Universität Ulm
Abt. Datenbanken und Informationssysteme
Leiter: Prof. Dr. P. Dadam

Schema Evolution in Process Management Systems

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

vorgelegt von

STEFANIE BEATE RINDERLE
aus Memmingen
Oktober 2004

Amtierender Dekan:	Prof. Dr. H. Partsch
Gutachter:	Prof. Dr. P. Dadam Prof. Dr. F. Schweiggert
Tag der Promotion:	21. Dezember 2004

Preface

This thesis has been written during my work as a team member of the Department of Databases and Information Systems (DBIS) at the University of Ulm. The motivation for working on the project "Schema Evolution in Process Management Systems" stems from the fundamental approaches on e.g., process design, dynamic process changes, and distributed process execution already conducted within the ADEPT project as well as from experiences with many practical projects.

First of all, I thank my supervisor Prof. Dr. Peter Dadam for his guidance and support throughout my whole "thesis process". The fruitful discussions with him exceedingly contributed to make this thesis a success. Especially his enormous knowledge on related work and his problem–oriented view ("What's the message?") saved me from missing the red line. Finally, he always motivated me in being able to do anything I want.

Furthermore, I thank Prof. Dr. Franz Schweiggert who accepted to act as second referee. His comments on my work and the discussion with him were very important for my work and motivating for me.

Special thanks go to Dr. Manfred Reichert. The numerous discussion with him always helped me in proceeding with the content of my work. It was a pleasure to work with him on our diverse publications. His indefatigable effort to improve the quality of publications is admirable. I also thank him for proofreading my thesis.

I also want to thank the whole DBIS team for helping and motivating me during my whole time at the department. The friendly and collaborative atmosphere largely contributed to feel happy during working hours. Finally thanks go to all students who contributed to this work (especially Markus Lauer for the implementation of the proof–of–concept prototype, the fruitful discussion, and the proofreading of the algorithms).

My best thanks go to my parents Evi and Hermann Rinderle, my brother Matthias, my sister Julia, and my partner Oliver. Their love and support have made this work possible.

Danke!

Abstract

Continuously arising new trends in information technology and developments at the (e–business) market let companies crave for automated business process support. *Process management systems* offer the promising possibility to (electronically) define, control, and monitor business processes. However, if this technology shall be applicable in practice it must be possible to change running business processes even at runtime. Basically, such process changes can take place at two levels – the *process type* level and the *process instance* level. If a process type is modified a new version of the respective *process type schema* is created. Then, at minimum, the process instances running according to the old process type schema version must be able to finish without being disturbed. However, this simple versioning approach is only sufficient for short–running business processes. For long–running ones like, for example, car leasing contracts or medical treatment processes which may last from 3 up to 5 years, it must be possible to apply the process type changes to the collection of running process instances as well but without causing inconsistencies or errors in the sequel. Apart from process schema evolution and change propagation a flexible process management system must also enable instance–specific (ad–hoc) changes, for example, if exceptional situations occur. If then a process type change takes place the challenging question arises how to adequately deal with the interplay of process type and process instance changes.

Subject of this work is a formal framework for the comprehensive support of process type and process instance changes in respective management systems. Based on this framework it is possible to propagate process type changes to running process instances and to migrate *compliant* process instances to the changed process type schema afterwards. Thereby, a main goal is to provide a correct, efficient, and usable solution. Furthermore, process type change propagation can even be conducted at the presence of individually modified process instances.

For this, a formal foundation and respective algorithms are developed. We differentiate between process instances which still run according to the process type schema they were created on (*unbiased* process instances) and process instances which have been individually modified (*biased* process instances). The fundament of our approach is the formal framework for the migration of unbiased process instances to a changed process type schema. Thereby we present a formal and comprehensive correctness criterion as well as routines to efficiently check this criterion. Furthermore, we provide algorithms to automatically adapt process instance states when migrating these instances to the changed process type schema. To be able to adequately handle biased process instances we differentiate between two kinds of them – those ones for which their bias is *disjoint* with the process type change and those ones for which their bias *overlaps* the process type change. For process instances with disjoint bias we provide an adequate extension of our general correctness criterion not only covering state–related but also structural correctness. In addition, we present quickly to check structural conflict tests what contributes to the efficient applicability of the presented approach. Finally, we present an adequate migration strategy for process instances with disjoint bias. For process instances with overlapping bias we introduce a classification ranging from equivalent changes to changes with minor overlap. For all these classes of overlapping changes we provide migration strategies. These include the automatic re–linking of process instances to the changed process type schema and the necessary structural and state–related adaptations. All concepts presented have been implemented in a powerful proof–of–concept prototype.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For companies and large organizations the computerized support of their business processes becomes more and more important. Examples of such business processes include leasing contracts, customer orders, and medical treatment processes. For traditional application systems (e.g., enterprise resource planning systems) as well as for rapidly evolving e-business applications (e.g., e–procurement, supply chain management) a comprehensive process support is heavily desired by users [44, 31]. The same holds for technologies supporting enterprise application integration, which crave for business process integration as the glue for orchestrating distributed, heterogenous applications [3, 77]. Such process–aware information systems (PAIS) [125] shall allow explicit definition of the process logic, actively coordinate the execution and monitoring of processes, integrate distributed application components in a robust and secure manner, provide worklists to users, and be completely integrated with respect to documentation and authorization [52, 83, 139].

However, any computerized support of business processes without a vision for flexibility is short–sighted and expensive [88]. In practice, PAIS have to be adapted much more frequently than purely function–oriented information systems [32, 58]. Such process adaptations become necessary, for example, when new laws come into effect, optimizations or re–design of business processes take place [46, 110], or reactions on current market trends are required [44]. Doing so it is crucial to realize the necessary process changes and the adaptations of supporting PAIS very quickly, if need be even within a few days or hours. Ideally, it should be also possible to migrate the already running "old" business cases (i.e., process instances) – where desired and semantically reasonable – to the new process schema [26, 39, 100, 104, 118, 136].

Unfortunately, today's information systems either lack more comprehensive process support or business processes are implemented in an extremely unflexible way. Frequently, process control is directly coded within the application programs what makes application development as well as program maintenance very complex and error–prone. When implementing a new process type (e.g., the handling of a customer order), additional programming effort becomes necessary. Likewise future process changes cause immense costs for customizing the software.

Thus, IT departments which are already burdened with a high maintenance mortgage will run into massive problems.

1.1 Process Management Systems

Today a transition from data–centered to process–aware systems takes place [23]. The current trend towards service–oriented software architectures [3] increases the need for flexible application service composition. This trend is reflected at the conceptual level as well as at the system level. At the conceptual level, languages for process-oriented composition of (web) services like, for example BPEL4WS [5, 29] and BPML [84, 8] are developed. At the system level, new middleware systems like Microsoft BizTalk [79], IBM Websphere Choreographer [63], and SAP Netweaver [108] are to serve as platforms for the realization of these (service) flows.

The trend towards orchestration and choreography of service flows has revitalized the idea of process and workflow management [35, 74, 125, 122, 130]. Characteristic to *Process Management Systems (PMS)*[1] is the separation of process logic and application code, i.e., the logic of the supported processes (e.g., control and data flow between tasks) is explicitly modeled within the PMS and is not "hidden" within the program code (cf. Figure 1.1). Usually for this, graph–based languages like Petri Nets [1, 39, 118], UML Activity Charts [36], and Activity Nets [73] are used.

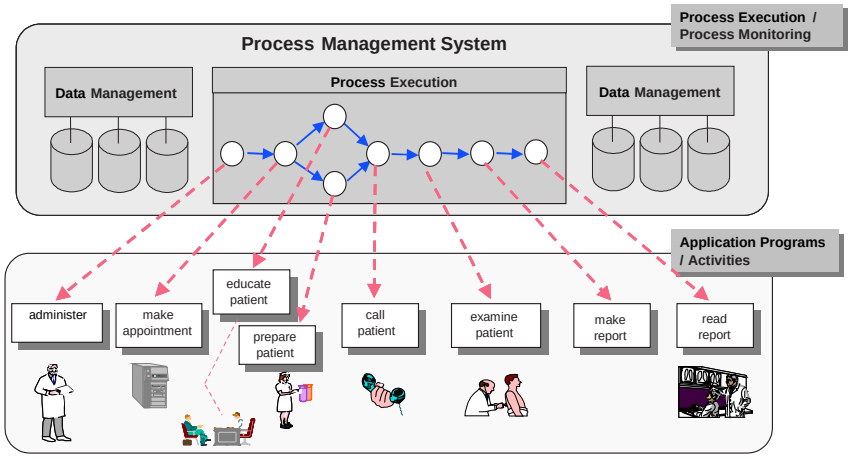


Figure 1.1: Separating Process Logic and Application Code Within PMS

In PMS for each *process type* T (e.g., purchase order handling) a *process type schema* S has to be generated and deposited within the PMS. In general, several process type schemes of a process type T may exists which comply to different versions of T. A process type schema

[1] We consciously avoid to use the term "Workflow Management Systems" due to its multiple meanings. We also want to achieve a higher degree of generality using the term "Process Management Systems" instead.

describes different aspects of a business process like control and data flow, actor assignments, or exception handling procedures (e.g., to jump backward in the flow in case of semantic failure). Furthermore, it is possible to assign application programs to single process steps (the so called *activities*) which are then executed during runtime. Based on a process type schema, *process instances* $I_1, \ldots, I_n$ can be created and started which are controlled by the PMS during their whole life time. The time span during which a process instance is active ranges from a few minutes up to several years as, for example, usual for car leasing processes. Process instances are executed, managed, and monitored by the PMS. The PMS coordinates the execution of process activities, offers upcoming activities within *worklists* to users, starts the associated application programs (with proper input data), and observes whether the activities are worked in due time.

The extraction of the process logic from its "hard–wiring" within application programs offers promising perspectives. Since the PMS takes over process control the development of applications becomes much easier. Due to the explicit modeling of the process logic the future system behavior can be evaluated in a very early stage such that design errors may be detected even before the application components are implemented. For the same reasons future process changes and associated adaptations of the application systems are principally simplified. If the application functions have been carefully implemented once, one may change the order of activities or insert new activities without concerning existing program modules. Another advantage of separating process logic from application programs is the reusability of process support functions, i.e., it is not necessary to newly implement these function for every process type or PAIS respectively.

1.2 Adaptive Process Management Systems

Despite the described perspectives of PAIS, PMS still lack widespread acceptance in practice. Today's process management technology is rather weak with respect to dynamic process changes. However, the reasons mentioned before force any company to change their business processes ever more frequently [1]. Therefore, a critical challenge for the competitiveness of any enterprise is its ability to quickly react to business process changes [37, 61, 98, 120, 127].

Basically, in PMS changes can take at two levels – the *process type* or the *process instance* level [133] (cf. Figure 1.2). Process type changes become necessary, for example, to adapt the PMS to optimized business processes or to new laws [91, 118]. In particular, applications supporting long–running processes (e.g., handling of leasing contracts or medical treatments) and the process instances controlled by them are affected by such type changes [89, 91]. As opposed to this, changes of single process instances (e.g., to insert, delete, or shift single process steps) often have to be carried out in an ad–hoc manner in order to deal with an exceptional situation, e.g., a sudden circulatory collaps of a patient or evolving process requirements [88, 87].

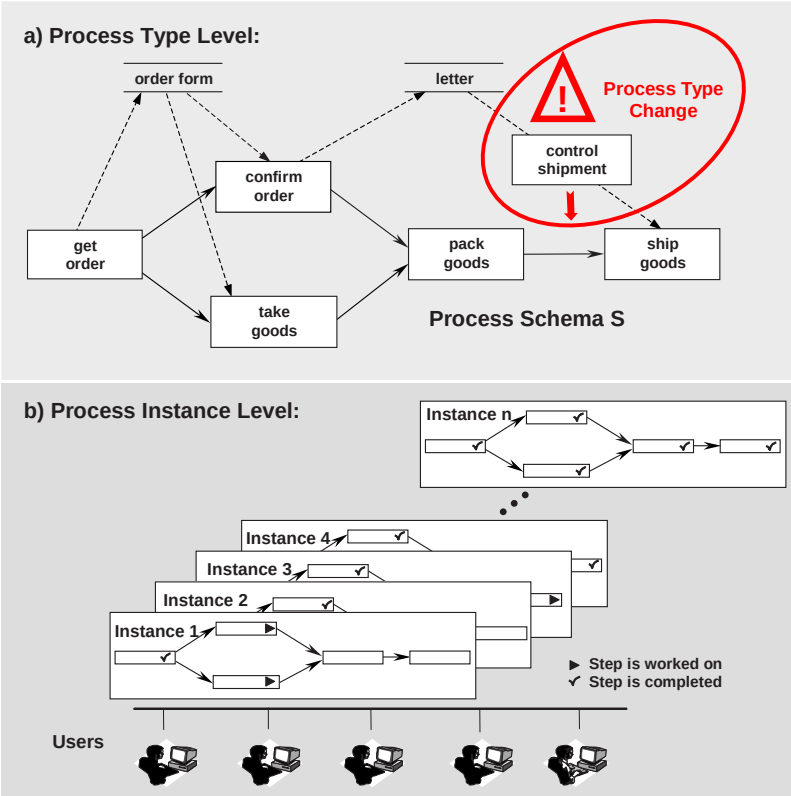


Figure 1.2: Order Process

1.2.1 Challenges Of Process Schema Evolution

Process type changes are handled by modifying the respective *process type schema* S based on which a collection of process instances $I_1, \ldots, I_n$ is running (cf. Figure 1.2). Of course, it is important that the modification of a *correct* process type schema S again results in a *correct* process type schema S'. Thereby a process type schema is denoted as being correct if it satisfies the correctness constraints set out by the underlying process meta model, i.e., the formalism to describe the business processes, e.g., Petri Nets. Examples for such constraints are the bipartite graph structure for Petri Nets [118] or the acyclic graph structure for Activity Nets [73]. This *static* schema correctness [26], for example, can be preserved by the applied change operations themselves. Examples for process meta models offering change operations preserving schema correctness are Flow Nets [117, 118] and ADEPT [87, 88].

From a static point of view, process schema evolution can be compared to the evolution of database schemes [95] which mainly happens at a static level as well; i.e., how to map data types, data structures, and integrity constraints of the "old" database schema to the respective data types, data structures, and integrity constraints of the "new" database schema (in a semantics–

preserving way).

This point of view is fundamental but not sufficient for practical purposes. After changing process type schema S we also have to deal with process instances $I_1, \ldots, I_n$ running according to S. Different strategies for this so called *dynamic process schema evolution* have been proposed in the workflow literature [105, 107]. The first "sledgehammer" method is to *cancel* all running process instances $I_1, \ldots, I_n$ and to re–start them according to the new process type schema S'. Of course, this strategy causes an immense loss of work and would usually not be accepted by users. Therefore, at minimum, we claim that process instances $I_1, \ldots, I_n$ started according to process type schema S can be finished according to S without being interrupted. This strategy can be implemented by providing adequate versioning concepts [67]. These concepts must ensure that future process instances are executed according to new process type schema S' whereas already running process instances are carried out according to the "old" schema version S. This simple strategy may be sufficient for processes of short duration. However, it raises grave problems in conjunction with long–running processes as common, for example, within clinical or engineering environments [20, 32]. Due to the resulting mix of process instances which run according to old process type schema S and those which run according to new process type schema S', a chaos within the production or the offered services may occur. Furthermore, an execution of process instances according to the old process type schema may be not acceptable if laws or business rules (e.g., clinical guidelines) are violated [112]. If these problems are not solved in a comprehensive way, companies can become even more inflexible when using PMS as their manual organization was before.

For these reasons it is quite important to be able to apply process type schema changes to running process instances $I_1, \ldots, I_n$ as well. We call this scenario the *propagation* of process type schema changes to running process instances or – in other words – the *migration* of the ("old") running process instances to the changed process type schema. Note that the number of running process instances $I_1, \ldots, I_n$ may be very large; in environments like hospitals or telecommunication companies, for example, $n > 10.000$ may easily hold [16, 57].

To better understand the challenge of propagating a process type schema change to running process instances one can compare a running process instance with an application program [134]. This program is currently executed and consists of several procedures. These steps can be successively executed (sequential execution), can be situated within if–then–else conditions (alternative branchings), or can be even executed in parallel threads. Process schema evolution can then be compared to manipulating the running program by inserting one or more procedures, deleting procedures, or changing the order of procedures in the midst of program execution. In particular, the changes have to be carried out by maintaining a correct program execution (e.g., correct parameter provision) for all possible execution alternatives.

Supporting the propagation of process type schema changes to running process instances, but without causing inconsistencies and errors in the sequel, is an extremely important task if PMS shall become really successful in practice. Today's commercial PMS lack an adequate support of process schema evolution. They either forbid the propagation of process type schema changes

to running process instances (e.g., MQSeries Workflow) or they allow inconsistencies and even system crashes after change propagation (e.g., Staffware) [78]. As a consequence, in practice, process descriptions are often split into a series of smaller, short-running process fragments that are maintained as separate schemes and correlated through application data at runtime [48]. Such a fragmented representation, however, does not provide a natural view of the process and is also unfavorable in other respects. In particular, it does not abolish the need for dynamic instance migrations (even if techniques such as late binding [33] are applied). Altogether the weak support of dynamic process changes is one of the major reasons why today's PMS are not fully accepted by users.

Consequently, it is crucial for PMS to enable process type change propagation to running process instances and the correct migration of the compliant process instances afterwards. This raises the following challenges:

1. **Completeness:** Process designers must not be restricted, neither by the used process meta model nor the offered change operations. Therefore, a process meta model ought to provide a complete set of control and data flow constructs, e.g., allowing the designer to model sequences, parallel/alternative branchings, and loops [37]. For practical purposes, at minimum, change operations for inserting and deleting activities as well as control/data dependencies between them are required. Furthermore, it must be able to combine change primitives to define complex changes, e.g., to modify the order of activities.

2. **Correctness:** The ultimate ambition of any adaptive PMS must be correctness of dynamic changes [98]; i.e., introducing changes to the runtime system without causing inconsistencies or errors (like deadlocks or improperly invoked activity programs). Therefore, adequate *correctness criteria* are needed. These criteria must not be too restrictive, i.e., no process instance should be needlessly excluded from applying a dynamic change.

3. **Efficient Compliance Checks:** Assume that we have found an appropriate correctness criterion for deciding on compliance of process instances with a changed process type schema. Then the challenging question is how to ensure this criterion *efficiently*. This is especially important for large-scale environments with hundreds up to thousands of running process instances [57]. The question behind is somewhat comparable to serializability of database transactions (correctness criterion). At runtime, serializability is not ensured by trying to find an equivalent serial execution for the schedule under consideration. Instead, efficient concurrency control mechanisms like two–phase locking or optimistic methods have been developed [30, 68].

4. **Usability:** The migration of process instances to a changed process type schema must not require expensive user interactions. Firstly, such interactions lead to delays within process instance executions. Secondly, users (e.g., designers or process administrators) must not be burdened with the job to adapt process instances to changed process type schemes. Complex process structures and extensive state adaptations quickly overstrain them. Therefore it is crucial to find methods to automatically determine those process

instances to which the process type change can be correctly applied. Furthermore, in sequel, it must be possible to automatically migrate these process instances to the changed process type schema.

Fulfilling challenges 1 – 4 is essential for the applicability and practicability of any adaptive PMS. However, the approaches presented in the workflow literature have disregarded one or more of these aspects so far.

1.2.2 Vision And Big Picture

The comprehensive support of process instance migrations to a changed process type schema along the stated challenges 1 – 4 is an important milestone on the way towards flexible and adaptive process management. However, process schema evolution alone is not sufficient in order to offer fully flexible PMS. The reason is that in addition to changes at the process type level (cf. Figure 1.2), it must be possible to modify single process instances as well (cf. Figure 1.3). Such process instance changes are often carried out in an ad–hoc manner in order to deal with an exceptional situation, e.g., an unforeseen correction of the inventory information in an order process [112] as depicted in Figure 1.3. In the following, we denote such individually modified process instances as *biased* process instances since their logical execution schema deviates from the process type schema they were derived from. Respectively, process instances which have not yet undergone an invidual change are called *unbiased* process instances.

In the literature process type and instance changes have been an important research topic for several years [1, 26, 37, 39, 67, 96, 98, 104, 118, 136]. However, there are only few adaptive PMS which support both kinds of changes in one system [65, 88, 100, 101, 136]. All of them (except ADEPT) have in common that once a process instance has been individually modified (i.e., it possesses an instance-specific schema), it cannot longer benefit from process type changes; i.e., changes of the process type schema they were originally created from. In WASA_2 [136], for example, a process instance change is carried out by deriving a new schema version to which the process instance is migrated. In the sequence, this instance is excluded from further adaptations of its original schema version at the process type level. However, doing so is not sufficient in many cases, especially in connection with long-running processes. Think of, for example, a medical treatment process which is normally executed in a standardized way for every patient. Assume that due to an unforeseen reaction an additional drug is given to a certain patient. However, this deviation from the standard process must not imply that this special process instance (and therefore the respective patient) is excluded from further process optimizations. Therefore, it must be possible to propagate process schema changes at the type level to such biased process instances as well.

Our vision is to master all possible kinds of process changes within a PMS in a correct and automatic manner. The basic case is to be able to migrate unbiased process instances to a changed process type schema obeying the four challenges described in Section 1.2. In addition, this thesis treats the **interplay** of process type and process instance changes. Note that process

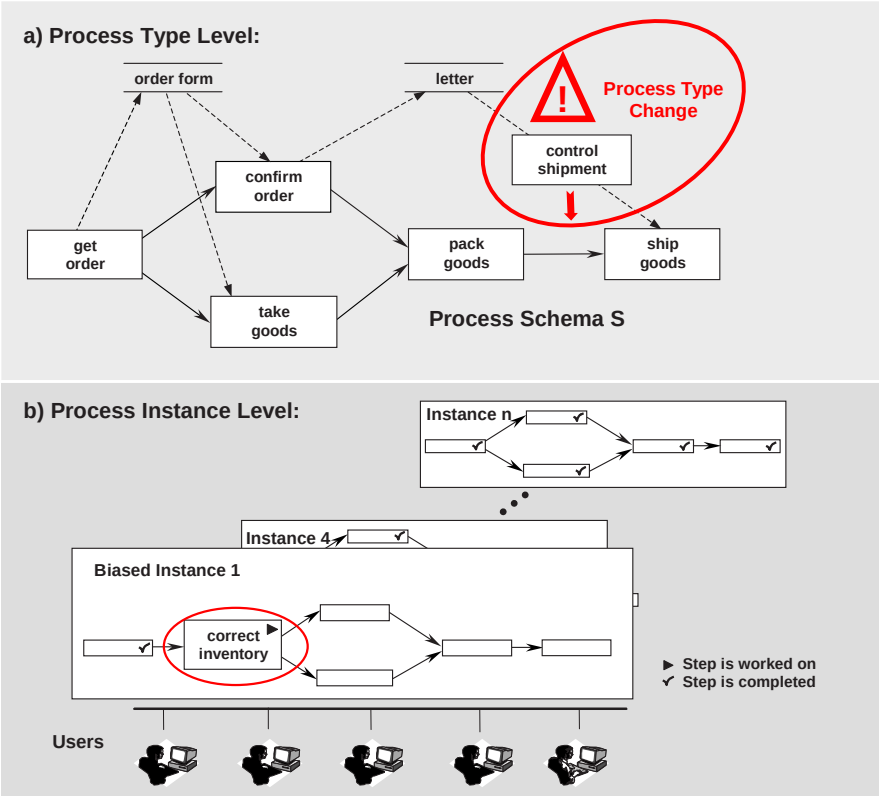


Figure 1.3: Biased Order Process Instance

instance changes and process type changes work on the same process schemes and can therefore be considered as *concurrent* changes on these schemes which may interfere with each other.

