRepository.ApplyRulesTo(process);
8                 })
9             .Case<MicroStateInstance>(state =>
10                {
11                    _processRuleRepository.ApplyRulesTo(state);
12                })
13            .Case<MicroStepInstance>(step =>
14                {
15                    _processRuleRepository.ApplyRulesTo(step);
16                })
17            .Case<ValueSpecificMicroStepInstance>(vsstep =>
18                {
19                    _processRuleRepository.ApplyRulesTo(vsstep);
20                })
21            .Case<ValueStepInstance>(vstep =>
22                {
23                    _processRuleRepository.ApplyRulesTo(vstep);
24                })
25            .Case<CommonBase>(cbase =>
26                {
27                    _processRuleRepository.ApplyRulesTo(cbase);
28                })
29            .Case<MicroStepBaseInstance>(stepBase =>
30                {
31                    _processRuleRepository.ApplyRulesTo(stepBase);
32                })
33            .Case<MicroTransitionInstance>(trans =>
34                {
35                    _processRuleRepository.ApplyRulesTo(trans);
36                })
37            .Case<MicroBackwardTransitionInstance>(btrans =>
38                {
39                    _processRuleRepository.ApplyRulesTo(btrans);
40                });
41        break; // belong to switch
42    case ...
43        //continue with reaction rule types
```

Listing 5.1: *TypeSwitch* embedded in a normal switch

5.4. Initialization

When a process manager is instantiated, it creates its Process Rule Repository and Reaction Rule Mapper and fills them with all currently configured process rules. The rule creator currently has to ensure manually that a process rule is added to the initialization routine of the Process Rule Manager. To aid with this responsibility, unit tests have been designed to indicate that something may have been forgotten. Also removing a rule must currently be managed by the programmer. Due to the expected low number of new or obsolete rules this inconvenience has been considered acceptable.

5.5. Summary

In summary, this section showed how the Process Rule Manager fits into the layout of the PHILharmonicFlows Runtime. The concept of event-driven rule application has been explained and illustrated with an exemplary process execution. Further, the internal workings of the Process Rule Manager were demonstrated. In the context of discussing the Process Rule Manager, a few requirements have been hinted at that other components need to meet in order to work with the Process Rule Manager.

6

Related Work

This section discusses other approaches for defining rules. It is not limited to rules used for executing business processes, but rules in general, for example business rules. Furthermore, an algorithm frequently used in business rule engines is discussed.

6.1. QuestionSys

QuestionSys [46, 45, 47] is a system for designing and evaluating electronic questionnaires. The questionnaires themselves are realized as an activity-centric business process. It also comprises a system called *QuestionRule* to create boolean expressions. These expressions are called rules and are used to evaluate the questionnaire. Rules can aggregate information from the questionnaire and summarize the result as string, e.g., “The participant is a heavy drug user” if the participant admits to using cocaine or other illegal drugs regularly.

The rules comprise simple comparisons of process variables with constants, but can also include functions like counting values of process variables. For example, such a function can be used to determine if a participant has answered at least three questions. Unlike process rules in PHILharmonicFlows, rules can also be built with a graphical rule editor. The editor includes all basic comparison operators and functions and can be extended with custom function definitions. The resulting rule expression is parsed and converted into executable code. A detailed description of QuestionRule can be found in [25].

Compared to the Process Rule Framework, QuestionRule lacks the ability to define rule effects other than returning a string which describes the result. Rules cannot change the state of the questionnaire process. However, this is not a requirement of QuestionSys.

6.2. Rete Algorithm

The Rete algorithm [11] was developed by Dr. Charles Forgy and is widely used in business rule engines [33, 14]. The algorithm solves the many pattern/many objects matching problem efficiently. It finds all objects that match each pattern.