Concurrently applied process type and process instance changes raise many challenging issues. One of the most important ones is being able to distinguish between process instances for which their particular bias is *disjoint* with the process type change and process instances for which their particular process instance change and the process type change *overlap*. Disjoint means that process type and process instance change have totally different effects on the underlying process schema. This is the case, for example, if they work on different "regions" of the same process schema (for example, process type and process instance change as depicted in Figure 1.3). As opposed to this, process type and process instance changes overlap if they (partially) have the same effects on the underlying process schema. This is, for example, the case if some process instances have anticipated a future process type schema change in conjunction with a flawed process modeling. If then, subsequently, the respective process type schema change is applied to these process instances this may cause conflicts between the overlapping process type and process instance changes (e.g., if the same activities are deleted).

Why is it so important to differentiate between process instances having a disjoint bias and process instances having an overlapping one? As we will see in the sequel the correct handling of a particular process instance (when migrating it to a changed process type schema) depends on whether it has no bias, a disjoint bias, or an overlapping one. In other words, in order to be able to find and offer adequate *migration strategies* we have to be able to detect the different kinds of process instances. The resulting migration strategy comprises the re–linking of a process instance to the new type schema as well as the necessary state and structure adaptations.

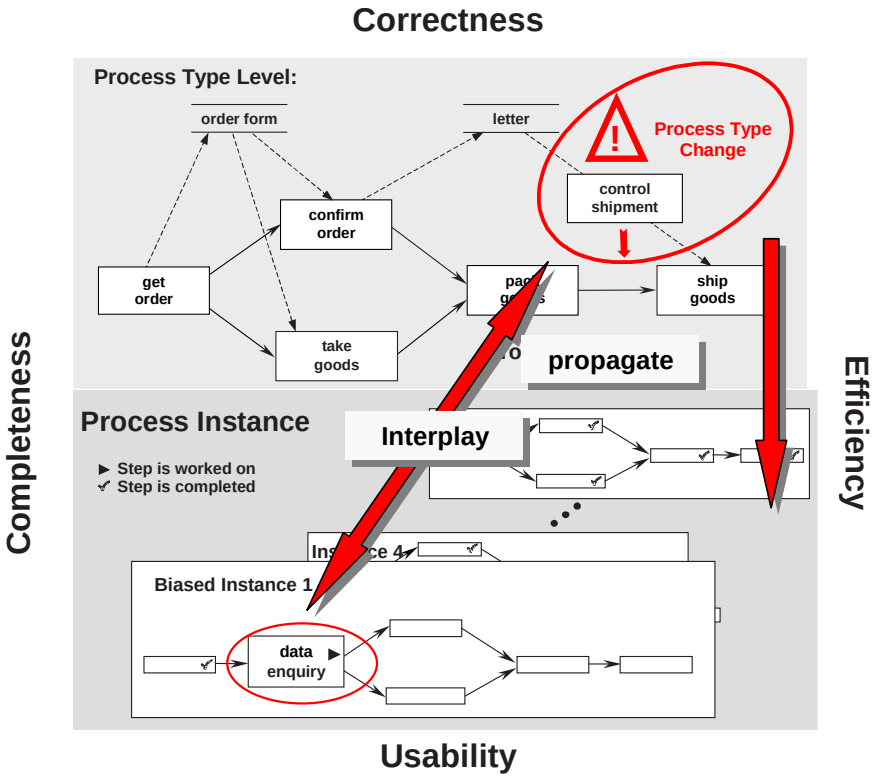


Figure 1.4: Our Vision Of Adaptive Process Management

Of course, any solution for migrating biased process instances to a modified process type schema must further fulfill challenges 1 – 4 (cf. Figure 1.4). If we are able to find adequate solutions we achieve our aim of a PMS fully supporting users in any change they carry out. In detail, this means to provide adequate correctness criteria which are applicable for the migration of unbiased as well as for the migration of biased process instances. In order to be able to deal with a multitude of running process instances it is necessary to find methods to quickly and efficiently ensure the stated correctness criteria for both cases, unbiased and biased process instances. Furthermore, usability of an adaptive PMS depends on whether it is able to automatically migrate unbiased and biased process instances or not, i.e., without costly and cumbersome user interactions. In addition, meaningful messages have to be generated which, for example,

report and explain the classification of particular process instances to users. Furthermore, users must be able to exclude (certain) process instances from migration if desired. Finally, users should be also able to choose between different migration strategies.

Altogether, a solution fitting in our vision of adaptive process management releases users from the possibly rigid corset they are pressed in by current systems. If this comes true the advantages of PMS become so mind–blowing that they will finally make their expected breakthrough in practice.

1.3 Aims and Organization of this Work

Aim of this work is to develop a formal framework for the support of process type and process instance changes within PMS which does not only consider the single aspects of process changes in a separated manner but also analyzes their interactions. Thereby, we follow the four challenges set out in Section 1.2, namely completeness of the solution, system correctness, efficient implementation of the presented concepts at the logical and physical level, and user–friendly realization of the solution. More precisely, this thesis addresses the following questions.

Scenario 1: Migration of Unbiased Process Instances: Consider Figure 1.5. Process schema S is changed by inserting new activity *control shipment*. The challenging question is how to treat the process instances running on S. In the present scenario these instances have not been individually modified, i.e., they are unbiased.

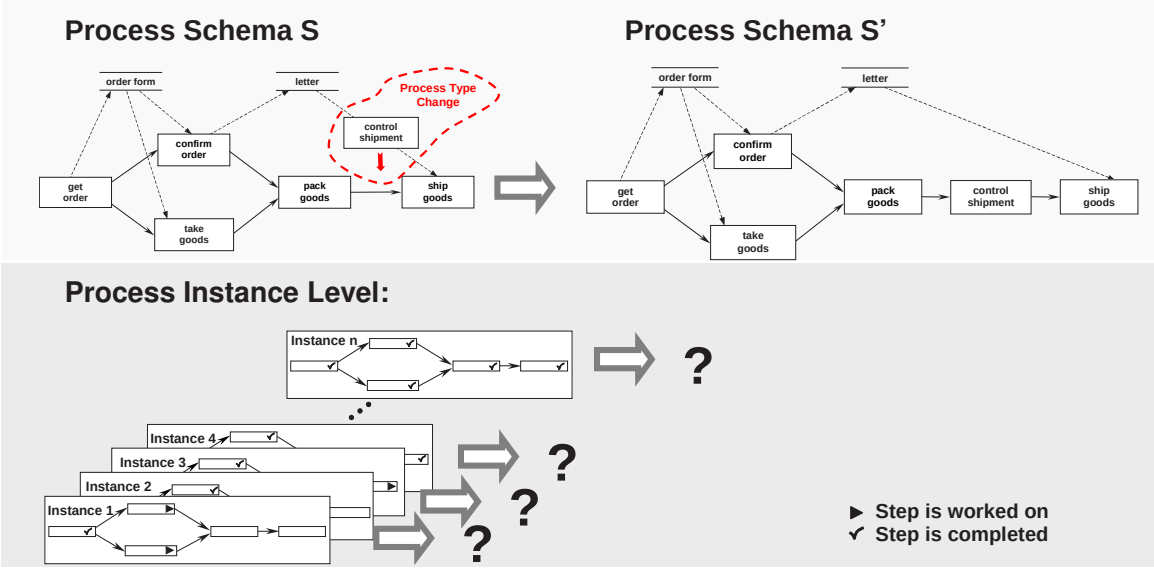


Figure 1.5: Scenario 1: Migration of Unbiased Process Instances

The support of Scenario 1 poses the following questions:

- How can we ensure that the migration of unbiased process instances to a changed process type schema is done correctly? In particular, what kind of correctness criteria can serve as basis for respective compliance checks without unnecessarily restricting expressiveness of the used process meta model or completeness of the offered change operations?
- Can we find methods to quickly and efficiently decide on compliance criteria in order to avoid interruption of process instance executions for a longer time?
- How can we automatically and efficiently adapt (compliant) process instances when migrating them to the changed process type schema?
- How do we deal with *non–compliant* process instances, i.e., process instances which cannot be migrated to the changed process type schema according to the correctness criterion?

Scenario 2: Migration of Biased Process Instances: Consider Figure 1.6. Process schema S is changed by inserting new activity *control shipment*. All depicted process instances have been already modified by previous ad hoc changes, *Instance 1* by deleting activity *confirm order*, *Instance 2* by inserting activity *control shipment*, and *Instance 3* by inserting activities *control shipment* and *send reminder*. Thereby the instance change for *Instance 1* is disjoint with the process type change since they work on totally different "regions" of the underlying process schema. By contrast, the instance changes for *Instance 2* and *Instance 3* overlap with the process type change since they have (partially) the same effects on the underlying process type schema.

The challenges arising for Scenario 2 can be summarized as follows:

- How can we appropriately define the notions of disjoint and overlapping process type and process instance changes?
- How can we efficiently determine whether process type and process instance changes are disjoint or overlapping?
- Which criteria are sufficient to guarantee correctness when migrating biased process instances to a changed process type schema? In particular, can we extend the compliance criteria found for unbiased process instances (cf. Scenario 1) in order to fit for biased process instances as well? Can we even find a general correctness criterion for all kinds of process instances?
- How can we efficiently check extended compliance for biased process instances, e.g., are there methods which avoid to lock process instances for a longer time?
- How can we classify overlapping process changes according to their particular degree of overlap (equivalent, partially equivalent, etc.)?

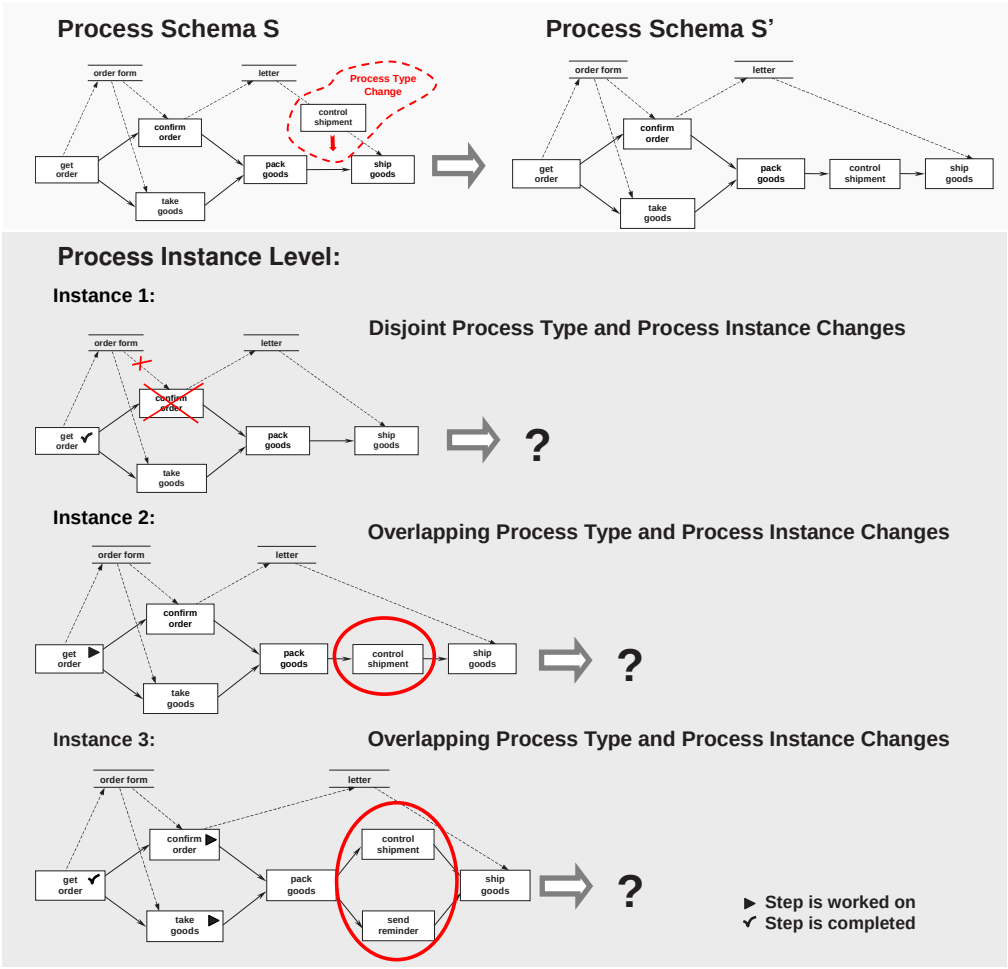


Figure 1.6: Scenario 2: Migration of Biased Process Instances

- For which degrees of overlap between process type and process instance changes, an automatic treatment is possible (e.g., if the bias of a particular instance has anticipated the process type change)? What are reasonable default strategies for these cases?
- For degrees of overlap between process type and process instance changes for which no (default) migration strategy can be determined: can we find decision rules in order to ease their handling for users?
- How can we adequately guide users through the migration process? In particular, which cases have to be reported?
- How can we deal with non–compliant process instances? Which are the different kinds of non–compliance and what are adequate solutions?

- How can we implement the concepts for adaptive process management in a reasonable way? Which possibilities do we have for additional optimizations at the physical level?

All these challenges are not only a point of how to technically realize an adaptive PMS. In fact, these questions demand theoretical foundations and concepts. Since the mentioned aspects depend on each other they determine the order for the organization of this work.

In Chapter 2 we discuss related work on adaptive process management. Thereby, we firstly establish a classification of the different approaches along their specific process meta model and the theoretical foundations of their concepts for adaptive process management. Secondly, we introduce five typical problems in conjunction with process changes. Along these problems we discuss the different approaches and show their specific strengths and limitations. After this we present further comparisons between the different approaches before we offer a detailed discussion about the different aspects of dynamic process changes. These considerations comprise ad–hoc change and process schema evolution in research prototypes as well as in commercial systems and "newer" paradigms as, for example, case handling [126]. From this related work discussion different approaches and suggestions for correctness criteria in conjunction with adaptive processes are extracted. They serve as basis for the further considerations.

Chapter 3 summarizes important background information about the used process meta model used in this thesis. In detail, we introduce WSM Nets as a process description language to define process schemes and show how process instances can be started, executed, and individually modified based on WSM Nets. Furthermore we present a set of change operations and change transactions offered to users in our prototype.

Chapter 4 provides a complete solution for the migration of unbiased process instances to a changed process type schema. We start with a detailed discussion of the particular challenges in this context. Then we have a look on existing correctness criteria (cf. Chapter 2) and choose one of them as the basis for our further considerations. However, we adapt and extend this correctness criterion in order to overcome its limitations. After this we present precise compliance conditions based on which it is possible to quickly and efficiently ensure the imposed correctness criterion (we also formally prove this). To release users from the burden of manually adapting instance states when migrating compliant instances to the changed process type schema we provide respective algorithms. These algorithms automatically adapt process instance markings for all kinds of applicable changes with linear complexity. Finally, we discuss several approaches how to deal with non–compliant process instances and we sketch their particular strenghts and limitations. Chapter 4 finishes with a summary of the presented results.

In Chapter 5, firstly the challenges arising in the context of migrating biased process instances to a changed process type schema are presented. One extremely important challenge is to be able to distinguish between process instances with disjoint and process instances with overlapping bias. Based on the particular kind of process instances the further migration strategy depends. Therefore we provide the formal framework for dividing the set of biased process instances into those with disjoint bias and those with overlapping bias. This formal framework is based on the theoretical notions of process trace equivalence and process schema isomorphism. In the

remainder of this chapter we deal with the correct migration of process instances with disjoint bias. Therefore we firstly establish an adequate correctness criterion which is a generalization of the correctness criterion introduced for the migration of unbiased process instances in Chapter 4. We present methods to quickly verify the extended parts of the general correctness criterion which are also formally proven. Finally, the strategy for migrating process instances with disjoint bias to a changed process type schema is given together with automatic adaptation methods. A summary closes Chapter 5.

Chapter 6 deals with advanced migration issues; i.e., how to migrate process instances for which their bias overlaps with the process type change. Therefore, firstly, the challenges arising in this context are discussed. In order to adequately meet these challenges it is fundamental to find a further classification of process instances for which their bias and the process type change overlap. This classification is carried out along the particular *degree of overlap* between respective process type and process instance changes. However, determining this degree of overlap is far away from being trivial, and so we have to discuss different approaches ranging from *structural* ones to *operational approaches* as well as a specific combination (*hybrid approach*) between them. In this context, we present a very interesting method to *purge noisy* information from change logs. Doing so we obtain a canonical representation of change logs which is useful input for the final classification of process instances. Furthermore, we show that the determination of the degree of overlap between process type and process instance changes becomes too rough if we use whole change logs as basis. Therefore we introduce the concept of change log *projections* which reflect the different types of changes. Together with these projections and the hybrid approach we are able to formally define the different degrees of overlap between process type and process instance changes. Along the particular degree of overlap we present sophisticated migration strategies which can be automatically (and correctly) applied if users want to do so. For those cases for which no automatic migration strategy can be deposited within the PMS we state meaningful decision rules which support users in finding the right treatment of the "handish–to–deal" process instances. Finally, we present algorithms which calculate the remaining bias after propagating a process type change to the respective process instance. Chapter 6 end with a summary of the presented results.

In Chapter 7, we present our proof–of–concept prototype which implements and demonstrates the theoretical results of this work. We start with introducing the architecture of this system and explain the different components. After this, a complete example for process change management is presented at an abstract level. In this context, we also show how this example is implemented within the prototype. This example ranges from the migration of unbiased process instances over the migration of process instances with disjoint bias to the migration of process instances with overlapping bias. In particular, we show how efficiently the prototype works when checking compliance and when adapting unbiased and biased process instances. Finally, we close this chapter with a summary.

Chapter 8 summarizes the results of this work and gives an outlook on continuative aspects like semantical issues and changes of other components of the PAIS (e.g., the organization model).

Chapter 2

Related Work

2.1 Schema Evolution in Database Management Systems

Obviously, there are similarities between schema changes in PMS [26, 100, 104, 118] and in database management systems (DBMS) (e.g., [4]). The underlying problems are similar if considerations are restricted to the mapping of schema elements (activity nodes, control/data flow edges) from the old to the new schema. These static adaptations can be compared to database operations at the schema level, like `CREATE/DROP/ALTER TABLE`. Such database schema adaptations become necessary, for example, in conjunction with schema integration in distributed and federated databases [14, 28, 30]. This aspect has been analyzed for a long time and is well understood in the meantime [4, 85, 28, 30, 69]. Process schema evolution, however, raises additional orthogonal issues. If changes at the process type level shall be applied at the process instance level as well, one has to consider that process instances may be in different states when a schema change propagation takes place. Depending on their current state and on the applied change operations, a migration to the new schema may then be possible or not. For deciding which instances are compliant with the new schema and can therefore be migrated to it, sound and efficient solutions are required. Furthermore, we have to deal with concurrent schema changes at the process type and the process instance level in order to provide an appropriate process change management.

2.2 Challenges for Approaches Dealing With Process Schema Evolution

Propagating changes of the process type schema to already running process instances has been a field of interest in the literature [1, 26, 39, 67, 104, 118, 136] for a longer time. In the following we present the existing approaches and discuss how far they lead towards a correct and manageable support for process schema evolution as claimed in the introduction. In detail we compare the

approaches along the following aspects:

1. **Completeness:** Users should not be unnecessarily restricted, neither by the applied process meta model nor the offered change operations. Therefore, expressive control/data flow constructs must be provided [37]. For practical purposes, at minimum, change operations for inserting and deleting activities as well as control/data dependencies between them are needed.

2. **Correctness:** The ultimate ambition of any adaptive approach must be correctness of dynamic changes [1, 26, 39, 67, 88, 104, 118, 136]. More precisely, we need adequate *correctness criteria* to check whether a process instance I is *compliant* with a changed process type schema or not; i.e., whether the respective change can be correctly *propagated* to I without causing inconsistencies or errors (like deadlocks or non–conform data flows). These criteria must not be too restrictive, i.e., no process instance should be needlessly excluded from being adapted to a process schema change.

3. **Change Realization:** Assuming that a dynamic change can be correctly propagated to an instance I (along the stated correctness criteria), it should be possible to automatically *migrate* I to the new schema. In this context, one challenge is to correctly and efficiently adapt instance states.

4. **Interplay Between Process Type and Instance Changes:** To support really flexible PMS it must be possible to do both, process schema evolution and ad hoc modifications of single process instances. Furthermore, it is not sufficient to forbid the propagation of process schema changes to already modified (biased) process instances. Therefore an important challenge for any flexible PMS is to adequately support the interplay between process type and process instance changes.

In the following, we provide a classification of actual approaches based on the operational semantics of the underlying process meta models and on the kind of correctness criteria applied for dynamic process changes (Sections 2.3 + 2.4). Section 2.4 also introduces a selection of typical *dynamic change problems* and discusses strengths and weaknesses of the approaches when dealing with these problems. A detailed comparison of the different approaches is presented in Section 2.5. We sketch important change scenarios and existing approaches in Section 2.6 and close with a summary in Section 2.7.

2.3 Process Meta Models of Approaches Dealing With Process Schema Evolution

Current approaches supporting adaptive processes are based on different process meta models. Very often, the solutions offered by them are dependent on the expressiveness as well as on the formal and operational semantics of the used formalism. Figure 2.1 summarizes process meta

models for which adaptive process solutions have been realized. According to [60] we classify those meta models with respect to their operational semantics and how they represent process instance states.

The first strategy uses only one type of (control flow) token passing through each process instance (*True-Tokens*). The other strategy is based on two types of tokens – *True-* and *False-Tokens*. Simplistically, True-Tokens trigger activities that are to be executed next and False-Tokens describe skipped activities. Formalisms which solely use True-Tokens include, for example, Petri-Nets [1, 39, 118] (cf. Section 2.3.1). Approaches which, in addition, use False-Tokens to represent skipped activities or skipped execution branches can be found in the area of graph-/activity-based meta models [26, 67, 88, 104, 136] (cf. Sect. 2.3.2). They can be further divided according to the way they represent the tokens. One possibility is to gain them from *execution histories* [26], with log events like activity start and activity completion. Alternatively, special (*model-inherent*) activity markings, which represent a consolidated view on the execution histories, can be used [88, 104, 136].