6. Related Work

A naive approach for such a task is to check a rule against all facts in a knowledge base, applying the rule if all conditions are satisfied, then moving on to the next. For moderate to large-sized rule and knowledge bases, this approach performs far too slowly. The Rete algorithm (Latin *rete* 'net', 'network') constructs a network of nodes, where each node (except the root node) corresponds to a pattern occurring in the preconditions of a rule. A path from root node to a leaf node means that all preconditions of the rule represented by the leaf node match. Therefore, the rule can be applied. Data is organized as tuples, which are called facts in the rete algorithm. The efficiency is based on the following properties:

- The network reduces or eliminates certain types of redundancy. If preconditions have the same patterns, they share the same node representing the pattern.
- The nodes memorize fact matches when evaluating rule preconditions. This allows avoiding complete re-evaluations of all facts each time changes are made to the knowledge base. Instead, only the changes (deltas) are evaluated.

The current fact and pattern base (i.e. target objects and process rules) of the process rules of PHILharmonicFlows is sufficiently small to make the speed gains by implementing the Rete algorithm negligible. This is supported by the running time of the process rule tests. As of the time of writing this thesis, there are seven elements of the micro process graph as facts, and nine different *IConditionItems* serving as patterns. The implementation of the process rules for macro processes will increase these numbers, but not significantly. The architecture of the Process Rule Manager and the definitions of the process rules themselves support a future implementation of the Rete algorithm in the PHILharmonicFlows runtime, should the need arise.

6.3. Microsoft BizTalk Server Business Rule Engine

Microsoft offers a business rule engine as part of the *BizTalk Server* [33] software. Business Rules are comparable to PHILharmonicFlows' process rules and consist of *conditions* and *actions*, which are similar to preconditions and effects. As a commercial product, the rules are capable to process data from XML, databases or .NET classes to be used with conditions. Actions are also able to write to the data sources, e.g. adding data from a XML file to a database table. Rules are built using a graphical editor which contains all possible condition and effect options, no programming is necessary.

Since BizTalk Server is closed source, implementation details of the rules cannot be discussed. However, it is known the business rule engine uses the Rete algorithm (cf. Section 6.2) for executing its business rules [32].

7

Summary and Outlook

7.1. Summary

This thesis aimed at providing an easy-to-use and maintainable way to create process rules. With the Process Rule Framework, presented in Chapter 4.2, this has been achieved. The simple template of a process rule with preconditions and effects and the high-level, fluent style definition syntax allow a rapid development pace. Additionally, the Process Rule Framework has the possibility to be extended to a full-fledged business rule engine that allow designing, compiling and applying rules entirely at runtime. The capabilities of the Process Rules Framework have been verified with the implementation of the process rules.

To enable the complex interactions and cascading effects between process rules, the Process Rule Manager (cf. Chapter 5) was designed. It matches incoming events to appropriate rules and passes data requests to the User Interface. An exemplary run from the perspective of the Process Rule Manager has been presented in Section 5.2. While macro processes are currently not implemented, the Process Rule Manager was designed to handle these process rules as well.

One of the main goals for the Process Rule Manager was to keep process rules independent from each other, which increases the maintainability. However, during the tests of the Process Rule Manager, it was observed that some of the process rules desired execution behavior depended on the definition of other process rules. These dependencies are not intended by the PHILharmonicFlows Framework to enable the cascading behavior, but implicit dependencies that stem from the practical implementations and the inherent complexity of process rule interactions. As these dependencies are implicit, they are not easy to find and resolve. This has a heavy impact on maintainability and can only be mitigated by careful testing.

7.2. Outlook

The immediate next steps in the development of the PHILharmonicFlows runtime is the extensive testing of the Process Rule Manager and the implemented process rules. A functional micro process execution is a prerequisite for the macro process implementation

7. Summary and Outlook

and the integration of the PHILharmonicFlows permission and authorization system. As soon as micro process execution is deemed stable, macro processes will be implemented, allowing to realize most of the potential of the PHILharmonicFlows framework. In parallel, the concept of the permission and authorization system will be finalized and subsequently implemented.