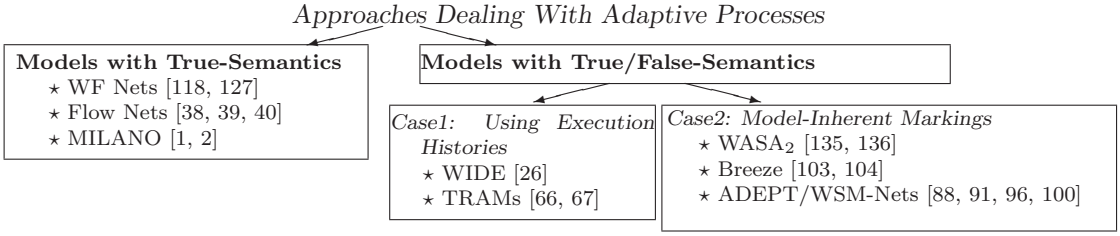


Figure 2.1: Meta Models of Approaches Supporting Adaptive Processes

In the following, for each approach shown in Figure 2.1, we sketch the basic formalism used for process modeling and execution together with its structural and dynamic properties. This background information is useful for better understanding the criteria applied by these approaches to guarantee dynamic change correctness (cf. Sect. 2.4).

2.3.1 Approaches With True-Semantics

All models with True–Semantics summarized in Figure 2.1 are based on Petri Nets [94, 102]. Figure 2.2.a shows a (marked) Petri Net for which the (True) token signals that activity C (called transition in this context) is ready to fire. The marked Petri Net resulting after completion of C is depicted in Figure 2.2.b. From this example the essence of models with True–Semantics can be deduced: After firing of transition C it can be seen which activities are currently activated (namely activity F) but it is not clear which firing sequence has led to the current marking.

Now we introduce the basic formalisms for the different models with True–Semantics (cf. Figure 2.1).

WF Nets: A *WF Net* [118] is a labeled place/transition net $N = (P, T, F, l)$ representing

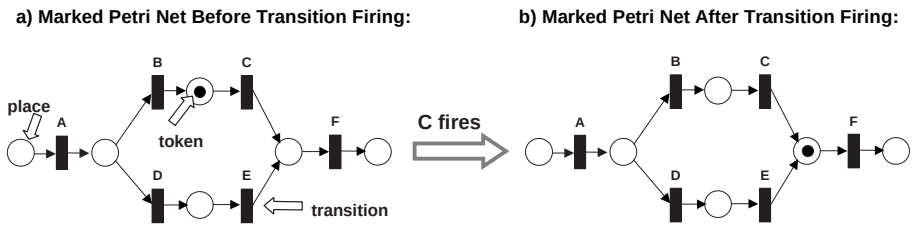


Figure 2.2: Marked Petri Net Before and After Transition Firing

a control flow schema (cf. Figure 2.10). Thereby, P denotes the set of places, T the set of transitions, $F \subseteq (T \times P) \cup (P \times T)$ the set of directed arcs, and l the labeling function, which assigns a label to each transition. Data flow issues are not considered. A WF Net must have one inital place i and one final place f. In [118] a *sound* WF Net has to be connected, safe, and deadlock free as well as free of dead transitions. Furthermore, sound WF Nets always properly terminate, i.e., the end state – which contains one token in f and no other tokens – is always reachable. The behavior of a process instance is described by a *marked WF net* (N, m) with marking function m and associated firing rules. A transition t is enabled if each of its input places contains a token. If t fires, all tokens from its input places are removed and to each output place of t a token is added.

Flow Nets: The operational semantics of Flow Nets [39] is comparable to (safe) WF Nets but with one major difference: Places can be equipped with more than one token. Process instances I_1 and I_2 as depicted in Figure 2.13(1) would actually run on the same process schema but being distinguished by tokens of different color. In this thesis, for presentation purposes, we separate I_1 and I_2 into two marked Flow Nets.

Chautauqua [40] offers an implementation where Flow Nets are generalized to *Information Control Networks (ICN)*. An ICN bases process enactment on instance-specific data tokens. Different process instances are again distinguished by coloured tokens and are controlled by the same ICN.

MILANO Nets: Another Petri-Net-based approach is offered by MILANO [1, 2]. As opposed to WF Nets and Flow Nets the expressiveness of MILANO Nets is restricted to marked, acyclic Free-Choice Petri Nets (so called *Net Models (NM)*). Data flow is not explicitly considered. A NM S can be mapped to a *Sequential Model (SM)* (cf. Figure 2.12(1)) which represents global states and state transitions of S. Thus, *SM* corresponds to the *reachability graph* of S.

Interestingly, the above approaches abstract from internal activity states, i.e., they only differentiate between activated and non-activated transitions. As we will see later, this coarse classification may be unfavorable in conjunction with certain kind of dynamic changes.

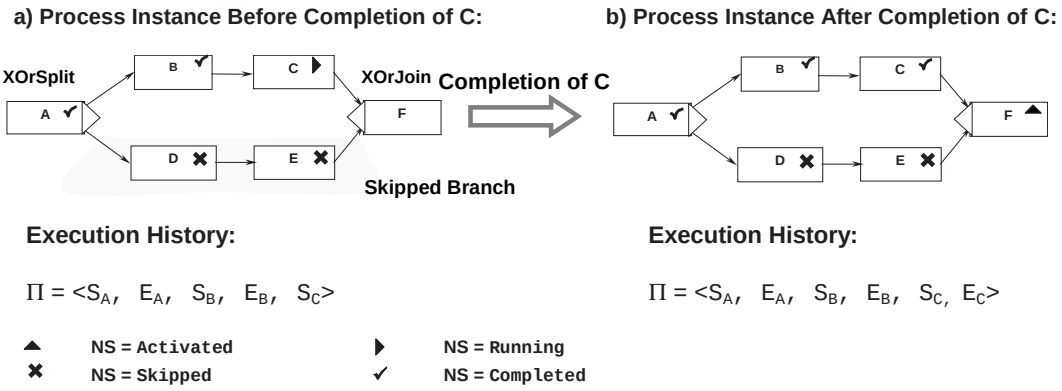


Figure 2.3: Instance Execution for Models with True–/False–Semantics

2.3.2 Approaches With True–/False-Semantics

As opposed to the above approaches, the following process meta models distinguish between different states an activity may go through. Regarding the example depicted in Figure 2.2 we cannot distinguish whether activity C is only activated (and therefore offered in worklists) or is already worked on. As shown in Figure 2.3.a this distinction is possible for models with True–/False–Semantics (here activity C is actually running).

Figures 2.2 and 2.3 illustrate the main difference between models with True–Semantics and models with True–/False–Semantics: For models with True–/False–Semantics we always know the previous process instance execution, in particular which branches have not been chosen for execution (cf. Figure 2.3.a) what is actually not the case for models with True–Semantics (cf. Figure 2.2.b).

Basically, there are two possibilities to represent process instance states using a True–/False–Semantics (cf. Figure 2.1): The first one is to maintain an execution history $\Pi_I^S = < e_0, \ldots, e_k >$ for each instance I with $e_i \in \{$`(S`$_a$`, <var, val>*), (E`$_a$`, <var, val>*)`$\}$ (cf. Figure 2.3). Thereby `S`$_a$ corresponds to the event of starting activity a and `E`$_a$ to the event of completing activity a. For each started activity X the values of process data elements read by X and for each completed activity Y the values of data elements written by Y are logged.

The other possibility is to use model–inherent markings[1] (cf. Figure 2.3). Generally, the initial status of an activity is set to `NotActivated`. It changes to `Activated` when all preconditions are met. Either the activity execution is then started automatically or corresponding worklist entries are generated. When starting the activity execution the activity status changes to `Running`. Finally, at successful termination, status passes to `Completed`. In addition, some of the models assign status `Skipped` to activites belonging to non-selected execution branches.

[1]Note that often an execution history Π_I^S is stored in addition.

Case 1: Approaches Using Execution Histories

WIDE Graphs: WIDE [26] uses an activity-based process meta model which allows the modeling of sequential, parallel, conditional, and iterative activity executions (examples for WIDE graphs are depicted in Figure 2.14). A process schema has to meet several constraints to be correct: First there must be a path from the start activity to all other activities and the end activity has to be reachable from all of them. The other constraints refer to the correct use of splits and joins. Furthermore, each process schema S is associated with a set of global process variables whose values may be read or written by activity instances during runtime. A particular process instance I is described by its schema S and its execution history Π_I^S. As opposed to the following approaches, WIDE only logs activity completion events.

TRAMs Graphs: In TRAMs [66] – in contrast to other graph-/activity-based approaches – control flow is not realized by control edges (cf. Figure 2.15). Instead, it is described in a declarative way by using conditions for starting/finishing activities. Process schema correctness is preserved by invariants, i.e., schema-related conditions which must be fulfilled. Data flow is explicitly specified by connecting output and input parameters of subsequent activities. TRAMs logs status changes (start and end events of activities) in the execution history Π_I^S of the respective instance I.

Case 2: Approaches Using Model Inherent Markings

WASA$_2$ Activity Nets: WASA$_2$ [135] uses an activity-based meta model. A *process schema* $S = (V_S, C_S, D_S)$ is a tuple with sets of activity nodes V_S, control connectors $C_S \subset V_S \times V_S$, and data connectors $D_S \subset V_S \times V_S$ (cf. Figure 2.11). Similar to TRAMs, the flow of data is modeled by connectors which map output and input parameters of subsequent activities. A process schema S is correct iff all input parameters are correctly mapped onto a type-conform output parameter and the graph structure is acyclic, i.e., loops are excluded. A *process instance* I is described by an instance graph $I = (V_I, C_I, D_I)$ whose state is denoted by model-inherent activity markings. WASA$_2$ distinguishes between markings `NotActivated`, `Activated`, `Running`, `Completed`, and `Skipped`.

Breeze Activity Nets: Similar to TRAMs and WASA$_2$, Breeze [104] uses model-inherent activity markings. A process schema is described by a directed acyclic graph $W =<$N, F$>$ with finite set of activity nodes N and flow relation $F \subseteq N \times N$. It is possible to model sequences, parallel/conditional branchings and complex activities. Process data is described by a set of WF variables. A process schema is correct if there is a unique initial node n_i and a unique final node n_f, and for all n $\in N$ there is a path from n_i to n_f via n. Breeze also ensures correct data provision of invoked activities.

Comparing the above formalisms we can find many differences. For example, only WF Nets, Flow Nets, and WIDE Graphs allow the modeling of loops. Data flow issues are factored out by WF Nets and MILANO Nets. While WIDE only logs end entries of activities, the execution histories in TRAMs store the start entries of activities as well.

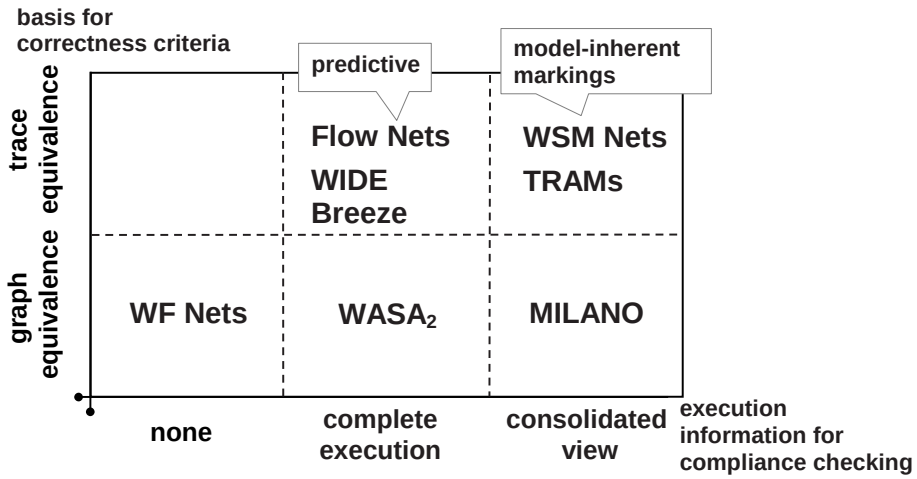


Figure 2.4: Classification of Approaches Along the Applied Correctness Criteria

2.4 Approach Classification and Dynamic Change Correctness

In this section, we present a classification of the approaches introduced in Section 2.3. It is based on the *correctness criteria* applied in conjunction with dynamic process changes. This classification is fundamental for better understanding the different solutions as well as their strengths and limitations. At this point, we do not make a difference between changes of single instances and adaptations of a collection of instances (e.g., due to a process type change). Instead we focus on fundamental correctness issues related to dynamic process changes.

In the following, let S be a process schema and let I be a process instance based on S. Assume that S is transformed into another correct schema S' by applying change Δ. What schema correctness exactly means depends on the structural and dynamic correctness properties of the used process meta model (cf. Sect. 2.3). Examples are the maintenance of the bipartite graph structure for Petri Nets or the maintenance of the acyclic graph structure for Activity Nets.

2.4.1 Classification and Problem Framework

Figure 2.4 presents a two-dimensional classification: The first dimension (marked on the vertical axis) is grouped by the kind of correctness criteria the different approaches are based on. The second dimension (marked on the horizontal axis) indicates on which information the different approaches check their particular correctness criterion. Regarding the first dimension, we distinguish between approaches founding their correctness criteria on *process graph equivalence* – *WF Nets*, *MILANO Nets*, and *WASA$_2$ Activity Nets* – and approaches with correctness criteria based on *process trace equivalence* – *Flow Nets*, *WIDE Graphs*, *Breeze Activity Nets*, *TRAMs*

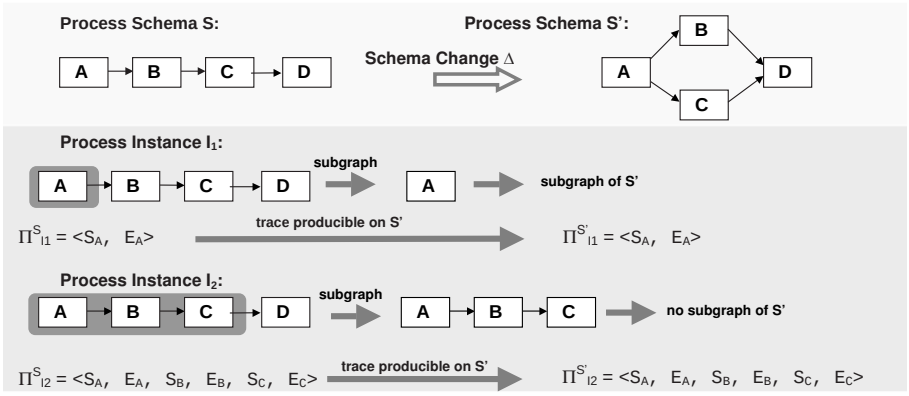


Figure 2.5: Basic Idea of Trace and Graph Equivalence

Graphs, and *ADEPT WSM-Nets*. The core idea of *graph equivalence* is either to compare the respective process schema before and after the change [118] or to map the process instance (sub)graph of I reflecting its previous execution to the changed process schema S' [1, 136]. Consider Figure 2.5. For process instance I_1 the depicted subgraph contains already completed activity A. Since this graph is also a subgraph of changed process schema S' schema change Δ is applicable to I_1. By contrast, the respective subgraph for process instance I_2 consisting of completed activities A, B, and C is not a subgraph of S'. Consequently, approaches based on graph equivalence (e.g. [136]) would reject the application of Δ to I_2.

Generally, *trace equivalence* focuses on the work done by I so far [26, 67, 104]. If it could have been achieved on S' as well, I can be migrated to S'. For example, the execution histories of process instances I_1 and I_2 (cf. Figure 2.5) are also producible on changed process schema S'. Note that in the case of process instance I_2 trace equivalence is less restrictive than graph equivalence.

As a special kind of trace equivalence is proposed by Flow Nets [39]: Here, a *predictive* approach is offered where, in addition, future instance execution on the changed schema is taken into account.

Regarding the second dimension of our classification (marked by the horizontal axis in Figure 2.4), approaches can be further distinguished depending on how they check their particular correctness criterion. Some of them consider complete history information of respective instances [26] whereas others use a consolidated view of previous instance execution [67, 100].

We show how the approaches from Figure 2.4 ensure correctness in conjunction with dynamic process changes. In addition, for comparison purposes, we discuss the approaches along the five problems depicted by Figure 2.6. These problems are very typical in the context of dynamic changes and therefore provide a good basis for comparing existing approaches.

(1) *Changing the Past Problem.* This problem corresponds to the rule of thumb not to "change the past of an instance". Neglecting this rule may lead to inconsistent instance states (e.g., livelocks or deadlocks) or missing input data of subsequent activity executions. Consider, for example, Figure 2.6(1). The insertion of activity X before already completed activity B changes the past of process instance I. One consequence could be that X is not executed, therefore would not write data element *data*, and in the following Y would not be correctly supplied with input data.

(2) *Loop Tolerance Problem.* This problem refers to an approach's ability to correctly and reasonably deal with changes on loop structures (see Figure 2.6(2)). In particular, approaches should not needlessly exclude instances from migrating to a new schema solely based on the fact that the respective changes affect loops.

(3) *Dangling States Problem.* This problem arises in conjunction with approaches not distinguishing between activated and started activities (see Figure 2.6(3)). As a consequence, very often such approaches either forbid the deletion of activated activities (i.e., activities put into user worklists but not yet started) – what is too restrictive – or they allow the deletion of such activities – what may lead, in case of already started activities, to loss of work.

(4) *Order Changing Problem.* This problem refers to the correct adaptation of instance markings when applying order changing operations like parallelization, sequentialization, and swapping of activities (see Figure 2.6(4)).

(5) *Parallel Insertion Problem.* This problem may arise when inserting a new parallel branch. Concerning Petri Nets, for example, after such a change we may have to insert additional tokens to avoid deadlocks in the sequel (see Figure 2.6(5)). The Order Changing and the Parallel Insertion problems are closely related to the *dynamic change bug* (cf. Figure 2.8) as it has been presented in [118].

In the following we refer to these characteristic problems as the *dynamic change problems*, and we show how the different approaches supporting adaptive workflows deal with them.

2.4.2 Approaches based on Graph Equivalence

The approaches discussed in this section base their particular correctness criteria on graph equivalence. These approaches can be further subdivided into approaches which do and which do not use instance execution histories for checking compliance.

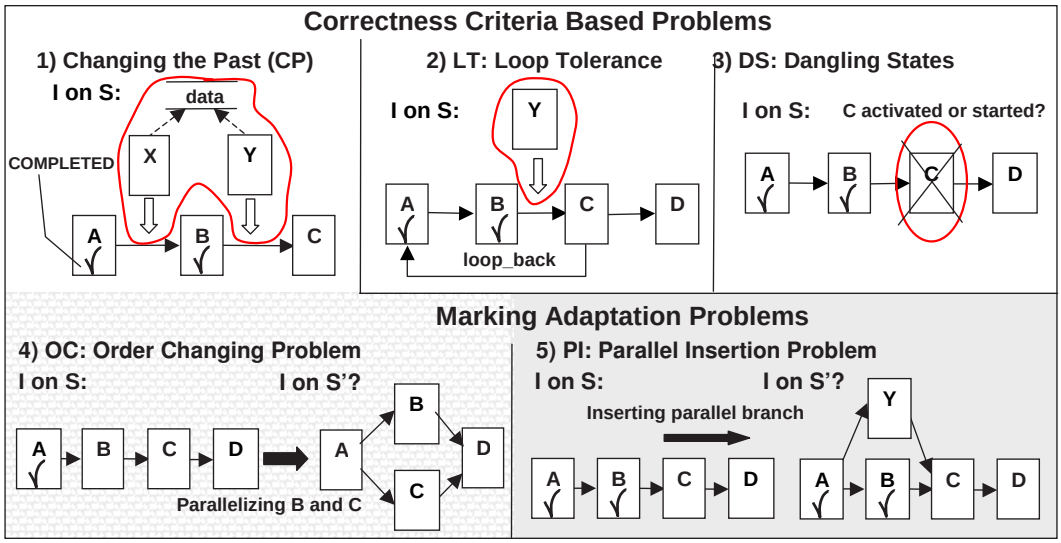


Figure 2.6: Five Typical Problems Regarding Dynamic Process Change

2.4.2.1 Approaches Not Requiring Instance Execution Information

WF Nets: Informally, the core idea of the approach presented in [118] is as follows: An instance I on schema S (represented by a marked WF Net) is compliant with the modified schema $S' := S + \Delta$, if S can be mapped to S' (or vice versa) after applying special operators to S, S', or both (graph equivalence). One example is depicted in Figure 2.7: Process schema S can be mapped to process schema S' if we *hide* newly inserted activity X in S'.

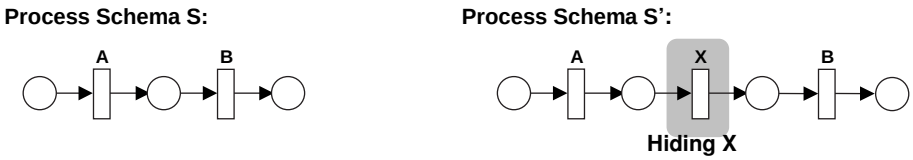


Figure 2.7: Graph Equivalence When Hiding Activities

By contrast, for the process schema change depicted in Figure 2.8 no mapping between S and S' can be found, neither by hiding nor by blocking activities. Accordingly, for process instance I no corresponding marking on changed process schema S' can be found. This phenomenon arising in conjunction with order–changing operations is called the dynamic change bug.

More precisely, an instance I on schema S (represented by a marked WF Net) is compliant with the modified schema $S' := S + \Delta$, if S and S' are related to each other under given inheritance constraints; i.e., either S is a subclass of S' or vice versa. In this context, the

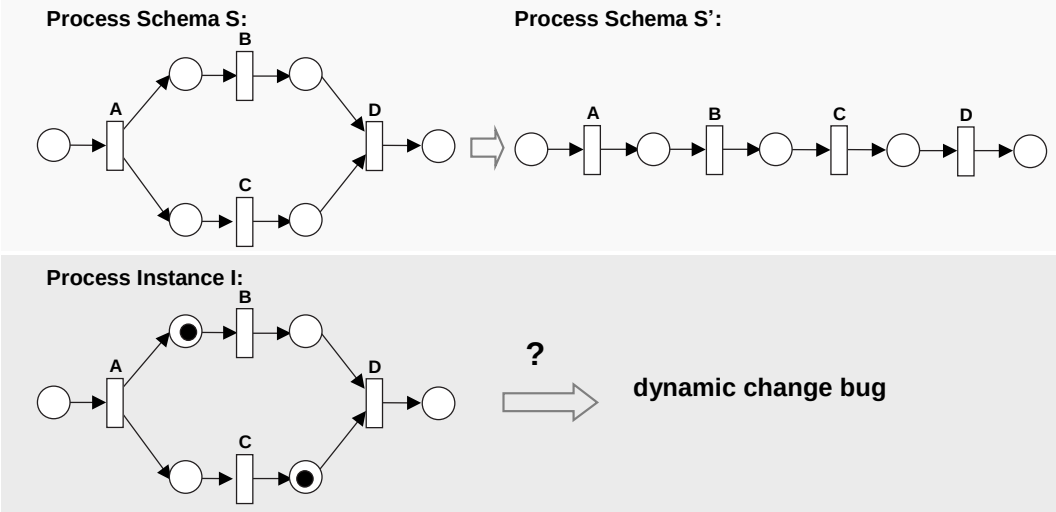


Figure 2.8: Dynamic Change Bug

following two kinds of basic inheritance constraints are provided [118]: A schema S is a subclass of another schema S' if one cannot distinguish the behaviors of S and S' (1) either when only executing tasks of S which are also present in S' or (2) when arbitrary tasks of S are executed but only effects of those tasks are taken into account which are present in S' as well. Thus, inheritance constraint (1) works by *blocking* and inheritance constraint (2) works by *hiding* a subset of tasks of S. More precisely, *blocking* of tasks means that these tasks are not considered for execution. *Hiding* tasks implies that the tasks are renamed to the silent task τ. (A silent task τ has no visible effects and is used, for example, for structuring purposes.) One example is depicted in Fig. 2.10(1) where the newly inserted activities `X` and `Y` are hidden by labeling them to the silent task τ. In addition, further inheritance constraints can be achieved by combining hiding and blocking of process tasks. Based on these inheritance constraints we can state the following correctness criterion:

Correctness Criterion 1 (Compliance Under Inheritance Relations) *Let S be a process schema which is correctly transformed into another process schema S'. Then instance I on S is compliant with S' if there is a mapping between S and S' under inheritance constraints (for a more formal definition see [118]).*

The challenging question is how to ensure Criterion 1. In [118] van der Aalst and Basten present an elegant way by providing special change operations which automatically preserve one of the presented inheritance relations between the original and the changed schema. These change operations comprise additive and subtractive changes or, more precisely, the insertion and deletion of cyclic structures, sequences, parallel and alternative branches. Let therefore again schema S be transformed into schema S' by change Δ. In order to check whether Δ is an

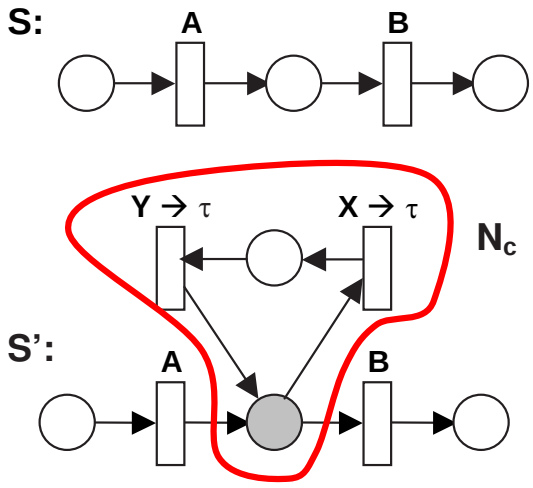


Figure 2.9: Inheritance Preserving Change

inheritance preserving change and therefore S and S' are related under inheritance (cf. Criterion 1) the authors define precise *conditions* with respect to S and S'.

As an example take the insertion of a cyclic structure N_c into S (resulting in S') where N_c and S have exactly one place in common (see Fig. 2.9). Then it can be ensured that S' is a subclass of S when hiding `X` and `Y` in N_c.

As it can be seen from Figure 2.10(1), using Criterion 1 it is possible to change already passed process regions (Changing the Past Problem). One problem in this context is to correctly adapt control flow tokens. By using anonymous tokens (i.e., excluding data tokens) the problem is simplified.

Using inheritance relations as described above restricts the set of applicable changes to additive and subtractive ones. More precisely, there is no adequate inheritance relation based on hiding or blocking activities when applying an order-changing operation. Consequently, the Order Changing Problem (cf. Figure 2.6) is factored out. Nevertheless, van der Aalst and Basten [118] offer an original and very important contribution by ensuring compliance for many practically relevant changes without need for accessing instance data.

After having decided whether an instance I on S is compliant with S' or not (cf. Criterion 1), we need rules to adapt the marking of I on S'. For this purpose, [118] provides *transfer rules* based on inheritance relations (cf. Definition 1). After inserting activities, cyclic structures or alternative branches, necessary marking adaptations are realized by directly mapping tokens of S onto S'. The insertion of parallel branches is more complicated since in some cases we have to insert additional tokens to avoid deadlocks. One example is given in Figure 2.10(5) where we add one token to an input place of the parallel join transition.

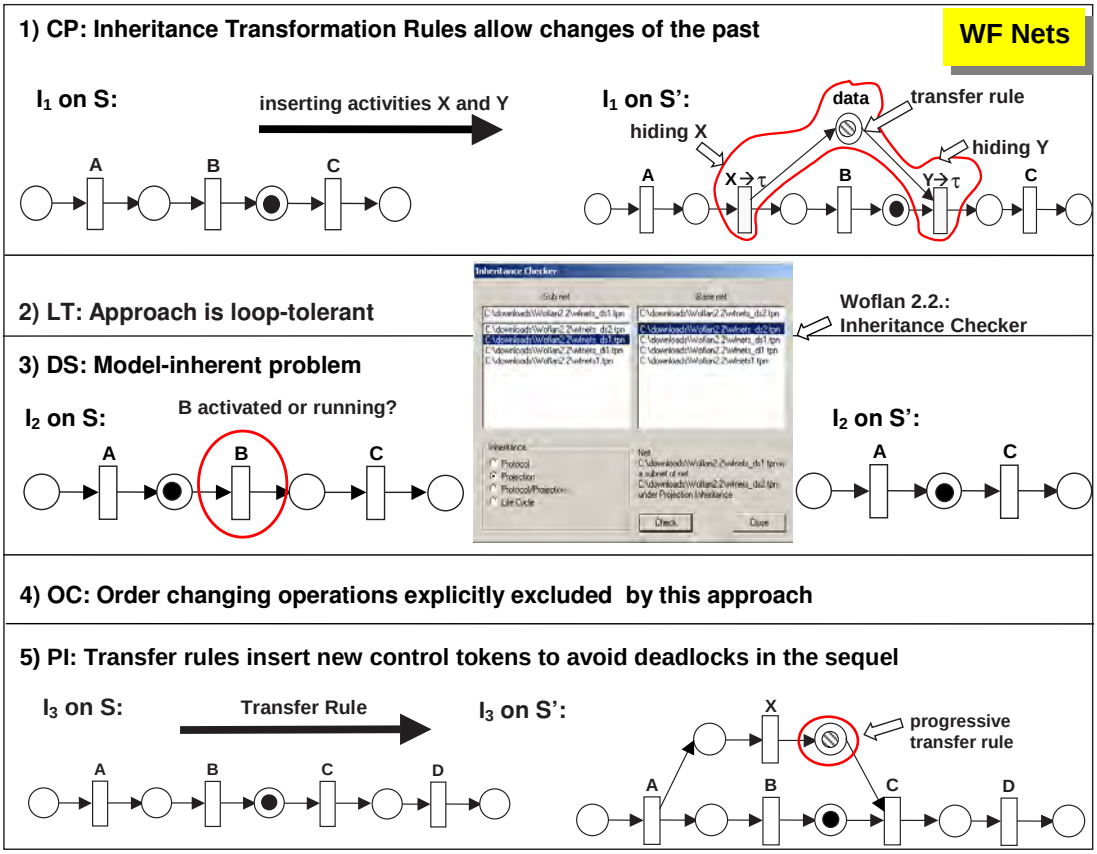


Figure 2.10: Correctness Checking and Marking Adaptations in [118, 127]

Taking our classification provided in Section 2.2 the approach presented by [118] can be estimated as follows:

1. **Completeness:** Data flow issues as well as order changing operations are not considered.
2. **Correctness:** A formal criterion based on graph equivalence is provided. Furthermore, for the allowed changes, there are automatic rules for adapting markings when migrating process instances to the changed process type schema.
3. **Change Realization:** Checking inheritance of arbitrary process schemes is PSPACE-complete [118]. However, a diagnosis tool called *Woflan* has been developed [118, 128, 129] to automatically decide inheritance rules for two given schemes.
4. **Interplay Between Process Type and Process Instance Changes:** This aspect is not taken into account.

2.4.2.2 Approaches Using Complete Execution Information

WASA$_2$ Activity Nets: WASA$_2$ [127, 135, 136] also uses graph equivalence to state formal correctness for dynamic changes. As opposed to WF Nets WASA$_2$ additionally takes instance information into account. More precisely, the execution state of an instance is described by its *purged instance graph* which is derived from original schema S by deleting all activities which have not been started yet and by removing all associated control and data edges.

Formally, a mapping $m : V_I \mapsto V_{S'}$ between process instance $I = (V_I, C_I, D_I)$ and process schema $S' = (V_{S'}, C_{S'}, D_{S'})$ assigns to every instance node $n \in V_I$ a unique schema node $m(n) \in V_{S'}$. With this, the following correctness criterion based on *valid mappings* between instance and schema graph can be stated:

Correctness Criterion 2 (Valid Mapping) *Let $I = (V_I, C_I, D_I)$ be a purged process instance graph derived from process schema $S = (V_S, C_S, D_S)$. Let further Δ be a change which correctly transforms S into another process schema $S' = (V_{S'}, C_{S'}, D_{S'})$. Then: I is compliant with S' iff a valid mapping $m: V_I \mapsto V'_S$ exists; i.e.,*
$(\forall\ i', j' \in V_{S'}$ with $\exists\ (i', j') \in C_{S'}\ \exists\ i, j \in V_I$: $i' = m(i)$, $j' = m\ (j) \wedge\ (i, j) \in C_I)$ and vice versa $\wedge$
$(\forall\ k', l' \in V_{S'}$ with $\exists\ (k', l') \in D_{S'}\ \exists\ k, l \in V_I$: $k' = m(k)$, $l' = m\ (l) \wedge\ (k, l) \in D_I)$ and vice versa

Intuitively, a process instance I can be migrated to a changed process schema S' if each completed activity of I is also contained in S' and all control and data dependencies existing in I have counterparts in S' (cf. Figure 2.11(5)). Criterion 2 can be paraphrased using the notion of *schema prefixes* [135] which leads to Criterion 3.

Correctness Criterion 3 (Schema Prefix) *Let $I = (V_I, C_I, D_I)$ be a purged process instance graph derived from process schema $S = (V_S, C_S, D_S)$. Let further Δ be a change which transforms S into another process schema $S' = (V_{S'}, C_{S'}, D_{S'})$. Then: I is compliant with S' iff I is a prefix of S', i.e.,*
$V_I \subseteq V_{S'}$, $C_I \subseteq C_{S'}$, $D_I \subseteq D_{S'}$ and $\forall$ (p,q) $\in (C_{S'} - C_I) \cup (D_{S'} - D_I)$: $q \notin V_I$.

Instance graph I_4 from Figure 2.11(5) is a prefix of S' but I_3 in Figure 2.11(4) is not. From Figure 2.11(1) we can see that Criteria 2 + 3 prohibit changes of already passed graph regions. Thus, correct data provision of activities and consistent instance states are guaranteed. Since WASA$_2$ Activitiy Nets are acyclic (cf. Section 2.3.2) the Loop Tolerance Problem (cf. Figure 2.6) is not present. Details about how Criteria 2 + 3 can be checked and instances be adapted to the changed schema have not been available. However, a powerful prototype exists. Interestingly, for some cases Criteria 2 + 3 are too restrictive regarding the Order Changing Problem (cf. Figure 2.6). An example is depicted in Figure 2.11(4) where I_3 could be smoothly migrated to S' but no valid mapping between I_3 and S' exists.

Finally, we classify this approach according to the challenges introduced in Section 2.2:

1. **Completeness:** WASA$_2$ Activity Nets are to be acyclic, i.e., loop structures are not allowed.

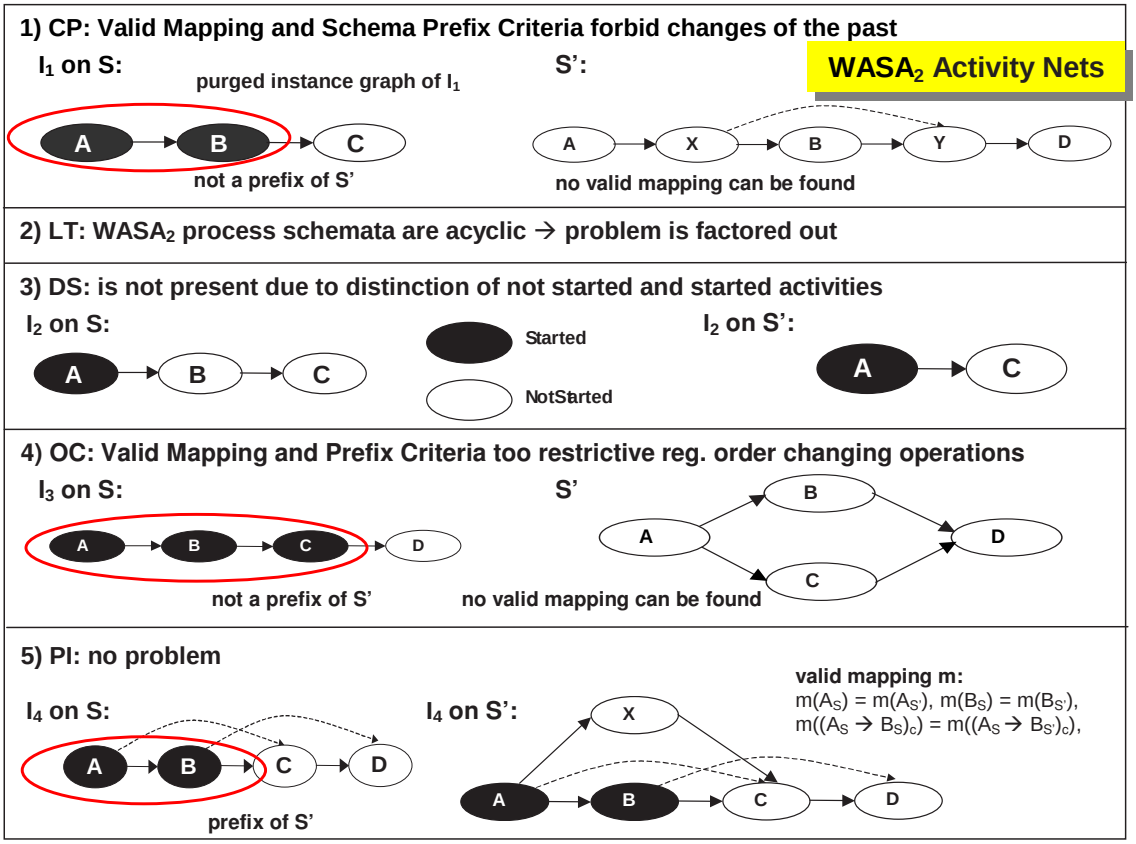


Figure 2.11: Criteria 2 and 3 [135, 136] Applied to Typical Change Problems

2. **Correctness:** A formal criterion based on graph equivalence is provided. However, this criterion is too restrictive in conjunction with order–changing operations.

3. **Change Realization:** It is not clear whether the correctness criterion is actually checked by graph mappings or if there are more efficient methods. Furthermore, no methods for marking adaptations after process instance migrations are provided.

4. **Interplay Between Process Type and Process Instance Changes:** This approach allows both, process type and process instance changes within one system. However, it is not possible to propagate process type changes to already modified (biased) process instances.

MILANO Nets [1, 2]: Only a special class of schema transformations is considered, namely parallelization, sequentialization, and swapping of activities [1]. In doing so, special constraints (summarized by the *Minimal Critical Specification (MCS)*) are obeyed for the underlying Sequential Model (cf. Sect. 2.3). For these restricted changes the following correctness criterion

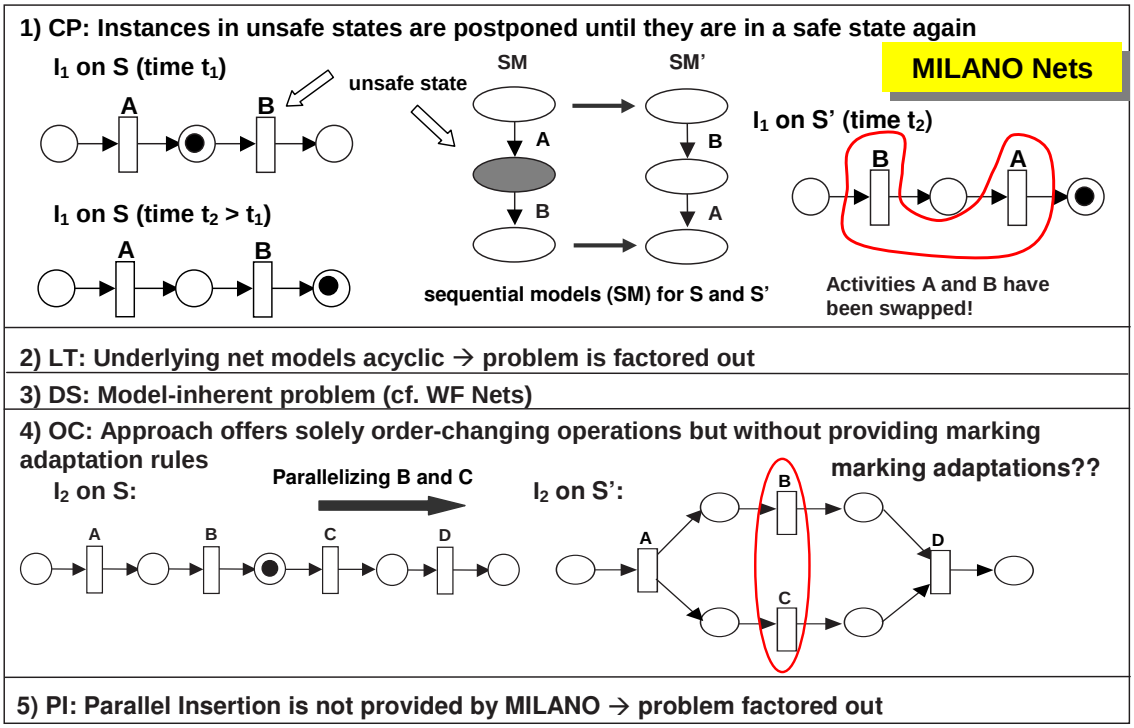


Figure 2.12: Typical Change Problems in MILANO [1]

is provided:

Correctness Criterion 4 (Safe States) *Let S be a process schema and I an instance on S. Let further Δ be a change which transforms S into another correct process schema S'. Then: I is compliant with S' if I is not in an unsafe state on S regarding S'. A state of S is unsafe regarding S' if this state is not present in S'.*

Potential states of S and S' can be determined by constructing their Sequential Models (reachability graphs). An example for an instance with unsafe state is depicted in Figure 2.12(1). For such cases MILANO postpones instance migration until the instance will be in a safe state again. Doing so cultivates the *Changing the Past Problem* (cf. Figure 2.6). The *Loop Tolerance* and the *Parallel Insertion Problems* cannot be evaluated since the underlying process schemes are acyclic and parallel insertion is not supported (cf. Section 2.3.1). Parallelization of activities is always allowed, but no details are given how to adapt instance markings in this context (cf. Figure 2.12(5)).

Taking our classification provided in Section 2.2 the MILANO approach can be estimated as follows:

1. **Completeness:** The expressiveness of this approach is strongly restricted by claiming acyclic process schemes and neglecting data flow issues. Furthermore, only certain order changing operations are allowed.
2. **Correctness:** A formal correctness criterion is provided.
3. **Change Realization:** This approach neither gives any idea how to ensure its correctness criterion nor how to actually adapt markings after migrating process instances to a changed process schema.
4. **Interplay Between Process Type and Process Instance Changes:** This aspect is not considered.

2.4.3 Approaches Based on Trace Equivalence

In this section we discuss approaches which base dynamic change correctness on *trace equivalence* (cf. Figure 2.4).

2.4.3.1 Predictive Approaches

Flow Nets offer a first approach based on trace equivalence [38, 39]. In [39], process instance changes on S are carried out by substituting the marked sub-net N_1 of S, which is affected by Δ, by another marked sub-net N_2, which reflects the modifications set out by Δ. Thereby, N_1 is referred to as the *old* and N_2 as the *new change region* (cf. Figure 2.13(4)). As the authors point out, the selection of the change regions cannot be fixed. Roughly, the old change region is defined as the smallest marked sub-net containing all activities affected by Δ.

For the following considerations, please remember that $\bar{i}$ denotes the initial and $\bar{f}$ the final marking of S (cf. Section 2.3.1). Furthermore, we formally define the *FiringSequenceSet (FSS)* of a schema S as follows: Let m and m' be two markings on S. Then $FSS(S, m, m')$ is the set of all possible firing sequences leading from m to m' on S.

Correctness Criterion 5 (Pre-Change Firing Sequence) *Let I be an instance on process schema S with marking m and let* $\omega \in FSS(S, \bar{i}, m)$. *Let further* Δ *be a change which transforms S into another correct process schema S' and let m' be the resulting marking of I on S'. Then I is compliant with S' iff*
- $FSS(S, m, \bar{f}) \neq \emptyset \Longrightarrow FSS(S', m', \bar{f}) \neq \emptyset$
- $\forall \omega' \in FSS(S', m', \bar{f}) \Longrightarrow (\omega' \in FSS(S, m, \bar{f}) \vee \omega\omega' \in FSS(S', \bar{i}, \bar{f}))$

Criterion 5 presupposes that the marking m' resulting from the migration of instance I to the changed schema S' is known. Then starting from m' it has to be verified that all firing sequences ω' leading from m' to the terminal marking on S' are either producible on S starting from m as well (cf. Figure 2.13(1)) or firing sequence ω leading to m on S can be continued

on S' by ω' (cf. Figure 2.13(2)). Criterion 5 is very interesting in the context of the Changing the Past Problem (cf. Figure 2.6): On the one hand it allows "pure" changes of the past (cf. Figure 2.13(1),i). On the other hand, it forbids changes which affect both already passed regions and regions which will be entered in the sequel (cf. Figure 2.13(1),ii). Whether Criterion 5 is loop tolerant or not depends on the definition of the pre-change firing sequence ω (cf. Figure 2.13(2)).

Regarding the Order Changing Problem (cf. Figure 2.6) the authors present two kinds of change operations and a special change class, the *Synthetic Cut-Over Change (SCOC)*[2]. Applying SCOC, the old change region N_1 is maintained in S' together with N_2 (for an example see Figure 2.13(4)); i.e., S' contains two versions of the modified subnet. How this "fusion" of old and new change region is carried out depends on the applied change. In [39] two change scenarios – *Upsizing* and *Downsizing* – are introduced. Upsizing means that N_2 can "do more" than N_1, i.e., the set of all valid firing sequences on N_1 is a subset of all valid firing sequences on N_2. Downsizing is the dual counterpart of upsizing, i.e., N_2 can "do less" than N_1. For example, Figure 2.13(4) shows an upsizing. In this case, the SCOC can be constructed by sticking N_1 and N_2 together over *flow-jumpers* (cf. Figure 2.13(4)). Flow-jumpers are transitions, which map each marking of N_1 to a marking of N_2 [38]. This way of constructing the SCOC in conjunction with upsizing operations is correct regarding Criterion 5. In the other case – downsizing – the SCOC is constructed by merging N_1 and N_2 over one output place, i.e., instances with tokens in N_1 are further executed according to the old net.

We can summarize the above discussion as follows:

1. **Completeness:** Regarding the expressiveness of Flow Nets and the offered change operations there is no restriction.
2. **Correctness:** A formal criterion is provided.
3. **Change Realization:** The authors do not explain how to check the correctness criterion. For special change operations the approach offers the SCOC for adapting process instance markings when migrating to the changed process schema. However, due to the complexity of this method it cannot be used in practice. There are no marking adaptation methods for all other change operations.
4. **Interplay Between Process Type and Process Instance Changes:** This topic is not taken into account.