To extend the theoretical base of the PHILharmonicFlows framework, the following research topics of object-aware process management are currently under consideration or actively researched:

Schema evolution describes the adaptation of processes or databases to changing circumstances. While schema evolution has been researched and applied for activity-centric process management systems [10, 44], a concept is still being researched for object-aware processes [6]. Since process schemas and data schemas are tightly integrated in object-aware processes, schema evolution in PHILharmonicFlows presents a particular challenge. The reason is that evolving data schemas affects corresponding process schemas and vice versa. Also, a migration to an evolved process schema for running process instances is desired.

Process variants describe processes in different contexts or environments. For example, an application for a job in the the human resource department and a job in the engineering department is in large parts the same (e.g. Name, address, salary). However, the professional qualification requirements are different for each department. To accommodate for the necessities of each department, process variants are needed. For the activity-centric modeling and execution paradigm, process variability is well understood [1]. For object-aware processes, however, this is not yet the case. In object-aware processes, variants of objects exists with different attributes and therefore different micro processes. It is also possible to have different micro processes for one and the same object. Research into object-aware process variability is planned for the future.

For most data storage needs it is sufficient to only store the currently valid data. In case of a change, data which is no longer valid will simply be overwritten. However, it can be necessary to keep track of these changes and to document the development of data (e.g. prices or employment). To this end, PHILharmonicFlows needs to support historization. Historization is therefore a possible future research topic in context of object-aware processes.

Bibliography

- [1] Ayora, C., Torres, V., Weber, B., Reichert, M., Pelechano, V.: VIVACE: A Framework for the Systematic Evaluation of Variability Support in Process-Aware Information Systems. *Information and Software Technology* 57, 248–276 (2015)
- [2] Bhattacharya, K., Gerede, C.E., Hull, R., Liu, R., Su, J.: Towards Formal Analysis of Artifact-Centric Business Process Models. In: *Business Process Management, Lecture Notes in Computer Science*, vol. 4714, pp. 288–304. Springer (2007)
- [3] Chiao, C.M., Künzle, V., Andrews, K., Reichert, M.: A Tool for Supporting Object-Aware Processes. In: *Proceedings of the 18th IEEE International Enterprise Distributed Object Computing Conference*. IEEE Conference Press (2014)
- [4] Chiao, C.M., Künzle, V., Reichert, M.: Enhancing the Case Handling Paradigm to Support Object-aware Processes. In: *Proceedings of the 3rd International Symposium on Data-Driven Process Discovery and Analysis (SIMPDA'13)*. pp. 89–103. CEUR Workshop Proceedings, CEUR-WS.org (2013)
- [5] Chiao, C.M., Künzle, V., Reichert, M.: Integrated Modeling of Process- and Data-Centric Software Systems with PHILharmonicFlows. In: *Proceedings of the 1st IEEE International Workshop on Communicating Business Process and Software Models, Workshop held in conjunction with the 29th International Conference on Software Maintenance*. pp. 1–10. IEEE Computer Society Press (2013)
- [6] Chiao, C.M., Künzle, V., Reichert, M.: Towards Schema Evolution in Object-aware Process Management Systems. In: *Proceedings of the International Workshop on the Evolution of Information Systems and their Design Methods (EMISA 2014)*. pp. 101–115. *Lecture Notes in Informatics (LNI)*, Koellen-Verlag (2014)