2.4.3.2 Using Complete History Information

WIDE Graphs: A widely-used correctness property is the *compliance criterion* introduced by [26]. Intuitively, change Δ of schema S can be correctly propagated to process instance I on S

[2]The method of constructing SCOC is also applied in the area of reconfigurable high–level Petri Nets [6]. Here, different variants of a process a pre–modeled by using the SCOC method

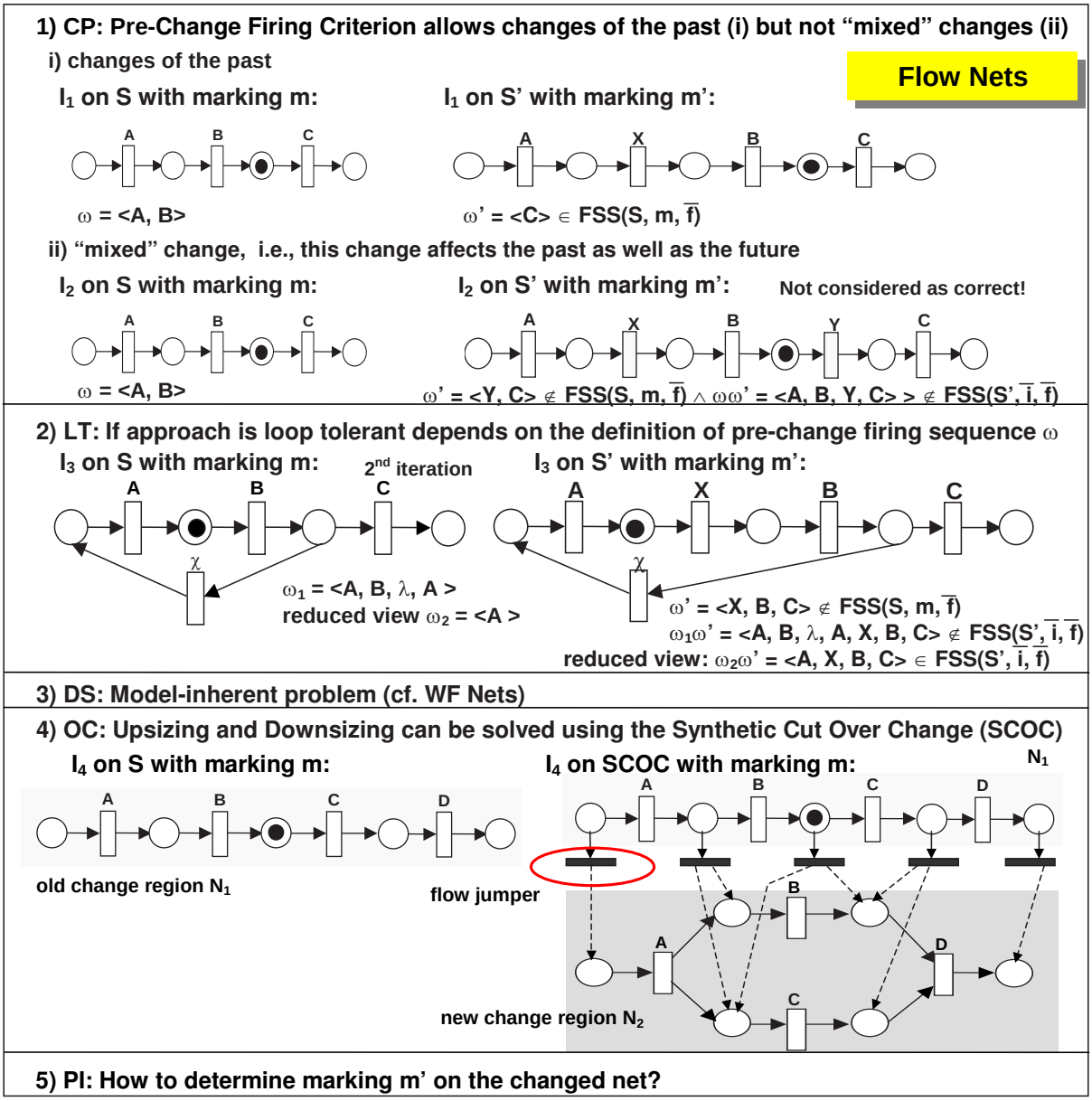


Figure 2.13: Pre-Change Criterion and SCOC [39] Applied to Typical Change Problems

if and only if the execution of I, taken place so far, can be "simulated" on the modified schema S' as well. Since WIDE works with a history-based execution model, compliance is based on *trying to replay the execution history* Π_I^S of instance I on the changed schema S'. Formally:

Correctness Criterion 6 (Compliance Criterion) *Let S be a process schema and I be a process instance on S with execution history* Π_I^S*. Let further S be transformed into another schema S' by change operation* Δ*. Then I is compliant with S' if* Π_I^S *can be replayed on* $S = S + \Delta$ *as well, i.e., all events stored in* Π_I^S *could also have been logged by an instance on S' in the same order as set out by* Π_I^S*.*

Criterion 6 forbids changes of the past (cf. Figure 2.14(1)). However, using execution histories as defined in WIDE (cf. Section 2.3.2) is too restrictive in conjunction with loops, i.e., it is not loop tolerant as can be seen from Figure 2.14(2). Obviously, Π_{I2}^S cannot be produced on S'. Therefore, I_2 is excluded from migration to S' though there would be no problems when proceeding execution of I_2 based on S'. Since [26] gives no information about how to check Criterion 6, we assume that compliance is ensured by trying to replay the whole execution history on the changed schema. Thus, we get the necessary marking adaptations automatically when checking compliance without additional effort. However, doing so causes an overhead due to the possibly extensive volume of history data which is normally not kept in main memory [67, 82, 121].

Taking our classification provided in Section 2.2 the WIDE approach can be estimated as follows:

1. **Completeness:** In WIDE, there are no restrictions regarding the expressiveness of the meta model and the offered change operations.
2. **Correctness:** WIDE introduces compliance as correctness criterion. Based on the particular representation of execution histories in WIDE this criterion is too restrictive in conjunction with loops. Furthermore, there might be problems with dangling states since execution histories in WIDE only contain `End` entries (representing activity completions).
3. **Change Realization:** Checking compliance and adapting process instance markings are realized by replaying execution histories on the changed process schema. The complexity is $O(n * m)$ where n corresponds to the number of activities of the respective process schema and m to the number of process instances.
4. **Interplay Between Process Type and Process Instance Changes:** Since WIDE does not support changes of single process instances no solutions for the interplay of process type and process instance changes are provided.

Similarily, several other approaches [104, 115] exist which propose correctness criteria based on instance execution information. In [115], firstly, the region of the process schema is detected which is affected by the change (*change region*). Then for each process instance it is checked

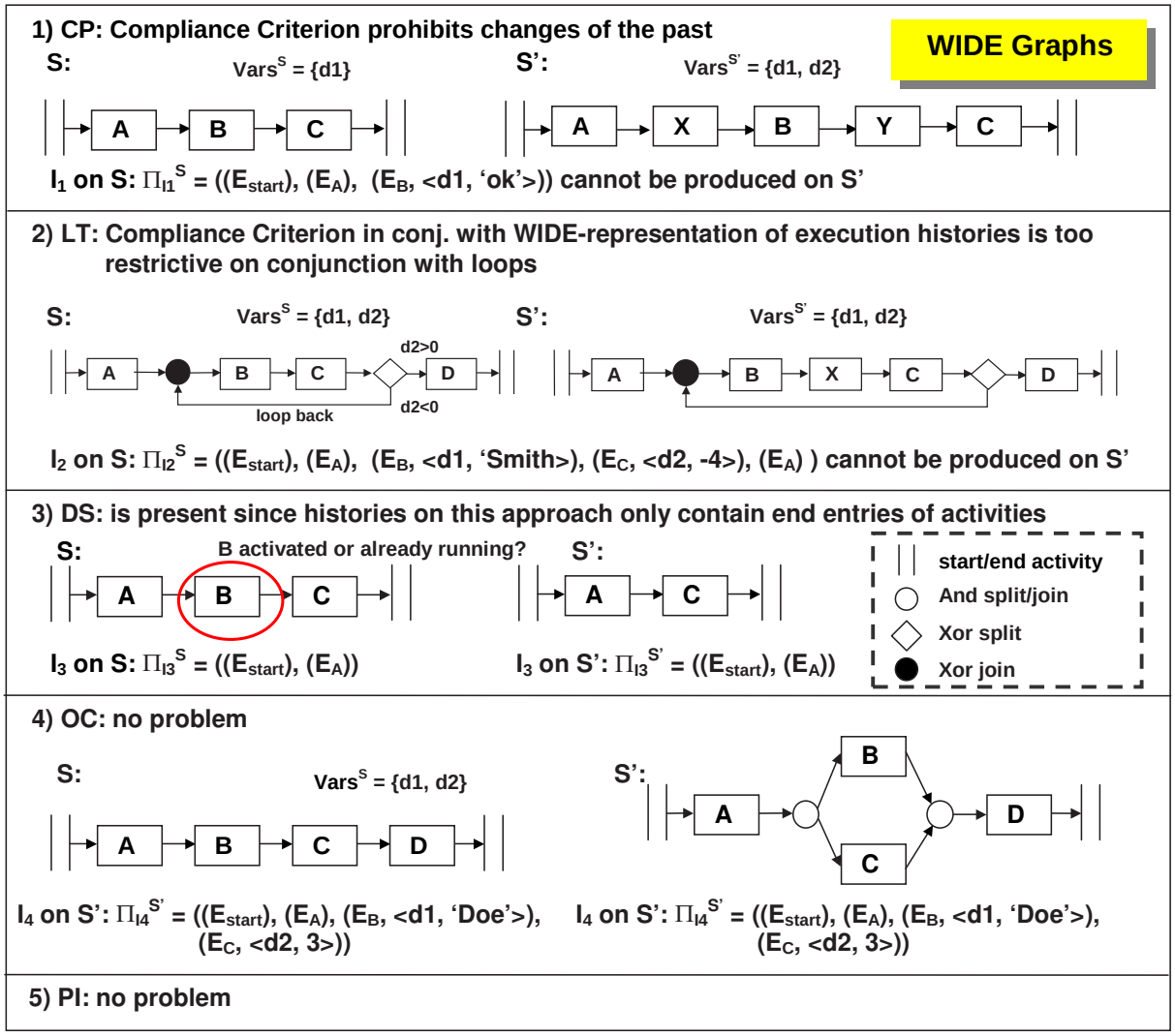


Figure 2.14: Checking Compliance by Replaying the Complete Execution History [26]

Table 2.1: Examples of Migration Conditions in TRAMs(cf. [67])

Change Δ	**Migration Condition for Instance I**
Insertion of Activity A	none
Modifying start condition sc_A of Activity A	$(\mathtt{S}_A \notin \Pi_I^S) \vee (\mathtt{S}_A \in \Pi_I^S \wedge sc_A \text{ holds})$
Deletion of Activity A	$\mathtt{S}_A \notin \Pi_I^S$
Insertion of read (write) data edge for Act. A	$\mathtt{S}_A \notin \Pi_I^S$ ($\mathtt{E}_A \notin \Pi_I^S$)

Underline: Π_I^S denotes execution history of I on S; S_A/E_A: start/end event of activity A

whether a process instance has already reached the change region or not. If not the respective instance is estimated as being compliant. The complexity for determining the change region is $O(n^4 * (n!)^2)$ (where n denotes the number of activities contained in the respective process schema). However, the author does not give any hint how to concretely migrate compliant process instances, i.e., methods for adapting process instances are missing.

2.4.3.3 Using a Consolidated View on the Execution History

TRAMs [66, 67] uses Criterion 6 as well. However, replaying each history entry of an instance on the changed schema is considered as too inefficient, especially when a large number of instances has to be migrated. Therefore, TRAMs provides migration conditions based on which compliance of a process instance with the changed schema can be checked more efficiently (cf. Table 2.1).

Due to the declarative control flow definition and the absence of explicit control edges, the insertion of activities is a complex change; i.e., first a new activity node A is inserted and then it is embedded into the control flow by setting the start conditions of A ("incoming edges") and the intended successors ("outgoing edges"). Figure 2.15(1) depicts the aggregated migration conditions for the insertion of two activities and a data dependency between them. Figure 2.15(4) shows the parallelization of activities. To our best knowledge TRAMs Graphs are acyclic and consequently the Loop Tolerance Problem cannot be decided on.

The discussion of TRAMs can be summed up as follows:

1. **Completeness:** It is not clear whether TRAMs supports loop constructs or not.
2. **Correctness:** TRAMs uses the compliance criterion as introduced in WIDE. Due to the representation of execution histories in TRAMs the use of the compliance criterion would be also too restrictive in conjunction with loops.
3. **Change Realization:** TRAMs states explicit compliance conditions based on which the compliance criterion can be ensured. Since these conditions are based on history scans the complexity is $O(n * m)$ as in WIDE (where n corresponds to the number of activities of

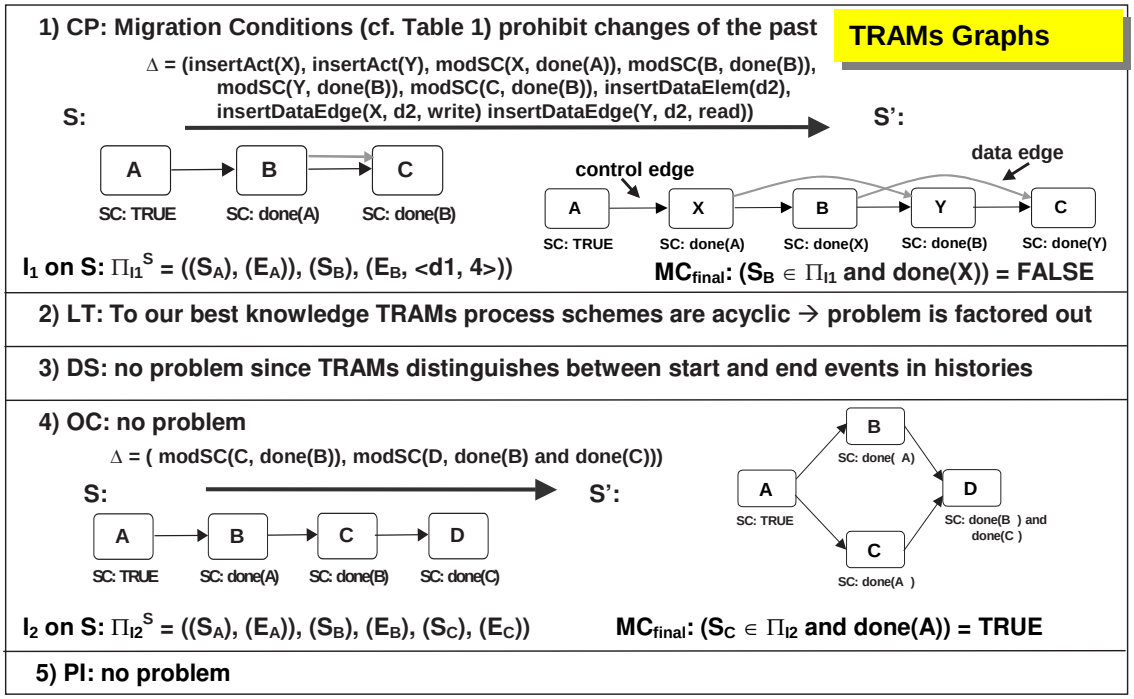


Figure 2.15: Migration Conditions of TRAMs

the process schema and m to the number of process instances). How markings are adapted after migrating process instances to the changed process schema is not explained.

4. **Interplay Between Process Type and Process Instance Changes:** TRAMs does not address changes of single process instances. Therefore there are no results regarding the interplay between process type and process instance changes.

2.4.4 Other Approaches

The Petri Net based approaches discussed so far have in common that they use anonymous tokens which represent the control flow. In the Petri Net literature, there are many suggestions on using tokens which are enriched with further information, e.g., about time constraints [83, 114]. One of these extended Petri Net models are Funsoft Nets [34] which use typed tokens to represent control as well as data flow. However, the inherent problems of models with *True Semantics* are still present for such high–level Petri Nets.

A rule–based approach is offered by ULTRAflow [41]. The authors focus on modification of the implementation and meta data of process activities. To handle instance access on changed specifications in a consistent manner special synchronization methods have been developed. To

reduce complexity several important change operation types like, for example, delete operations are factored out in ULTRAflow. Therefore this approach is interesting but due to the restricted set of change operations offered to users it is not suitable for practical applications.

MOKASSIN [53, 54] uses an object–oriented approach. Changes are realized by encapsulating change primitives within the process instances, i.e., the process instances themselves are responsible for ensuring correctness and consistency when carrying out changes. Furthermore, MOKASSIN tries to provide compliance by offering a fine–granular versioning concept. However the authors give no idea how compliance can be concretely ensured.

An approach based on state charts is presented in [43]. However, only process schema changes without considering their propagation on running process instances are analyzed.

Reflective approaches [9, 37] offer process languages based on which processes as well as meta processes describing the processes themselves can be defined. In SPADE [9], reflective process language SLANG allows to model meta processes like, for example, a meta process describing the different steps of a process change. If a process is changed (within a special editor) the change meta process is started. This is done by initializing the meta process with a special token which contains all relevant information for each process instance to be migrated. However, the authors give no idea how to adapt instance markings after their migration to the changed process schema.

2.5 Exterminating Dynamic Change Problems - A Comparison

As can be seen from Table 2.2 all presented approaches are based on formal correctness criteria. Obviously, there is a trade-off between complexity of the used process meta model and the flexibility offered by the system during runtime. The more powerful the process meta model is the more complex dynamic process changes are to handle. Agostini and De Michelis [1] have realized this in a very early stage and therefore vote to keep the process meta model as simple as possible in order to achieve a maximum of flexibility. For this reason, for example, loops cannot be modeled in MILANO, but must be handled dynamically (via backward jumps) if needed. Furthermore, MILANO limits adaptability to control flow changes, while data flow is managed at the level of single activities. Obviously, this simplifies the users's view on the process. However, in general, control flow changes cannot be treated in a isolated manner and independently from data flow and other process aspects.

Furthermore, Table 2.2 gives a comparison of the different approaches regarding compliance checking and marking adaptations. In [118], an elegant way for checking compliance as well as for automatically adapting instance markings is presented. $WASA_2$ [136] does not explicitly provide compliance checks. We assume that a valid mapping (cf. Definition 2) is determined by comparing nodes, control flow and data flow edges of the purged instance graph for each instance. Though both, replaying whole execution history (WIDE) and checking migration conditions (TRAMs) can be done with the same complexity, generally, there is a giant different in real effort. The reason is that one has to cope very likely with large amount of log data

Table 2.2: Comparison of Process Meta Models, Correctness Criteria and Marking Adaptation

	Expressiveness			**Completeness**	**Formal**	**Compliance**	**Marking**
	general	*loops*	*DF*	**of Changes**	**Criteria**	**Checks**	**Adaptations**
WF Nets	+	+	n.a.	−	+	✓	✓
$WASA_2$	+	n.a.	+	+	+	n.a.	n.a.
MILANO	∼	n.a.	n.a.	−	+	n.a.	n.a.
Flow Nets	+	+	+	+	+	n.a.	$O(e^n)$
WIDE	+	+	+	+	+	✓ $O(n)$	+
Breeze	0	n.a.	+	+	0	n.a.	n.a.
TRAMs	+	n.a.	+	+	0	✓ $O(n)$	n.a.

DF: data flow; n.a.: "not addressed"; ∼: "simplicity issues"; n: # activities in process schema

Table 2.3: Comparison by Means of 5 Typical Change Problems (cf. Figure 2.6)

	1) CP	**2) LT**	**3) DS**	**4) OC**	**5) PI**
WF Nets	possibly critical	+	possibly critical	+	+
$WASA_2$	prevented	−	prevented	0	+
MILANO	possibly critical	−	possibly critical	+	−
Flow Nets	possibly critical	?	possibly critical	+	+
WIDE	prevented	−	possibly critical	+	+
Breeze	prevented	−	?	+	+
TRAMs	prevented	−	prevented	+	+

+: "problem not present"; −−: problem present or factored out"; ?: "no statement possible"

[67] which is usually not kept in primary storage. However, by replaying the complete history information on the changed schema [26] we get the necessary instance markings for free.

Table 2.3 compares the different approaches from Figure 2.1 with respect to their ability to solve the *dynamic change problems* (cf. Figure 2.6).

1) Changing the Past Problem: From Figure 2.3 it can be seen that all Petri-Net based approaches with True-Semantics (WF Nets, MILANO Nets, and Flow Nets) allow changes of already passed regions of an instance. As mentioned in Section 2.4 doing so may cause two problems – incomplete input data when invoking activities and inconsistent instance execution. Such problems have been partially factored out in the presented approaches since data flow is not considered. As an example take instance I_1 on S' in Figure 2.10(1). Obviously, the newly inserted activity X will never be executed, i.e., the execution state of I_1 is not clearly defined. Assume that WF Nets do not exclude data flow and therefore activities X and Y can be inserted with the data dependency between them (see Figure 2.10(1)). This change would be considered as insertion of a parallel branch (*projection inheritance*) and a token be added to place `data`, but with unclear data semantics. Interestingly, this problem is excluded by Flow Nets since common changes of the past and the future are forbidden (cf. Figure 2.13(1),ii). Therefore, only the problem of inconsistent instance execution states remains.

2) Loop Tolerance Problem: Many of the presented approaches use acyclic process models whereby problem LT is factored out. The exclusion of loops, however, is out of touch with practical requirements. As discussed in Section 2.4, it depends on the exact definition of the pre-change firing sequence (cf. Criterion 5) whether Flow Nets are loop-tolerant or not. Anyway, applying Criterion 6 on basis of execution histories as offered in WIDE or TRAM is too restrictive in conjunction with loops.

3) Dangling States Problem: This problem of being unable to distinguish between activated and running activities is mainly present in Petri-Net based approaches. Transitions usually represent real-world tasks and consume a certain piece of time. If one of them is deleted the challenging question is how to handle in-progress work associated with this transition. One exception is WIDE where the special representation form of execution histories, more precisely the storing of `End` events, may lead to the dangling states problem as well. In contrast, approaches which explicitly differ between activity states `Activated` and `Running` like $WASA_2$ and TRAMs take care of this and ensure that running activities are not disturbed by dynamic changes.

4) Order-Changing Problem: This problem appears in conjunction with correctness criteria where certain process instances are needlessly excluded from migrating to the changed schema. As shown in Section 2.4.2.2, strict graph equivalence is too restrictive in certain cases.

5) Parallel-Insertion Problem: This problem refers to the necessary marking adaptations when inserting a parallel branch such that no deadlocks occur. The only Petri-Net based approach which presents concrete adaptation rules in this context is offered by WF Nets. The suggested strategies ensure a correct control flow in the sequel. However, with respect to data flow, semantics of the newly inserted tokens remains unclear.

2.6 Change Scenarios and Their Realization in Existing Approaches

In the previous sections emphasis has been put on fundamental correctness issues related to dynamic process changes. So far it has been circumstantial whether a single process instance or a collection of instances is subject to change. In this section we have a closer look at different change scenarios and related requirements. We provide a short categorization of adaptive research process engines which includes Chautauqua [40], $WASA_2$ [136], Breeze [104], and ADEPT [88]. In addition, we consider the respective approaches followed by AgentWork [80], EPOS [75], and DYNAMITE [49] as well as the flexibility support offered by commercial process management tools.