- [7] Chinnici, R., Moreau, J.J., Ryman, A., Weerawarana, S.: *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language* (2007), <http://www.w3.org/TR/wsdl20/>, last accessed on 2015/01/26
- [8] Church, A.: A Set of Postulates for the Foundation of Logic. *Annals of mathematics* pp. 346–366 (1932)
- [9] Cohn, D., Hull, R.: Business Artifacts: A data-centric Approach to Modeling Business Operations and Processes. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 32(3), 3–9 (2009)
- [10] Dadam, P., Reichert, M., Rinderle, S., Jurisch, M., Acker, H., Göser, K., Kreher, U., Lauer, M.: Towards Truly Flexible and Adaptive Process-Aware Information Systems. In: *Proceedings of the UNISCON 2008*. pp. 72–83. LNBIP, Springer (2008)
- [11] Forgy, C.L.: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial intelligence* 19(1), 17–37 (1982)

Bibliography

- [12] Fowler, M.: *FluentInterface* (2005), <http://www.martinfowler.com/bliki/FluentInterface.html>, last accessed on 2015/01/26
- [13] Hung, R.Y.Y.: *Business Process Management as Competitive Advantage: A Review and Empirical Study*. *Total Quality Management & Business Excellence* 17(1), 21–40 (2006)
- [14] JBoss Inc.: *JBoss Drools Documentation 3.2 Rete Algorithm* (2015), <http://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch03.html>, last accessed on 2015-02-09
- [15] Kreher, U.: *Konzepte, Architektur und Implementierung adaptiver Prozessmanagementsysteme*. Ph.D. thesis, Ulm University (2014)
- [16] Künzle, V.: *Towards a Framework for Object-aware Process Management*. In: *Proceedings of the 1st International Symposium on Data-driven Process Discovery and Analysis (SIMPDA'11)* (2011)
- [17] Künzle, V.: *Object-aware Process Management*. Ph.D. thesis, Ulm University (2013)
- [18] Künzle, V., Reichert, M.: *Integrating Users in Object-aware Process Management Systems: Issues and Challenges*. In: *Proceedings of the BPM'09 Workshops, 5th International Workshop on Business Process Design (BPD'09)*. pp. 29–41. LNBI, Springer (2009)
- [19] Künzle, V., Reichert, M.: *A Modeling Paradigm for Integrating Processes and Data at the Micro Level*. In: *Proceedings of the 12th International Working Conference on Business Process Modeling, Development and Support (BPMDS'11)*. pp. 201–215. LNBI, Springer (2011)
- [20] Künzle, V., Reichert, M.: *PHILharmonicFlows: Towards a Framework for Object-aware Process Management*. *Journal of Software Maintenance and Evolution: Research and Practice* 23(4), 205–244 (2011)
- [21] Künzle, V., Reichert, M.: *Striving for Object-aware Process Support: How Existing Approaches Fit Together*. In: *Proceedings of the 1st International Symposium on Data-driven Process Discovery and Analysis (SIMPDA'11)*. pp. 169–188 (2011)
- [22] Künzle, V., Weber, B., Reichert, M.: *Object-aware Business Processes: Fundamental Requirements and their Support in Existing Approaches*. *International Journal of Information System Modeling and Design (IJISMD)* 2(2), 19–46 (2011)
- [23] Landwerth, I.: *.NET Framework Blog: .NET Framework Blog* (2014), last accessed on 2015/02/08
- [24] Lanz, A., Kreher, U., Reichert, M., Dadam, P.: *Enabling Process Support for Advanced Applications with the AristaFlow BPM Suite*. In: *Proceedings of the Business Process Management 2010 Demonstration Track. CEUR Workshop Proceedings* (2010)

- [25] Mertes, B.: Concept and Implementation of a Rule Component Enabling Automatic Analysis of Process-Aware Questionnaires. Ph.D. thesis, Ulm University (2014)
- [26] Microsoft Developer Network: .NET Framework 4.5 and 4.6 Preview, <http://msdn.microsoft.com/library/vstudio/w0x726c2>, last accessed on 2015/01/26