2.6.1 Changes of Single Process Instances

Adaptations of single process instances become necessary when exceptional situations occur or the structure of a process dynamically evolves. Both scenarios can be found, for example,

in hospital and engineering environments [49, 80]. Besides state-related correctness properties instance-specific changes pose several challenging issues. In particular, change *predictability* influences the way how process instances are adapted during runtime. Regarding evolving processes, for example, necessary changes and their scope are often known at buildtime [49, 75, 80]. Consequently, respective adaptations can be *pre-planned* and *automated*. In contrast *ad-hoc changes* have to be applied as response to unforeseen exceptions [88]. Usually, *user interaction* becomes necessary in order to define the respective runtime change. Of course, we cannot always see process instance changes in terms of black and white, but the distinction between pre-planned and ad-hoc change contributes to classify existing approaches.

2.6.1.1 Approaches Supporting Ad-hocInstance Changes

In Breeze and $WASA_2$, instance changes can be defined by the use of a graphical process editor. Using a process editor for change definition, however, is only conceivable for *expert users*. If changes shall be definable by *end users* as well, application-tailored user interfaces must be offered to them. Obviously, this requires comprehensive programming interfaces. Only few approaches provide such interfaces [88, 136]. ADEPT, for example, offers a change API which enables change definition on WSM Nets at a high semantic level, e.g., to jump forward in the flow or to insert a new step between two sets of activities [88]. Very important in this context is to ensure that none of the guarantees which have been achieved by formal checks at buildtime are violated due to the ad hoc change. Note that this does not only require compliance checks and marking adaptations, but also checks with impact to correctness properties of the process schema itself (e.g. regarding data flow). Therefore ADEPT uses well-defined correctness properties for process models, formal pre- and postconditions for change operations, and advanced change protocols [88].

2.6.1.2 Approaches Supporting Pre-Planned Instance Changes

Application–specific support for automatic process changes is provided by AgentWork [80, 81], DYNAMITE [49], and EPOS [75], but may be realizable on top of adaptive PMS like $WASA_2$, ADEPT, or InConcert as well. The overall aim is to reduce error-prone and costly manual process adaptations. In order to realize automatic process adaptations, firstly, the PMS must be able to detect logical failures in which process instance changes may become necessary. Secondly, it must determine necessary adaptations, identify the instances to be adapted, correctly introduce the change to them, and notify respective users. This poses many additional issues ranging from the consistent specification of pre-planned changes at buildtime up to their concrete realization during runtime. Existing approaches supporting automatic process instance changes can be classified according to different criteria. The most important one concerns the basic method used for automatic failure detection and for change realization. We distinguish between rule-based, process-driven, and goal-based approaches:

Rule-based approaches use ECA (Event/Condition/Action) models to automatically detect logical failures and to determine necessary process changes. However, most of them limit adaptations to currently executed activities [25, 27]. In contrast, AgentWork [80, 81] enables automatic adaptations of the yet unexecuted regions of running process instances as well. Basic to this is a temporal ECA rule model which allows to specify adaptations at an abstract level and independently of concrete process models. When an ECA rule fires, temporal estimates are used to determine which parts of the running process instance are affected by the detected exception. Respective process regions are either adapted immediately (predictive change) or - if this is not possible - at the time they are entered (reactive change).

Goal-based approaches formalize process goals (e.g., process outputs). In ACT [17], necessary instance adaptations (e.g., substituting the failed activity by an alternative one) are automatically performed if an activity failure leads to a goal violation. EPOS [75], in addition, automatically adapts process instances when process goals themselves change. Both approaches apply planning techniques (e.g., [18, 138]) to automatically "repair" processes in such cases. However, current planning methods do not provide complete solutions since important aspects (e.g., treatment of loops) are not considered.

Process-driven approaches restrict the possible variants of process schemes as well as process changes in advance. DYNAMITE, for example, uses graph grammars and graph reduction rules for this [49]. Automatic adaptations are performed depending on the outcomes of previous activity executions. Interestingly, process-driven as well as goal-based approaches have been primarily applied in the field of engineering workflows [19]. Both DYNAMITE and EPOS provide build-in functions to support dynamically evolving process instances. Another approach using graph grammars is Obligations [22]. Here a process instance graph consists of different, overlaying "sheets". Process instances can be changed by adding or removing sheets.

Instance-specific changes pose several other challenges. For example, one must decide on the **duration** of an instance change. Concerning loop-related adaptations, ADEPT differentiates between loop-permanent and loop-temporary changes [88]. The latter are only valid for the current loop iteration. The handling of such temporary changes is not trivial since permanent changes must not depend on them in order to avoid potential errors. AgentWork [80, 81] even follows a more advanced approach by allowing rule designers to specify temporal constraints indicating how long process adaptations shall be valid.

2.6.1.3 Ad-hoc Changes in Commercial Tools

Production PMS like WebSphere MQ Workflow [51] and Staffware [111] provide powerful process support functions but tend to be very inflexible [78]. Particularly, ad-hoc changes of running process instances are not supported. Unlike these PMS, engines such as TIBCO InConcert, SERprocess, and FileNet Business Process Manager allow on-the-fly adaptations of in-progress instances [42, 55, 109]. For example, users may dynamically insert or delete activities for a given

instance in such a way that the past of this instance cannot be changed.[3] Though these PMS provide high flexibility, they have failed to support end users in an appropriate way. Particularly, they do not adequately support them in defining changes and in dealing with potential side-effects resulting from them (e.g., missing input data of an activity due to the deletion of a preceding step). Since one cannot expect from the end user to cope with such problems, this increases the number of errors and therefore limits the practical usability of respective PMS.

Case handling systems like FLOWer (Pallas Athena) [119, 126] try to address flexibility issues from another viewpoint. Unlike traditional PMS, case handling provides a higher operational flexibility and aims at avoiding dynamic changes. More precisely, users are allowed to inspect, add or modify data elements before activities, which normally produce them, are started. Consequently, the decision about which activities can be executed next is based on the available data rather than on information about the activities executed so far. Since FLOWer allows to distinguish between optional and mandatory data elements, a broad spectrum of processes can be covered with this data-driven approach. FLOWer also enables the definiton of causual dependencies between activities. The question remains, whether this mixed view on processes (process-driven, data-driven) contributes to completely avoid dynamic changes. Furthermore, the case handling paradigm does not solve the problem of process schema evolution, i.e., the propagation of process type schema changes to running process instances and their correct migration afterwards.

2.6.1.4 Flexibility By Design

There are several approaches where the process type schema only constitutes a skeleton roughly describing the process. The actual process structure is then specified by the user (more or less assisted by the PMS) and kept in instance–specific schemes.

Sadiq et al use so called "pockets of flexibility" [76, 107]. These are containers which are plugged into certain places of the process type schema. The containers comprise several activities and constraints imposed on them. If a process instance execution reaches a place where a container has been plugged in the current actor has to model the process consisting of the provided activities and obeying the imposed constraints. This concept has been implemented within the lightweight workflow engine *Chameleon*. It can import workflow templates modeled by use of the workflow editor *FlowMake*. Though this concept is interesting it is too restrictive regarding the possible change operations (i.e., it is only possible to add activities).

Another approach motivated by a practical application, i.e, the container transportation domain is presented in [11]. The process type schemes (representing basic tours) are only modeled as skeletons. If a process instance should be created firstly an optimal tour is figured out by applying certain algorithms of the operational research area (within the optimization engine). This optimal route is then fed into the system by ad-hoc changing the pre–modeled skeleton respectively.

[3] In order to avoid undesired side-effects on other cases, for each instance a private schema is kept.

2.6.2 Process Type Changes and Change Propagation

Process schema changes at the type level may become necessary, for example, to adapt business processes to a new law or to realize process optimizations. In case of long running processes we are confronted with the problem of how to migrate a potentially large number of process instances I_1, ..., I_n running on the old schema S to the new schema S'. Basically, things seem to be the same as for dynamic changes of single process instances. However, in addition, we are confronted with the problem that the process type change may have to be propagated to process instances whose current execution schema $S_I := S + \Delta$ does not completely correspond to S (due to a previous instance change Δ). To exclude such instances from migrating to the new schema S', however, is out of touch with reality, particularly in case of long-running flows. Interestingly, none of the process engines supporting process type changes and change propagation has dealt with this problem so far. In $WASA_2$, for example, individually modified instances cannot be further adapted to later type changes. Chautauqua [40] even does not support changes of single instances at all, since instances of a particular type are always connected to the same Flow Net.

Usually, commercial PMS/WfMS do not allow change propagation to in-progress instances when a process schema is modified at the type level. Instead, simple versioning concepts are used to ensure that already running instances can be finished according to the old schema. One exception is offered by Staffware [111]. However, there are several critical aspects arising in this context. For example, running activities can be deleted without any user information. If the deleted activity is finished the returned results are lost. Furthermore, Staffware suffers from the Changing the Past Problem (cf. Section 2.4) which may lead to missing input data and activity program crashs at runtime. Finally, Staffware is by far too restrictive (e.g., insertions before activated tasks are forbidden).

2.7 Summary

In this chapter we have elaborated the requirements for a comprehensive support of process schema evolution – completeness, correctness, change realization, and interplay between process type and process instance changes. Along these criteria and along typical problems arising in conjunction with process changes we have discussed approaches from research as well as commercial process technology which provide support for adaptive processes. And what is missing?

Almost all approaches fail with respect to completeness. They either restrict the expressiveness of the used process meta model, e.g., by forbidding loop constructs or neglecting data flow issues, or they factor out certain change operations. As a consequence, there is no comprehensive support regarding process changes when applying these approaches. Therefore it is indispensable to use a comprehensive process meta model supporting, e.g., loop constructs and data flow issues and to avoid the restriction of the set of offered change operations.

The main emphasis has been put on correctness criteria since they provide the basis for any

adaptive PMS. However, many of the suggested correctness criteria are too restrictive. They exclude process instances from migration to the changed process schema although there would be no inconsistencies or errors in the sequel. The challenge is to find correctness criteria which are not too restrictive on the one hand and which on the other hand support a comprehensive process meta model; i.e., the correctness criteria have to work in conjunction with process constructs neglected so far (see above) as well as in conjunction with all kinds of change operations.

The biggest gap between the offered approaches and practice diverges in conjunction with the concrete change realization. More precisely, most approaches do not provide any methods to (efficiently) check their correctness criteria. Regarding automatic process instance adaptation after migration to the changed process type schema most approaches either lack any methods for doing so or the offered methods are not practicable at all. In summary, the user is burdened with deciding whether a process instance is compliant with the changed process schema or not. This has to be done for a multitude of running process instances by deciding on correctness criteria. Afterwards users have to manually adapt markings for a possibly large number of process instances. Therefore in this thesis we release users from this complex task by offering methods for automatically deciding on compliance of process instances and for adapting markings after migration of process instances to the changed process type schema. Furthermore we provide an implementation of the presented concepts what is missing in most approaches.

But even if all limitations discussed above are overcome an adaptive PMS will be only accepted in practice if it supports all kinds of process changes; i.e., a PMS has to allow ad hoc changes of single process instances as well as changes of process types. If this requirement is taken seriously it has to be ensured that both kind of changes – process type and process instance changes – harmonize when occuring in interplay. In particular, it has to be possible to propagate a process type change to already ad hoc modified process instances. None of the presented approaches has dealt so far with respective questions like the support of disjoint or overlapping changes on process type and process instance level. In this thesis, a framework for supporting the interplay between process type and process instance changes is developed.

Chapter 3

Background Information

In this chapter we formally define the fundamental aspects for our further considerations. Thereby, we use *Well–Structured Marking Nets* (*WSM Nets*) [87, 88] as language to describe process schemes. By doing so we benefit from the expressiveness of this meta model which allows to model e.g., sequences, parallel and alternative branchings, loops, and data flow. Furthermore it offers a complete set of change operations to transform a WSM Net S into another WSM Net S' [87]. Altogether, using WSM Nets we avoid to restrict users from the beginning as many other approaches do (cf. Section 2).

WSM Nets are attributed, serial–parallel graphs with additional synchronization links. They cover all relevant aspects regarding process modeling like, for example, control flow, data flow, and time. We show under which constraints WSM Nets can be considered as being correct, i.e., can serve as a basis for correct process execution. Based on WSM Nets process instances can be started and executed. The execution state of a process instance is described by well–defined instance markings and the associated execution histories. To add flexibility to our approach we provide a comprehensive set of change operations on WSM Nets. We augment these change operations with precise formal pre– and post–conditions which ensure the transformation between WSM Nets under preserving the correctness constraints.

The formal framework capturing WSM Nets, process instances, and the possibility for changes has been established in [87]. In [87], main focus was put on the development of a expressive and usable process meta model as well as on ad–hoc changes of single process instances. In this work, we extend this fundamental approach towards process schema evolution and its various challenges.

Though we use WSM Nets to clarify the presented results our approach can be easily transferred to other process meta models with *True/False* semantics as well. How this transfer turns out for Activity Nets [73], for example, has been presented in [91].

3.1 Process Meta Model

In this section, we formally define all relevant buildtime and runtime aspects of our process meta model.

3.1.1 Buildtime Aspects – Well-Structured Marking Nets

In this work we use WSM Nets as process meta model (as for example applied in the ADEPT project [88]) and the change operations based on them. For the following discussion we will use a set–based definition of WSM Nets which facilitates the definition of change operations on WSM Nets. However, a WSM Net can be also graphically described. An example of a WSM Net is depicted in Figure 3.1 together with a summary of the used symbols and their specific semantics.

The following definition of WSM Nets is restricted to control and data flow issues since this is sufficient for the following considerations and does not overwhelm the reader with unnecesseray details. For control flow modeling a block–structured process model is chosen. As opposed to process models with strict block structuring WSM Nets show some extensions (e.g., synchronization of parallel branches). Using WSM Nets for process modeling, data flow is established by defining global data elements and connecting them to activities via read and write data edges. A read (write) data edge thereby denotes a read (write) access of an activity on the respective data element.

Definition 1 (WSM Net) *A tuple S = (N, D, NT, CtrlE, SyncE, LoopE, DataE, DP, EC) is called a WSM Net if the following holds:*

- *N is a set (bag)*[1] *of* activities
- *D a set of process* data elements
- *NT:* $N \mapsto \{$`StartFlow, EndFlow, Activity, AndSplit, AndJoin, XOrSplit, XOrJoin, StartLoop, EndLoop`$\}$
  NT assigns to each node of the WSM Net a respective node type.
- *The set of* control edges $CtrlE \subset N \times N$ *is a precedence relation*
- *The set of* synchronization edges $SyncE \subset N \times N$ *is a precedence relation between activities of parallel branches*
- $LoopE \subset N \times N \times (LoopCond \cup \{Undefined\})$ *is a set of* loop backward edges *with associated loop condition*

[1] In this work, bags are defined as finite multi–sets of activities (comparable to the definition of bags in [118]).

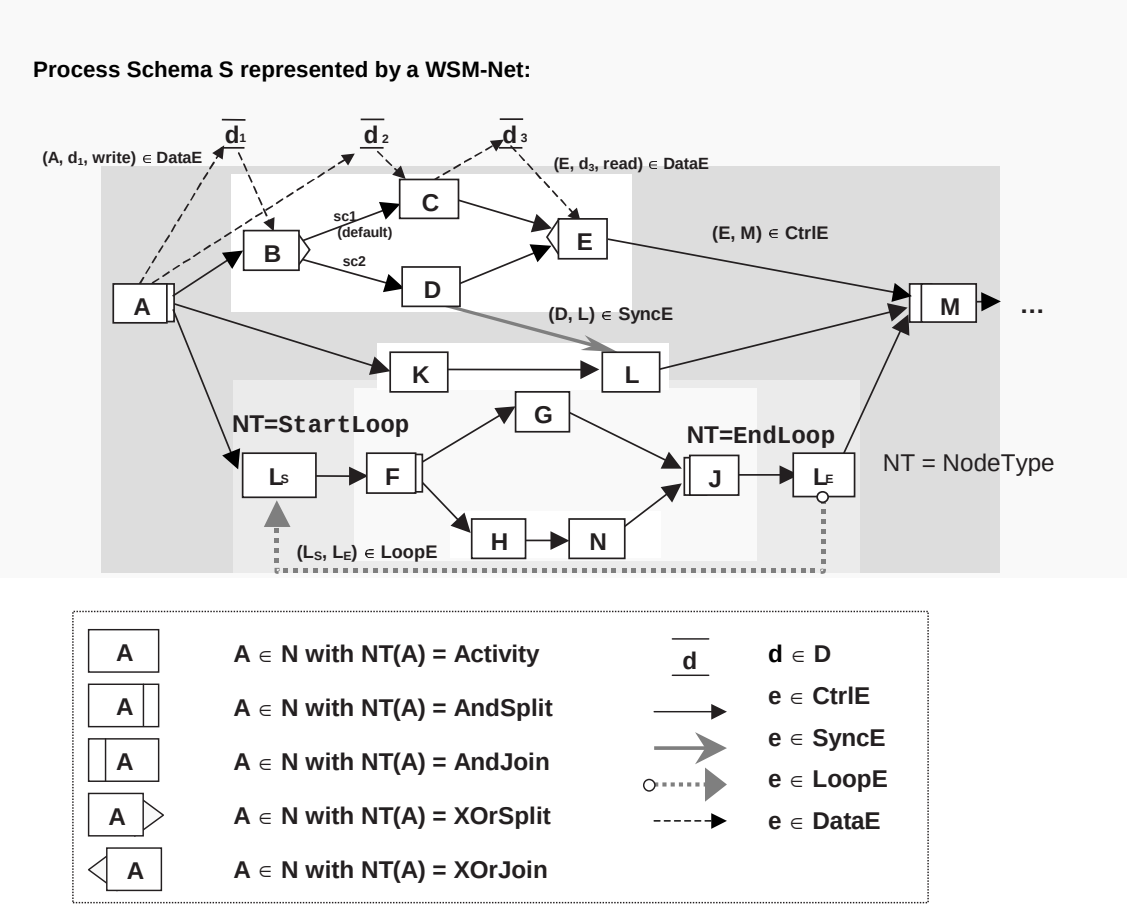


Figure 3.1: Process Schema Represented by a WSM Net (Abstract Example)

- $DataE \subseteq N \times D \times \{read,\ write\}$ *is a set of* read/write data links *between activities and data elements*
- $DP : N \mapsto N \cup \{Undefined\}$ *denotes the* decision parameter *which is associated to an alternative branching*
- $SC : CtrlE \mapsto SelCode \cup \{Undefined\}$ *denotes the* selection code *associated with a control edge.*

Thus, a process schema is represented by attributed, serial-parallel process graphs with additional synchronization links. As an example consider Figure 3.1. The depicted WSM Net S contains a parallel branching with `AndSplit` A and corresponding `AndJoin` M. This parallel branching contains an alternative branching with `XOrSplit` B and `XOrJoin` E as well as a loop construct with loop start node L_S and loop end node L_E. Furthermore, synchronization link

(D, L) sets out an order relation between activities D and L. There are three data elements d_1, d_2, and d_3. Activity A, for example, writes data element d_1 what is expressed by data write edge $(A, d_1, write)$ and activity C reads data element d_2 what is expressed by read data edge $(C, d_2, read)$.

Solely based on Definition 1 it is still possible to model WSM Nets which lead to an undesired behaviour at runtime like, for example, deadlock–causing cycles or missing input data. We explicitly forbid WSM Nets containing structural inconsistencies by imposing certain correctness constraints on them. A WSM Net S is *(structurally) correct* if the following constraints (1) – (7) hold [87, 88]:

Definition 2 (Correctess of a WSM Net) *Let S = (N, D, NT, CtrlE, SyncE, LoopE, DataE) be a WSM Net. Then S is a (structurally) correct WSM Net if and only if the following constraints (1) – (7) hold:*

(1) S has a unique start node Start (NT(Start) = `StartFlow`*) and a unique end node End (NT(End) =* `EndFlow`*).*

(2) Except for nodes Start and End each activity node of S has at least one incoming and one outgoing control edge e ∈ CtrlE.

(3) Let $S_{block} := (N, CtrlE, LoopE)$ be the subgraph of S which represents a projection on the activities, control edges, and loop edges of S. Then S_{block} has to be structured following a block concept, for which control blocks (sequences, branchings, loops) can be nested but must not overlap.

(4) Let S_{fwd} = (N, CtrlE, SyncE) be the subgraph of S which represents a projection on the activities, control edges, and sync edges of S. Then S_{fwd} has to be an acyclic graph, i.e., the use of control and sync edges must not lead to deadlock-causing cycles.

(5) Sync links must not cross the boundary of a loop block; i.e., an activity from a loop block must not be connected with an activity from outside the loop block via a sync link (and vice versa).

(6) For activities for which a mandatory input parameter is linked to a global data element $d \in D$ it has to be ensured that d will be always written by preceding activities at runtime independently of which execution path will be chosen.

(7) Parallel write accesses on data elements (and consequently lost updates on them) do not take place.

Obeying constraints (1) – (7) WSM Nets offer an expressive and usable way to define process schemes and to serve as adequate basis for process execution at runtime.

3.1.2 Runtime Aspects – Unbiased and Biased Process Instances

Taking a correct WSM Net S, new process instances can be created and started. *Logically*, each process instance I is associated with an instance-specific schema $S_I := S + \Delta_I$ (for unbiased instances $\Delta_I = \emptyset$ and consequently $S_I = S$ holds). An example of unbiased and biased process instances is depicted in Figure 3.3. Similar to firing rules in Petri Nets, the marking of a process instance is determined by well defined marking and execution rules (cf. Rules 1, Appendix B). Logically, for each process instance its own marking is captured by a marking function M^{S_I} $=(\mathrm{NS}^{S_I}, \mathrm{ES}^{S_I})$. It assigns to each activity n its current status $NS(n)$ and to each control, sync, and loop edge its marking $ES(e)$. These markings are determined according to well defined marking rules [87, 88], whereas markings of already passed regions and skipped branches are preserved (except loop backs). Concerning data elements, different versions of a data object may be stored, which is important for the context-dependent reading of data elements and the handling of (partial) rollback operations. Formally:

Definition 3 (Process Instance) *A process instance I is defined by a tuple $(S, \Delta_I, M^{S_I}, Val^{S_I}, \Pi_I^{S_I})$ where*

- *S = (N, D, NT, CtrlE, SyncE, ...) denotes the process schema I was derived from. We call S the original schema of I.*
- *Δ_I comprises instance-specific changes $op_1^I, \ldots, op_m^I$ that have been applied to I so far. We call Δ_I the bias of I. Schema $S_I := S + \Delta_I$ (with $S_I = (N_I, D_I, NT, CtrlE_I, SyncE_I, \ldots)$), which results from the application of Δ_I to S, is called the instance–specific schema of I.*
- *If $\Delta_I = \emptyset$ and consequently $S_I := S$ holds then we call I an unbiased process instance.*
- *If, in contrast, Δ_I actually comprises instance–specific changes (i.e., $\Delta_T \neq \emptyset$) then we call I a biased process instance.*
- *$M^{S_I} = (NS^{S_I}, ES^{S_I})$ describes activity and edge markings of I:*
  NS^{S_I}: $N_I \mapsto$ {`NotActivated, Activated, Running, Completed, Skipped`}
  ES^{S_I}: $(CtrlE_I \cup SyncE_I \cup LoopE_I) \mapsto$ {`NotSignaled, TrueSignaled, FalseSignaled`}[2]
  (Instance marking $M^{S_I} = (NS^{S_I}, ES^{S_I})$ is determined using the ADEPT marking and execution rules as defined in Rules 1).
- *Val^{S_I} is a function on D_I. It reflects for each data element $d \in D_I$ either its current value or the value* `UNDEFINED` *(if d has not been written yet).*
- *$\Pi_I^{S_I} = < e_0, \ldots, e_k >$ is the execution history of I. $e_0, \ldots, e_k$ denote the start and end events of activity executions. For each started activity X the values of data elements read by X and for each completed activity Y the values of data elements written by Y are logged. Formally:*

[2]The particular activity and edge states are explained in the following.

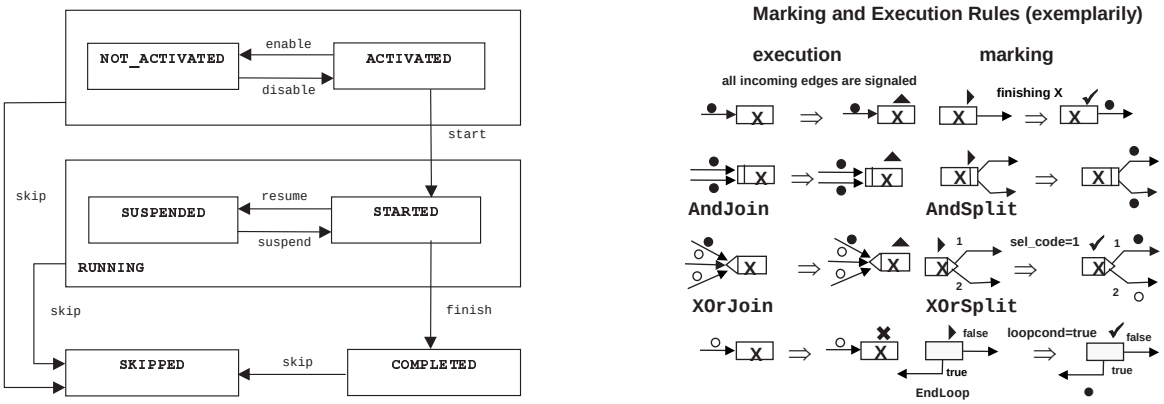


Figure 3.2: State Transitions and Marking/Execution Rules

$e_i \in \{\texttt{Start}^{(d_1^i,v_1^i),\ldots,(d_n^i,v_n^i)}\texttt{(<activity>, <It>)}, \texttt{End}^{(d_1^i,v_1^i),\ldots,(d_m^i,v_m^i)}\texttt{(<activity>, <sc|lc>)}\}$
Tuple (d_i^μ, v_i^μ) *thereby describes a read or write access of* e_μ *on data element* d_i^μ *with associated value* v_i^μ.

Figure 3.2 illustrates the possible state transitions of an activity. For each activity, its status is initially set to `NotActivated`. It is changed to `Activated` when all preconditions for its execution are met. If this is the case, the activity is becomes an executable task and inserted as a work item into user worklists. When selecting this work item for execution its activity status changes to `Running`. The corresponding work items are then removed from other user worklists and an application component associated with this activity is started. At successful termination of this component execution, activity status passes to `Completed`. Otherwise, if the scheduler recognizes that this activity cannot be selected for execution any longer, its status will change to `Skipped` (e.g., activity D of instance I_1 in Fig. 3.3b).

Edges are initially marked with `NotSignaled`. During process execution their status either changes to `TrueSignaled` or `FalseSignaled`. Finally, if a loop condition evaluates to true, the marking of the corresponding edge ($\in$ `LoopE`) is changed to `TrueSignaled` (cf. Fig. 3.2) and the markings of all activities/edges of the loop body are reset to their initial state. Otherwise the loop is left whereas the actual markings of the loop body remain. An overview about the particular state transitions and a sketch of the marking and execution rules (cf. Rules 1, Appendix B) are given in Figure 3.2.

Figure 3.3.b depicts unbiased process instance I_1 to which no instance–specific bias has been applied so far. In contrast, Figure 3.3.d shows biased process instance I_2 for which instance–specific bias Δ_{I_2} has deleted activities H and N.

Given a process instance I with particular instance marking M^{S_I} it is important for our further considerations to define whether this marking is a correct instance marking on instance–specific schema S_I or not.

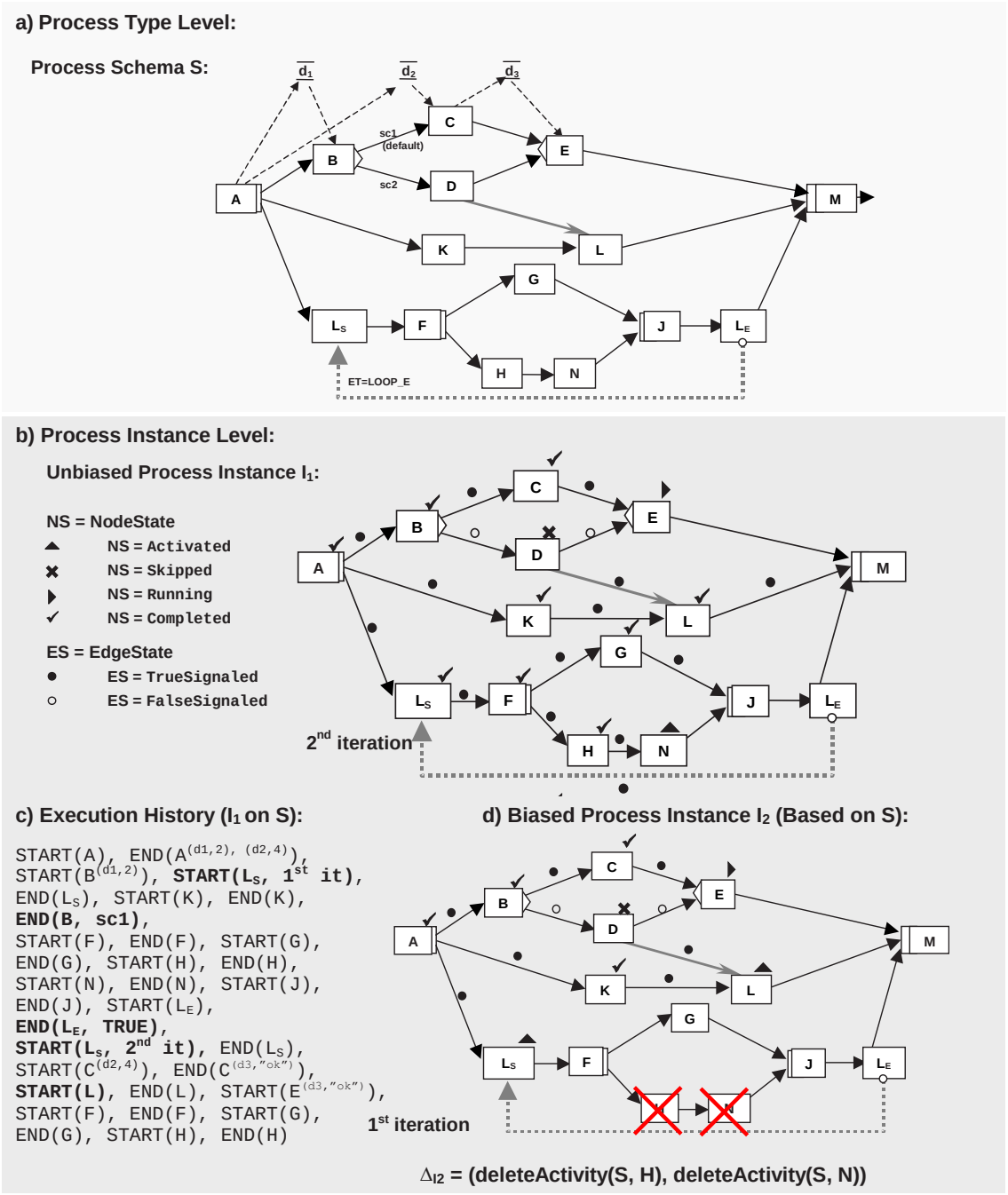


Figure 3.3: Process Type Schema and Process Instances (Abstract Example)

Definition 4 (Correctness of a Process Instance Marking) *Let $I = (S, \Delta_I, M^{S_I}, \ldots)$ be a process instance with (correct) instance–specific schema $S_I = (N_I, D_I, CtrlE_I, SyncE_I, \ldots)$, i.e., S_I is a correct WSM Nets (cf. Definition 2). Let further Start be the start node of S_I (i.e., $Start \in N_I$, $NT(Start) =$ `StartFlow`) and let $M_0^{S_I} = (NS_0^{S_I}, ES_0^{S_I})$ be the initial marking of I, i.e., $NS_0^{S_I}(Start) =$ `Activated`, $\forall n \in N \setminus \{Start\}$: $NS_0^{S_I}(n) =$ `NotActivated`, $\forall e \in (CtrlE_I \cup SyncE_I \cup LoopE_I)$: $ES_0^{S_I}(e) =$ `NotSignaled`. Then:*

M^{S_I} is a correct instance marking if and only if M^{S_I} can be reached from $M_0^{S_I}$ by applying the ADEPT marking and execution rules (cf. Rules 1, Appendix B), i.e.,
$\exists < e_0, \ldots, e_k >$ *($e_i \in$* {`Start(n), End(n)`}*, $n \in N_I$)*
with $M_0^{S_I}[e_0 >, M_1^{S_I}, \ldots, [e_k > M_k^{S_I} =: M^{S_I}$[3]

From Definition 4, a necessary but not sufficient property of a correct process instance marking can be derived.

Lemma 1 (Correctness Property for Process Instance Markings) *Let $I = (S, \Delta_I, M^{S_I}, \ldots)$ be a process instance based on a (correct) WSM Net $S = (N, D, \ldots)$. Marking M^{S_I} has been achieved by applying the ADEPT marking and execution rules (cf. Rules 1, Appendix B). Then for $M^{S_I} = (NS^{S_I}, ES^{S_I})$ the following condition holds:*

$\forall n \in N$ *with* $NS^{S_I}(n) \in$ {`Activated, Running, Completed, Skipped`} $\Longrightarrow$
($\forall\ n^* \in pred^*(S, n)$: $NS(n^*) \in$ {`Completed, Skipped`}*)*

(A formal proof of Lemma 1 is provided in [87].) Altogether, a process instance I is correct if its instance–specific schema is a correct WSM Net (*structural correctness*) and the instance–specific marking M^{S_I} is a correct marking according to Definition 4 (*state–related correctness*). Formally:

Definition 5 (Structural and State-Related Correctness of a Process Instance) *Let $I = (S, \Delta_I, M^{S_I}, Val^{S_I}, \mathcal{H})$ be a process instance based on a WSM Net S. Then: I is a correct process instance if and only if instance schema $S_I := S + \Delta_I$ is a correct WSM Net according to Definition 2 and marking M^{S_I} is a correct marking according to Definition 4.*

Claiming structural correctness for process instances is important in order to avoid undesired system behaviour due to scenarios like deadlock–causing cycles or missing input data. State–related correctness aims at a correct execution behaviour, i.e., avoiding undesired instance states.

[3] Thereby $M[e > M'$ means that starting with marking M event e can take place resulting in marking M'.

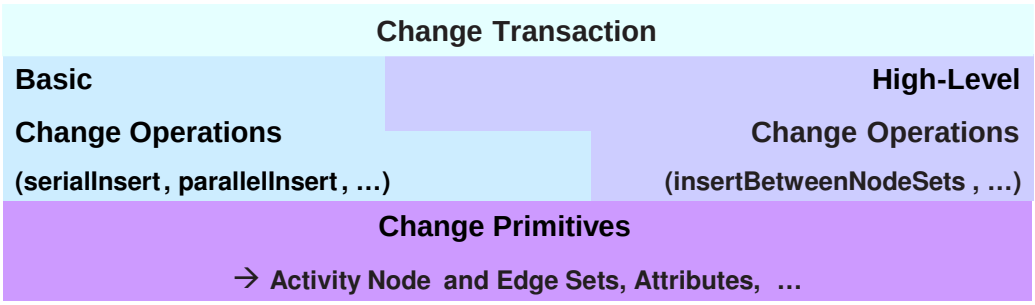


Figure 3.4: Change Primitives, Basic and High–Level Operations, and Transactions [87]

3.2 Change Operations on Well-Structured Marking Nets

In our approach we distinguish between different semantical levels when defining changes (cf. Figure 3.4). The fundament thereby is built by change primitives which are simple changes without formal pre– and post–conditions. Based on these change primitives, basic and high–level change operations can be defined. The correct applicability of these change operations is ensured by preserving special pre– and post–conditions. The set of high–level change operations is offered to application programmers and users to facilitate the change process itself. Finally, an ordered sequence of basic and high–level change operations can be captured within a change transaction. i.e., atomicity is guaranteed by the PMS when applying this sequence of operations.

3.2.1 Change Primitives

Change Primitives build the basis for the definition of other change operations, i.e., all basic and high–level change operations can be mapped onto a set of change primitives (cf. Figure 3.4). An example change primitive is the insertion of an activity node without embedding it into the process context. Further examples are the insertion or the deletion of single control edges. Change primitives are applied to a process schema and afterwards structural correctness of the resulting process schema (cf. Definition 1) has to be checked, e.g., checks on the presence of deadlock–causing cycles have to be carried out. Table 3.1 gives an overview about the change primitives used in our process meta model [87].

3.2.2 Basic and High–Level Change Operations

Generally, change primitives are not directly offered to application programmers or users since their application may be error–prone and difficult. Therefore to offer more convenience, in our approach, pre–defined change operations have been developed. These basic and high–level change operations are augmented with formal pre- and post-conditions such that users do not

Table 3.1: *A Selection of Change Primitives in ADEPT*

Change Primitive *prim* **Applied to Schema S**	**Effects on Process Schema S**
addNodes(S, nodeLabels)	adds activity nodes with identifiers from nodeLabels to N $N^* := N \cup$ nodeLabels
deleteNodes(S, nodeLabels)	deletes activity nodes with identifiers in nodeLabels from N $N^* := N \setminus$ nodeLabels
setNodeAttr(S, nodeLabel, attr, val) addCtrlEdges(S, CtrlEdgeSet)	sets activity attribute attr of activity node nodeLabel to value val adds CtrlEdgeSet to CtrlE $CtrlE* := CtrlE \cup$ CtrlEdgeSet
deleteCtrlEdges(S, CtrlEdgeSet)	deletes CtrlEdgeSet from CtrlE $CtrlE* := CtrlE \setminus$ CtrlEdgeSet
setEdgeAttr(S, edge, attr, val)	sets edge attribute attr of edge egde to value val
addDataElements(S, dataLabels, dom, defaultVal)	adds dataLabels to D whereby domain dom and default value defaultVal are assigned to the new data elements $D^* := D \cup$ dataLabels
deleteDataElements(S, elementSet)	deletes set of data elements elementSet from D $D^* := D \setminus$ elementSet
addDataEdges(S, dataEdges)	adds set of data edges dataEdges to DataE $DataE^* := DataE \cup$ dataEdges
deleteDataEdges(S, dataEdges)	deletes set of data edges dataEdges from DataE $DataE^* := DataE \setminus$ dataEgdes

have to care about correctness issues. Tables 3.2 and 3.3 summarize important basic change operations applicable on WSM Nets [87].

Example (Basic Change Operation): Consider Figure 4.1. Δ_T is a basic change operation. It serially inserts activity X into S by automatically embedding X between activities A and B.

By combining basic change operations high–level change operations can be built like, for example, insertion of activities between two given activity node sets. These high–level change operations are offered to the user in order to provide changes at a high semantical level. Table 3.3 shows examples for such complex operations: Operation $insertBetweenNodeSets(S, X, M_{before}, M_{after})$ inserts new activity X between activity sets M_{before} and M_{after}. Therefore, firstly, basic change operation $parallelInsert(S, X, minBlock(M_{before}, M_{after}))$ is carried out whereby $minBlock(M_{before}, M_{after})$ (cf. Table B.1, Appendix B) denotes the minimal control block containing all activities in M_{before} and M_{after}. Then the desired order relations are set out by inserting sync edges with source activity contained in M_{before} and destination activity X

as well as by inserting sync edges with source activity X and destination activity contained within M_{after}. Another high–level operation is $insertLoopEdge(S, (A, B), lCond)$. It inserts a loop block around an existing control block with start activity A and end activity B. In detail, this operation is carried out by inserting an empty loop block directly before A (by applying basic operations $serialInsertLoopBlock(S, c\_pred(S, A), A, lCond)$) and afterwards moving control block (A, B) from its current position to the position between the newly inserted start node L_S and end node L_E (by applying basic operation $moveBlock(S, (A, B), L_S, L_E)$). The correct application of high–level change operations directly results from the correct application of the used basic change operations [87].

3.2.3 Change Transactions

To offer full flexibility it must be possible to carry out arbitrary combinations of changes from all levels depicted in Figure 3.4. When, for example, inserting two activities and a data dependency between them, it is often desired to apply either all of these change operations or none of them (*atomicity*). In order to achieve this, the respective change operations must be carried out within the same *change transaction* Δ. For our purposes it is sufficient to define a change transaction[4] $\Delta = (op_1, \ldots, op_n)$ as an ordered series of basic and high–level change operations op_i (cf. Tables 3.2 and 3.3) [87].

3.3 Summary

In this section, we have introduced WSM Nets as a language to describe process schemes. WSM Nets are attributed, serial–parallel graphs with additional synchronization links and cover all relevant aspects like control and data flow. Furthermore, we defined how process instances can be started and executed on the basis of WSM Nets. Thereby we have distinguished between unbiased process instances – still running according to the process schema they were started on – and biased process instances. The latter are denoted as biased since they have already undergone an instance–specific change (bias). Finally, we introduced a change framework capturing changes on different semantical levels. Thereby the set of offered (basic) change operations is complete. The respective proof can be sketched as follows: Given two arbitrary (correct) WSM Nets S and S' there is always a sequence of change operations $\Delta = op_1, \ldots, op_n$ whose application to S results in S'.

[4] In the following, we use terms change transaction or change for short

Table 3.2: *Basic Change Operations in ADEPT (1)*

Basic Change Operation Δ Applied to Schema S	**Effects on Process Schema S**
	Additive Change Operations
$serialInsertActivity(S, X, A, B)$	insertion of activity X between directly succeeding activities A and B
$parallelInsertActivity(S, X, (b, e))$	insertion of activity X parallel to control block with start node b and end node e
$branchInsertActivity(S, X, split, join, selcode)$	insertion of activity X within a new branch (with selection code *selcode*) into an alternative branching with `XOr-Split` split and `XOr-Join` join
$serialInsertBlock(S, block, A, B)$	serial insertion of control block *block* between activities A and B
$serialInsertLoopBlock(S, lCond, A, B)$	serial insertion of (empty) loop block (L_S, L_E) between activities A and B (inclusive loop backward edge with loop condition $lCond$)
$parallelInsertBlock(S, block, (b, e))$	insertion of control block *block* parallel to control block with start node b and end node e
$parallelInsertLoopBlock(S, lCond, (b, e))$	serial insertion of (empty) loop block (L_S, L_E) parallel to control block with start node b and end node e inclusive loop backward edge with loop condition $lCond$
$branchInsertBlock(S, block, split, join, selcode)$	insertion of control block *block* within a new branch with selection code *selcode* into an alternative branching with `XOr-Split` split and `XOr-Join` join
$branchInsertLoopBlock(S, lCond, split, join, selcode)$	insertion of (empty) loop block (L_S, L_E) (with loop condition $lCond$) within a new branch with selection code *selcode* into an alternative branching with `XOr-Split` split and `XOr-Join` join
$insertSyncEdge(S, src, dest)$	insertion of a sync edge linking two activities src and $dest$ situated within parallel execution paths
	Subtractive Change Operations
$deleteActivity(S, X)$	deletes activity X from schema S
$deleteBlock(S, block)$	deletes control/loop block *block* from schema S
$deleteSyncEdge(S, edge)$	deletes $edge \in SyncE$ from schema S

Table 3.3: *Basic And High–Level Change Operations in ADEPT (2)*

Basic Change Operation Δ Applied to Schema S	**Effects on Process Schema S**
	Order-Changing Operations
$serialMoveActivity(S, X, A, B)$	moves activity X from current position to position between directly succeeding activities A and B
$parallelMoveActivity(S, X, (b, e))$	moves activity X from current position parallel to control block with start node b and end node e
$branchMoveActivity(S, X, split, join, selcode)$	moves activity X within a new branch with selection code *selcode* into an alternative branching with `XOr-Split` split and `XOr-Join` join
$serialMoveBlock(S, block, src, dest)$	moves control/loop block *block* from current position to position between activities *src* and *dest*
$parallelMoveBlock(S, block, (b, e))$	moves control block *block* parallel to control block with start node b and end node e
$branchMoveBlock(S, block, split, join, selcode)$	moves control block *block* within a new branch with selection code *selcode* into an alternative branching with `XOr-Split` split and `XOr-Join` join
	Attribute Changing Operations
$changeActivityAttribute(S, X, attr, nV)$	changes current value of attribute attr of activity X to nV
$changeEdgeAttribute(S, edge, attr, nV)$	changes current value of attribute attr of edge ∈ CtrlE ∪ SyncE to nV
	Nest and Unnest Operations
$nestBlock(S, ctrlBlock, X)$	nests control block ctrlBlock as a sub process "under" new activitiy node X
$unnestBlock(S, X)$	unfolds sub process deposited "under" activity node X
	Data Flow Change Operations
$addDataElements(S, \ldots)$, $deleteDataElements(S, \ldots)$, $addDataEdges(S, \ldots)$, $deleteDataEdges(S, \ldots)$ Thereby theses data flow change operations are directly mapped onto their counterpart in Table 3.1.	

High–Level Change Operation Δ Applied to Schema S	**Effects on Process Schema S**
$insertBetweenNodeSets(S, X, M_{before}, M_{after})$	insertion of activity node X between activity node sets M_{before} and M_{after}
$insertLoopEdge(S, (A, B), lCond)$	insertion of embracing loop block around existing control block with start node A and end node B inclusive loop backward edge with loop condition lCond

Chapter 4

Migrating Unbiased Process Instances

In this chapter, we consider unbiased process instances, i.e., process instances which have not been individually modified and which therefore still run according to the process schema they were started on, e.g., process instances I_1 and I_2 still running on their original schema S (cf. Figure 4.1). In large–scale environments hundreds up to thousands of such unbiased instances with same process type, being in different execution states, may be concurrently running [57]. Therefore it is crucial to efficiently decide on whether a process type change can be correctly propagated to unbiased process instances or not. Furthermore, it is indispensable to provide advanced algorithms to automatically and efficiently adapt state markings when migrating respective instances to the changed process type schema. In this thesis, we present a comprehensive approach which enables process schema evolution in conjunction with all kinds of process constructs and change operations (e.g., loops, data flow issues, and order–changing operations). In addition, our approach includes methods based on which it is possible to efficiently decide on whether a process instance can be correctly migrated to the changed process type schema or not. Finally, we provide algorithms for automatically adapting markings after migrating process instances to the changed process type schema.