- [27] Microsoft Developer Network: Using IntelliSense, [http://msdn.microsoft.com/en-us/library/hcw1s69b\(vs.71\).aspx](http://msdn.microsoft.com/en-us/library/hcw1s69b(vs.71).aspx), last accessed on 2015/01/26
- [28] Microsoft Developer Network: Visual C#, <http://msdn.microsoft.com/en-us/library/kx37x362.aspx>, last accessed on 2015/01/26
- [29] Microsoft Developer Network: What is XAML?, <http://msdn.microsoft.com/en-us/library/cc295302.aspx>, last accessed on 2015/01/26
- [30] Microsoft Developer Network: Windows Communication Foundation, [http://msdn.microsoft.com/en-us/library/dd456779\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/dd456779(v=vs.110).aspx), last accessed on 2015/01/26
- [31] Microsoft Developer Network: Windows Presentation Foundation, [http://msdn.microsoft.com/en-us/library/ms754130\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/ms754130(v=vs.110).aspx), last accessed on 2015/01/26
- [32] Microsoft Developer Network: BizTalk Server Business Rule Engine: Survival Guide (2012), <http://social.technet.microsoft.com/wiki/contents/articles/6480.biztalk-server-business-rule-engine-survival-guide.aspx>, last accessed on 2015/01/26
- [33] Microsoft Developer Network: Microsoft BizTalk Server (2014), [http://msdn.microsoft.com/en-us/library/dd547397\(v=bts.10\).aspx](http://msdn.microsoft.com/en-us/library/dd547397(v=bts.10).aspx), last accessed on 2015/01/26
- [34] Müller, D., Reichert, M., Herbst, J.: Data-driven Modeling and Coordination of Large Process Structures. In: Proceedings of the 15th International Conference on Cooperative Information Systems (CooplS'07). pp. 131–149. LNCS, Springer (2007)
- [35] Müller, D., Reichert, M., Herbst, J.: 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). pp. 48–63. LNCS, Springer (2008)
- [36] OMG: Business Process Model and Notation Version 2.0 (2011/01), <http://www.omg.org/spec/BPMN/2.0>
- [37] Pesic, M., Schonenberg, H., van der Aalst, W. M. P.: Declare: Full Support for Loosely-structured Processes. In: Proceedings of the 11th IEEE International Conference on Enterprise Distributed Object Computing 2007(EDOC 2007) (2007)

Bibliography

- [38] Rademakers, T.: *Activiti in Action: Executable Business Processes in BPMN 2.0*. Manning Publications Co (2012)
- [39] Reichert, M.: *Process and Data: Two Sides of the Same Coin*. In: *Proceedings of the 20th International Conf on Cooperative Information Systems (CoopIS'12), OTM 2012, Part I*. pp. 2–19. LNCS, Springer (2012)
- [40] Reichert, M., Dadam, P.: *ADEPTflex - Supporting Dynamic Changes of Workflows Without Losing Control*. *Journal of Intelligent Information Systems* 10(2), 93–129 (1998)
- [41] Reichert, M., Weber, B.: *Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies*. Springer (2012)
- [42] Reichert, M., Weber, B.: *Process Modeling and Flexibility-by-Design*. In: *Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies*, pp. 59–88. Springer (2012)
- [43] Reijers, H.A., Rigter, J., van der Aalst, W. M. P.: *The Case Handling Case*. *International Journal of Cooperative Information Systems* 12(03), 365–391 (2003)
- [44] Rinderle, S., Reichert, M., Dadam, P.: *Flexible Support of Team Processes by Adaptive Workflow Systems*. *Distributed and Parallel Databases* 16(1), 91–116 (2004)
- [45] Schobel, J., Ruf-Leuschner, M., Pryss, R., Reichert, M., Schickler, M., Schauer, M., Weierstall, R., Isele, D., Nandi, C., Elbert, T.: *A Generic Questionnaire Framework Supporting Psychological Studies with Smartphone Technologies*. In: *Proceedings of the XIII Congress of European Society of Traumatic Stress Studies (ESTSS) Conference*. p. 69 (2013)
- [46] Schobel, J., Schickler, M., Pryss, R., Maier, F., Reichert, M.: *Towards Process-Driven Mobile Data Collection Applications: Requirements, Challenges, Lessons Learned*. In: *Proceedings of the 10th International Conference on Web Information Systems and Technologies (WEBIST 2014): Special Session on Business Apps*. pp. 371–382 (2014)
- [47] Schobel, J., Schickler, M., Pryss, R., Reichert, M.: *Process-Driven Data Collection with Smart Mobile Devices*. In: *Web Information Systems and Technologies WEBIST 2014, Revised Selected Papers* (in press)
- [48] van der Aalst, W. M. P., Aldred, L., Dumas, M., ter Hofstede, A. H. M.: *Design and Implementation of the YAWL System*. In: *Advanced Information Systems Engineering, Lecture Notes in Computer Science*, vol. 3084, pp. 142–159. Springer (2004)
- [49] van der Aalst, W. M. P., Pesic, M.: *DecSerFlow: Towards a Truly Declarative Service Flow Language*. In: *Web Services and Formal Methods, Lecture Notes in Computer Science*, vol. 4184, pp. 1–23. Springer (2006)

- [50] van der Aalst, W. M. P., ter Hofstede, A. H. M.: YAWL: Yet Another Workflow Language. *Information Systems* 30(4), 245–275 (2005)
- [51] van der Aalst, W. M. P., Weske, M., Grünbauer, D.: Case Handling: A New Paradigm for Business Process Support. *Data & Knowledge Engineering* 53(2), 129–162 (2005)
- [52] Weber, B., Sadiq, S., Reichert, M.: 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)
- [53] Zugal, S., Soffer, P., Haisjackl, C., Pinggera, J., Reichert, M., Weber, B.: Investigating Expressiveness and Understandability of Hierarchy in Declarative Business Process Models. *Software & Systems Modeling* (2014)

A

Source

```
1 public bool ApplyTo(EffectContext<T, TProperty> context)
2 {
3     //The initial function that selects the property
4     Expression<Func<T, TProperty>> expression = context.
        PropertyFuncExpression;
5
6
7     //initial parameter, the object that has the property
8     ParameterExpression p = context.ParameterExpression;
9
10    //combine Function to determine the property and the Parameter
11    LambdaExpression lambda = Expression.Lambda(Expression.Convert(p,
        typeof (T)), p);
12
13    //Substitute parameters for a much nicer looking expression
14    Expression injected = new SwapVisitor(expression.Parameters[0],
        lambda.Body).Visit(expression.Body);
15
16    //convert expression "injected" to a new lambda function that gets
17    //the property
18    Expression<Func<T, TProperty>> combined = Expression.Lambda<Func<T,
        TProperty>>(injected, lambda.Parameters);
19
20    //Extract information about the property
21    var memberExpression2 = combined.Body as MemberExpression;
22
23    //Create parameter expression for the value that will be assigned
24    ParameterExpression valueExp = Expression.Parameter(typeof (
        TProperty), "value");
25
26    //Make the assignment expression
27    BinaryExpression body = Expression.Assign(memberExpression2,
        valueExp);
28
29    //collect parameters in a list
30    var parameters = new List<ParameterExpression> {p, valueExp};
```

A. Source

```
31 //convert the expression and parameters to a final function
    //expression that actually does the job on compilation
32 Expression<Action<T, TProperty>> testfunc = Expression.Lambda<
    Action<T, TProperty>>(body, parameters);
33
34 //compile the function expression so that things can be done
35 Action<T, TProperty> assignmentFunction = testfunc.Compile();
36
37 //finally execute the function
38 assignmentFunction(context.InstanceToCheck, MarkingToAssign);
39
40 return true;
41 }
```

Listing A.1: Expression Trees in the AssignMarking function

Glossary

Activity-centric processes: Are defined as a set of activities of variable granularity. Ordering is determined by \rightarrow *control flow*. Common notations are \rightarrow *BPMN* and \rightarrow *EPC*.

Attribute: Belongs to an \rightarrow *object type* and represents a specific property, e.g., “Name” of object type “Person”.

BPMN: Business Process Model and Notation. Defines a model and a notation for \rightarrow *activity-centric processes*. Alternative to \rightarrow *EPC*.

Control flow: Induces an ordering on activities in \rightarrow *activity-centric process management systems*.

EPC: Event-driven Process Chain. A flowchart-like notation for \rightarrow *activity-centric process modelin*, can be used as alternative to \rightarrow *BPMN*.

Event-driven: Actions such as user interactions trigger an event, which elicits an appropriate response from the system. The process rule application is event-driven.

Form: Standard input method for \rightarrow *attribute* values of objects in a user interface. A form usually comprises textboxes, checkboxes, drop-down menus and other form input fields. \rightarrow *PHILharmonicFlows* generates forms automatically depending on the attribute data types.

-Instance: Suffix identifying a runtime class.

Macro process: Captures object interactions. Interactions depend on \rightarrow *states*. Details can be found in [17]

Maintainability: The ease with which a product can be maintained in order to isolate or correct defects or their cause. Maintainability also describes the property to repair or replace faulty or worn-out components without having to replace still working parts.

Markings: Markings convey additional information about \rightarrow *micro steps*, \rightarrow *micro transitions*, \rightarrow *states* and \rightarrow *backward transitions*. Process rules depend on markings.

Micro process: A micro process captures object behavior (Section 2.1.2). It consists of \rightarrow *micro steps*, \rightarrow *micro transitions*, \rightarrow *states* and \rightarrow *backward transitions*.

Micro step: A micro step represents an \rightarrow *attribute* of an \rightarrow *object type* in a \rightarrow *micro process*. It has \rightarrow *micro transitions* connecting to other micro steps. Belongs to a \rightarrow *state*.

A. Source

Value step: A value step represents permitted value or range of permitted values for an \rightarrow *attribute*. It is used in conjunction with \rightarrow *value-specific micro steps* and may have outgoing \rightarrow *micro transitions*.

Value-specific micro step: A value-specific micro step is a variant of a \rightarrow *micro step* where input is limited to certain values. Allowed values are represented by \rightarrow *value steps*.

Backward transition: A backward transition allows redoing previous work, e.g., in order to correct errors. A backward transition belongs to a \rightarrow *micro process*. It connects \rightarrow *states*, but not \rightarrow *micro steps* as normal \rightarrow *micro transitions*.

Micro transition: A micro transition connects \rightarrow *micro steps* with micro steps and \rightarrow *value steps* with micro steps. Micro transitions connecting steps in different \rightarrow *states* are called external.

Modeling environment: The modeling environment allows modeling \rightarrow *object types* and corresponding \rightarrow *micro processes* and \rightarrow *macro processes*. Modeled processes are deployed to the \rightarrow *runtime environment* in order to execute the processes.

Object Type: An object type is a logical grouping of certain \rightarrow *attributes*, e.g., a person that has a name, first name and age can be represented as an object type. Object types may also have relations to other object types.

PHILharmonicFlows: Framework for supporting object-aware processes. Consists of a \rightarrow *Modeling Environment*, \rightarrow *Runtime Environment* and \rightarrow *Runtime User Interface*.

Runtime environment: Executes \rightarrow *micro processes*. The current prototype does not support \rightarrow *macro processes*. The graphical front end for process execution is the \rightarrow *runtime user interface*.

Runtime user interface Allows for user interactions with the processes in the \rightarrow *runtime environment*. Automatically generates \rightarrow *forms* from underlying process data.

State Represents the processing state of a \rightarrow *micro process*. Comprises at least one \rightarrow *micro step*. Object interaction in \rightarrow *macro processes* is based on states.

-Type: Suffix identifying a design-time class.

Name: Sebastian Steinau

Matrikelnummer: 697293

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den

Sebastian Steinau