This chapter is organized as follows: In Section 4.1 general challenges of migrating unbiased process instances to a changed process type schema are discussed. Section 4.2 provides a correctness criterion based on which it can be decided whether an unbiased process instance is compliant with a changed process type schema or not. In Section 4.3 we present methods to efficiently ensure compliance and provide algorithms to automatically adapt instance markings after their migration to the changed process type schema in Section 4.4. Finally, this chapter ends with a short outlook on dealing with non–compliant process instances (cf. Section 4.5) and a summary of the presented results (cf. Section 4.6).

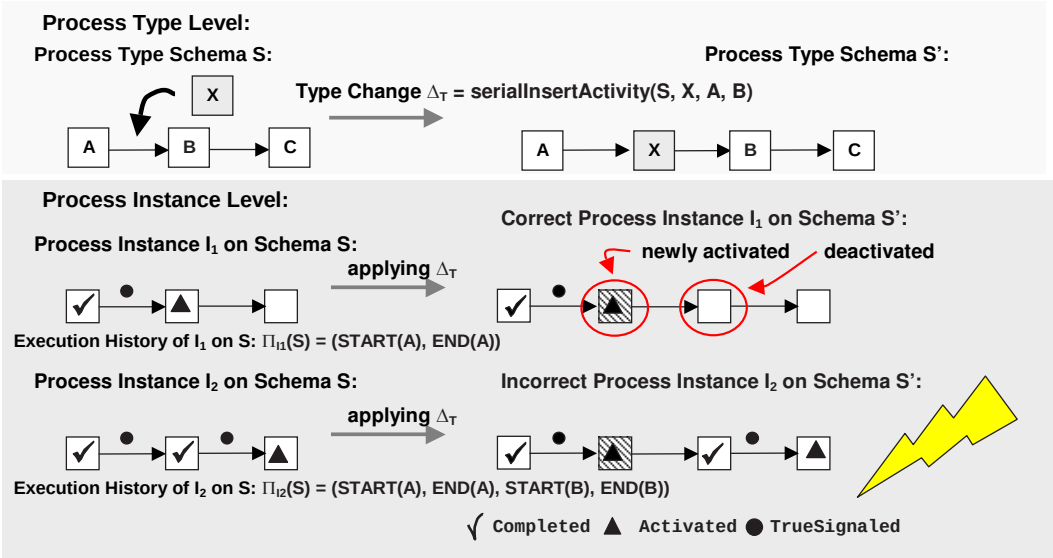


Figure 4.1: Change Scenario 1: Unbiased Instances

4.1 Challenges When Migrating Unbiased Instances

In the following let S be a process type schema (represented by a correct WSM Net) and I be an unbiased process instance running according to S (cf. Definition 3). Let further Δ_T be a process type change transforming S into another (correct) process type schema S'.

At first, to motivate our solution approach we repeat and precise the general challenges for the support of adaptive processes (cf. Section 2.2).

1. **Completeness:** Users should not be unnecessarily restricted, neither by the applied process meta model nor the offered change operations. In particular, important modeling concepts like loops or data flows must not be discounted only to reduce complexity as has been the case in many existing approaches (cf. Section 2). Therefore, we offer expressive control and data flow constructs as well as a complete set of change operations to the user (cf. Section 3).

2. **Correctness:** The key to success when migrating unbiased process instances to a changed process type schema is to find a generally applicable correctness criterion (as, for example, serializability is a correctness criterion for handling concurrent transactions in database management systems). Based on this criterion it should be possible to decide whether a process instance I is *compliant* with a changed process type schema S' or not; i.e., whether process type change Δ_T can be correctly *propagated* to I without causing inconsistencies or errors (like deadlocks or improperly invoked activity programs) in the sequel. More

precisely, it must be ensured that after propagating type change Δ_T to I this process instance is again a correct instance on changed process schema S' according to Definition 5. For unbiased instances structural correctness of the respective instance schema S is guaranteed by the applicability of type change Δ_T to process type schema S and therewith to the instance schema as well. However, things are not that easy when dealing with the second part of Definition 5 – state–related correctness. In this case, we need a criterion for guaranteeing state–related correctness, i.e., to ensure that resulting marking $M^{S'}$ of I on S' is a correct and consistent marking on the changed process type schema S' as well. However, what correct marking means in this context has been already described in Definition 4.

Example 4.1 (Change Propagation): Figure 4.1 depicts a process type schema S and two process instances I_1 and I_2 running on S. For I_1 activity B is activated whereas for I_2 activity C is activated. If now process type change Δ_T is applied to S (by inserting activity X between activities A and B) the challenging question is how to determine whether Δ_T can be correctly propagated to I_1 and I_2 or not.

Propagating type change Δ_T to instance I_1 results in a correct marking on S' (activation of the newly inserted activity X and deactivation of the previously activated activity B). In contrast, the propagation of Δ_T to I_2 has to be forbidden since the marking of instance I_2 becomes incorrect after migrating to changed process schema S': Activity X is inserted with status `Activated` (cf. Rules 1, Appendix B) but followed by an already completed activity B – this offends against Lemma 1 (cf. Section 3.1.2). This definition also complies with the intuitive understanding of a correct process execution: In case of instance I_2 on S' there are two possibilities to proceed the execution of I_2: either we finish activity C what results in an inconsistent final status of I_2 or we undo / redo activity B.

For approaches storing information about previous instance execution, Criterion 6 (as introduced in Section 2.4.2.2) provides a good basis for ensuring correct instance markings after propagating a process type change. More precisely, an unbiased process instance is said to be *compliant* with a changed process type schema if the execution history of this instance can be produced on the changed process type schema as well. For example, the execution history of instance I_1 (cf. Figure 4.1) can be replayed on changed type schema S' resulting in a correct instance marking for I_1 on S'. In contrast, the execution history of I_2 cannot be reproduced on changed type schema S' since the execution history does not contain an entry for newly inserted activity X. Consequently, an incorrect marking would result for I_2 when applying Δ_T in an uncontrolled manner.

As already discussed in Chapter 2, applying this restrictive compliance criterion (as, for example, proposed by [26]) suffers from substantial limitations, e.g., unnecessary restrictiveness regarding the use of loop constructs or lost updates in conjunction with data flow changes. Fundamentally, the compliance criterion itself does not cause these problems, but the particular view on the underlying execution history, i.e., the kind of information provided by the particular history representation. Figuratively, the compliance criterion can be seen as a "hull" where different forms of execution histories can be plugged in. The particular view on the used execution history then leads to different kinds of limitations

when applying the compliance criterion. Therefore it is crucial to find a special view on execution histories based on which compliance becomes a comprehensive correctness criterion; i.e., the criterion should work in conjunction with each kind of change operation applied to each kind of process construct. In our approach, therefore, we introduce the so called *loop–tolerant and data–flow–consistent* view on execution histories.

3. **Efficient Compliance Checks:** Assume that compliance based on loop-tolerant execution histories constitutes an adequate correctness criterion for deciding on compliance of unbiased instances with a changed process type schema. Then the challenging question is how to ensure this criterion *efficiently*. As discussed in Section 2.4.3.2, in many cases, it is very expensive to replay execution histories for a multitude of process instances. Therefore it is crucial for us to provide easily and quickly to check *compliance conditions*. These conditions are based on the actual state of the respective process instance and which consider the applied change semantics. As already mentioned above, this problem is similar to the serializability problem in database management systems. A given schedule of interleaved executed transactions is considered as being correct, if and only if there exists at least one serial (i.e., non–interleaved) execution of these transactions which leads – when starting from the same database state – to the same final state. – It would be very unefficient and also not practicable to directly use this criterion for controlling the concurrent execution of transactions. Instead, highly efficient concurrency control methods have been developed which enforce by construction that the resulting schedules are serializable. Applying these routines dramatically increases efficiency and so do our compliance conditions.

4. **Change Realization:** Assume that process type change Δ_T can be correctly propagated to instance I (along the stated loop–tolerant compliance criterion). Then it should be possible to automatically *migrate* I to changed process type schema S', i.e., without requiring expensive user interactions. In this context, one challenge is to correctly and efficiently adapt instance states. Doing so is a very important job in order not to overstrain users as it is for example the case when they must build up the *Synthetic Cut Over Change* as introduced in Section 2.4.3.1.

4.2 Towards a Loop–Tolerant and Data–Flow–Consistent Correctness Criterion

Core of each approach supporting process schema evolution is a criterion based on which it can be decided whether a process instance I is compliant with a changed process type schema S' or not. There are correctness criteria in the literature but as shown in Section 2.4 they suffer from restrictiveness. Either they restrict users regarding the use of process constructs, e.g., by neglecting loops or data flow issues, or they only allow a subset of change operations. In this section, we provide a correctness criterion which works in conjunction with each kind of change operation based on arbitrary process schemes.

For the sake of completeness we first provide a formal definition concerning the notion of *compliance* for unbiased process instances with the changed process type schema:

Definition 6 (Compliance of Unbiased Process Instances) *Let S be a correct process type schema and I = (S, M^S, Val^S, Π_I^S)[1] be an unbiased process instance on S. Let further S be transformed into another correct schema S' by process type change Δ_T. Then: I is compliant with S' if and only if the application of Δ_T to I again results in a correct process instance I = (S', $M^{S'}$, ...) (according to Definition 5).*

The challenge is to find an adequate criterion for deciding about compliance of unbiased instances with a modified schema. As discussed in the previous section the compliance criterion introduced in Section 2 fully exploits the possibilities of approaches with True/False semantics with respect to compliance; i.e., it does not restrict compliance considerations to the actual state of a process instance (like for example Petri Net based approaches do) but also uses the information about previous instance execution. We adapt this thinking and use this compliance criterion as basis for our approach:

Correctness Criterion 7 (Compliance Of Unbiased Process Instances) *Let the assumption be as in Definition 6. Then: I is compliant with S' if execution history Π_I^S can be replayed on $S' := S + \Delta$ as well, i.e., all events stored in Π_I^S could also have been logged by an instance on S' in the same order as set out by Π_I^S.*

As discussed in the previous section it is essentially important on which form of execution history the application of Criterion 7 is based. With other words we can see Criterion 7 as a "hull" in which different representation forms (views) of execution history Π_I^S can be plugged in. Now the challenging question is what kind of view on Π_I^S is the best.

A first possibility is to use a "minimal" view on Π_I^S which only stores `End` events (for finishing activities, cf. Figure 4.2). However, doing so may lead to three nasty problems – dangling states, data inconsistencies, and restrictiveness in conjunction with loops (cf. Table 4.1). The dangling states problem (cf. Section 2.4.1) results from neglecting `Start` entries in Π_I^S. However, without considering `Start` entries in Π_I^S we cannot distinguish between activities in state `Activated` and those in state `Running`. As a consequence, for example, it is possible to insert a new activity (with state `NotActivated`) before an already running one This, in turn, leads to an inconsistent execution state of the respective instance (cf. Lemma 1, cf. Section 3.1.2). To overcome this problem we need view on Π_I^S logging both, `Start` and `End` events[2]. We call this history view the

[1] Note that $S_I := S$ holds for unbiased process instances.

[2] If we, in turn, use a view on Π_I^S only storing `Start` events and neglecting `End` events we cannot distinguish between running and already finished activities. Using this history view may lead to inconsistencies in conjunction with data flow changes. The reason is that using such a `Start` view on Π_I^S it is not possible to decide whether the concerned activity has only executed its read accesses (when starting the activity) or has already executed its write accesses as well (when finishing the activity). Since this special history view is not common we obstain from further details here.

Table 4.1: Effect of Used View on Execution History on Compliance Checking

View on Execution History: $\Pi_I^S =< e_0, e_1, \ldots, e_k >$ **with**	**Resulting Limitations in Conjunction With Compliance Checking**
(1) `End` *Representation:*[2]	
e_i = `END(X)`	(i) Dangling States Problem (cf. Section 2.4.1)
$(i = 0, \ldots, k)$	(ii) Inconsistencies in Conj. with Data Flow Changes
	(iii) Restrictiveness in Conj. with Loops
(2) `Start / End` *Representation):*	
$e_i \in$ {`START(X), END(X)`}	(ii) Inconsistencies in Conj. with Data Flow Changes
$(i = 0, \ldots, k)$	(iii) Restrictiveness in Conj. with Loops
(3) Data–Flow–Consistent Representation (ADEPT):	
$e_i \in$ {`START(<X, readVal>), END(<X, writeVal>)`	(iii) Restrictiveness in Conj. with Loops
$(i = 0, \ldots, k)$	
(4) Reduced Representation:	
Projection $\Pi_{I\,red}^S$ of Π_I^S on acyclic process graphs (cf. Def. 7)	none of described problems

`Start / End` representation on $\Pi_I^S =< e_0, e_1, \ldots, e_k >$ (with $e_i \in$ {`Start(X), End(X)`}, $X \in N; i = 0, \ldots, k$, cf. Figure 4.2). Based on the `Start / End` representation Criterion 7 exterminates the dangling states problem.

Using the `Start / End` representation of Π_I^S it can be correctly decided whether a process instance is compliant with a changed process type schema or not. This follows directly from Definition 4: If $\Pi_I^S =< e_0, \ldots, e_k >$ can be replayed on changed process type schema S' it establishes a firing sequence $M_0^{S'}[e_0 > M_1^{S'}, \ldots, [e_k > M_k^{S'} =: M^{S'}$ starting from initial marking $M_0^{S'}$ on S' and resulting in (correct) marking $M^{S'}$. Unfortunately, as can be seen from Table 4.1 this representation form is still knotted with several limitations. In the following we illuminate these limitations and provide a further representation form of Π_I^S to overcome these problems.

First of all, the `Start / End` representation form of Π_I^S is not always suitable when considering data flow changes as the following example shows:

Example 4.3.a (Inconsistent Read Data Access Using `Start / End` *Representation of* Π_I^S*):* We consider the process instance depicted in Figure 4.3. Activity C currently has state `Running` and therefore has already read data value 5 of data element d_1. Assume now that due to a modeling error read data edge $(C, d_1, read)$ is deleted and new read data edge $(C, d_2, read)$ is inserted afterwards. Consequently, C should have read data value 2 of data element d_2 (instead of data value 5). This inconsistent read behavior may lead to errors in the sequel, if for example the execution of this instance is aborted and therefore has to be rolled back. However, using the `Start / End` representation form of Π_I^S, this erroneous case would not be detected. As a consequence, this instance would be classified being compliant with the changed process type schema.

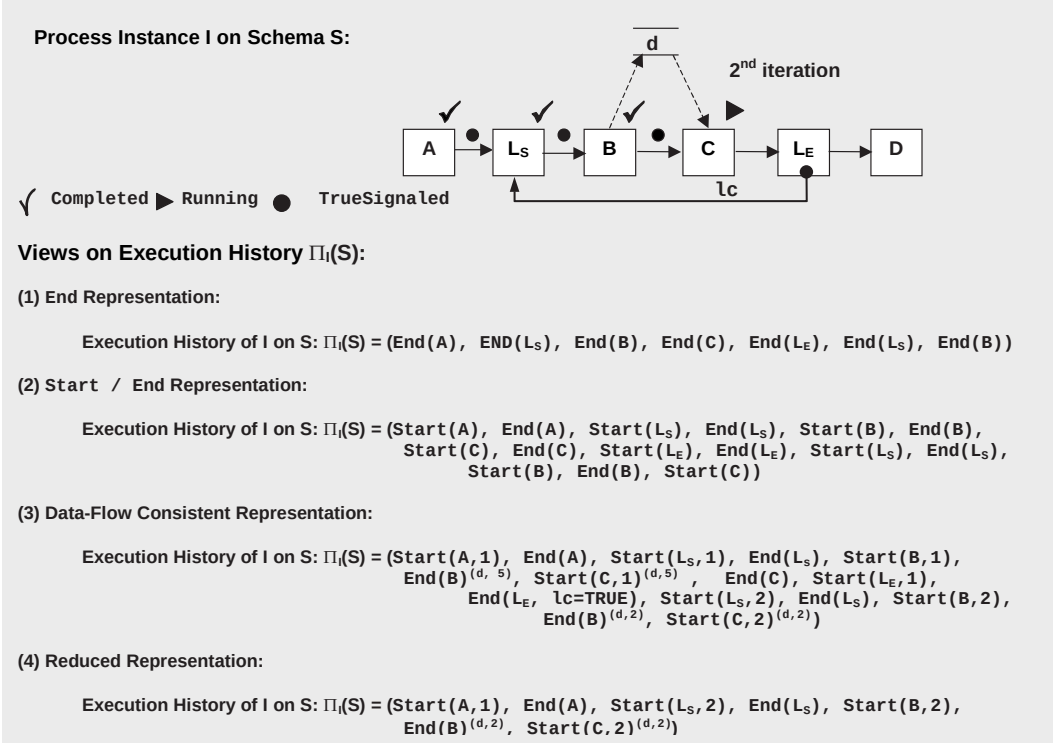


Figure 4.2: Different History Views

For this reason we need an adapted form of the `Start / End` representation of Π_I^S which also incorporates data flow aspects. Fortunately, execution histories as defined in our approach already contain adequate information. More precisely, according to Definition 3 an execution history in ADEPT is defined as

$$\Pi_I^S = \{e_0, \ldots, e_k\} \text{ with}$$
$$e_i \in \{\texttt{Start}^{(d_1^i,v_1^i),\ldots,(d_n^i,v_n^i)}\texttt{(<activity>, <It>)},$$
$$\texttt{End}^{(d_1^i,v_1^i),\ldots,(d_m^i,v_m^i)}\ \texttt{(<activity>, <sc|lc>)}\}.$$

At this tuple (d_i^μ, v_i^μ) describes a read (write) access of e_μ on data element d_i^μ with associated value v_i^μ $(i = 0, \ldots, k)$ if the respective activity is started (completed). We call this view on Π_I^S the *data–consistent* representation of Π_I^S (cf. Table 4.1 and Figure 4.2). Using the data–consistent representation of Π_I^S the problem sketched in Example 4.3.a is overborne as the following example shows.

Example 4.3.b (Consistent Read Data Access Using Data–Consistent Representation of Π_I^S) Again consider the scenario described by Example 4.3.a. Assume that the data–consistent form of Π_I^S is used instead of the `Start / End` representation. Then we obtain that the intended

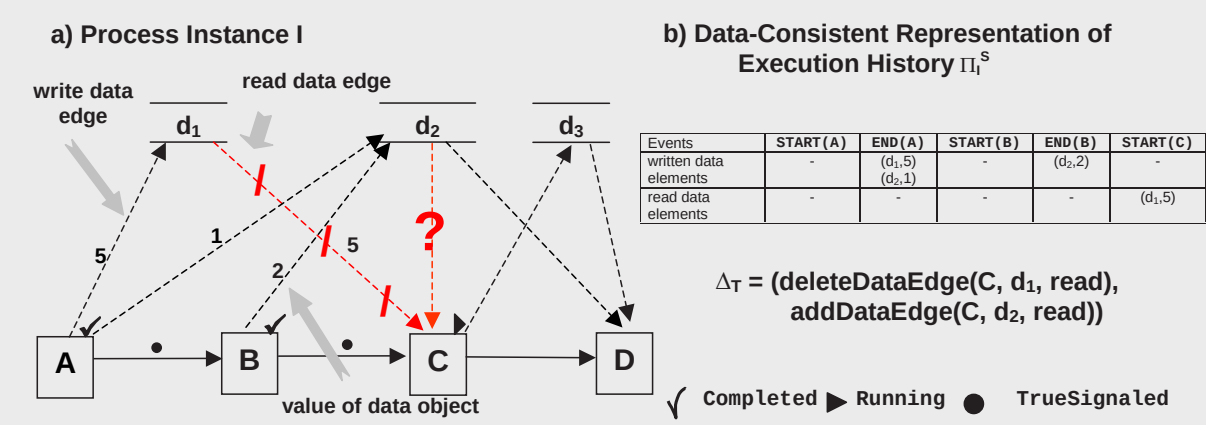


Figure 4.3: Process Instance With Data Flow History (Example)

data flow change Δ_T (deleting data edge $(C, d_1, read)$ and inserting data edge $(C, d_2, read)$ afterwards) cannot be correctly propagated to the depicted instance I. The reason is that the entry `Start(C)`$^{(d_1,5)}$ of Π_I^S cannot be reproduced on the changed type schema.

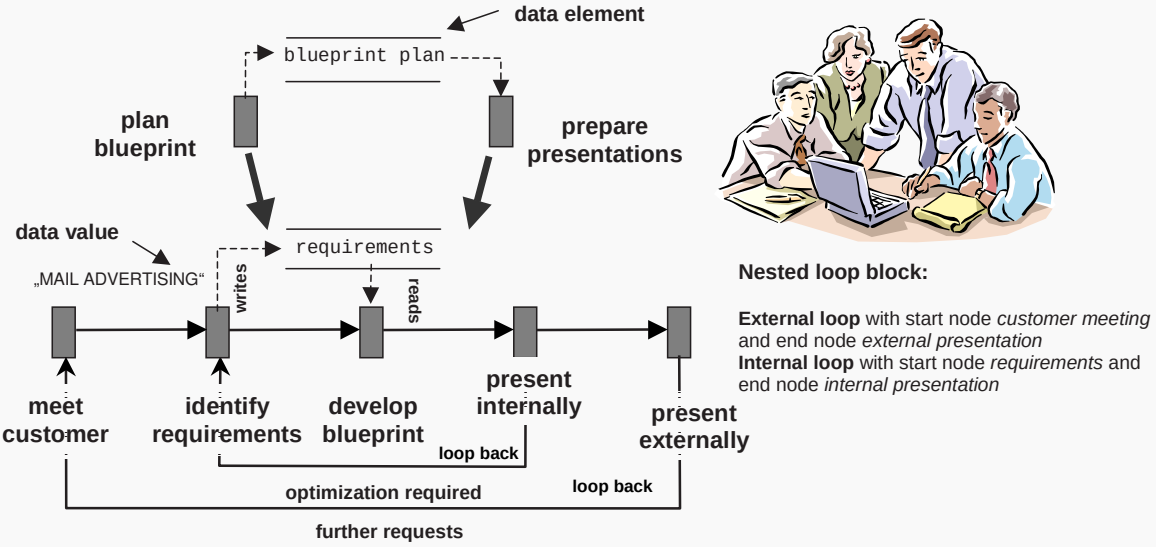
When using the data–consistent representation of execution history Π_I^S one drawback remains: Process instances might be excluded from migrating to the changed schema though this migration would not lead to inconsistencies or errors in the sequel. More precisely, if Criterion 7 is solely based on the data–consistent representation of Π_I^S we exclude actually compliant process instances (as non–compliant). This undesired behaviour occurs if the intended process type change affects loop constructs as the following example shows:

Example 4.4.b (Restrictiveness in Conjunction With Loops Using Data–Consistent Representation of Π_I^S) Consider process type schema S, initially consisting of a nested loop block with one external and one internal loop as depicted in Figure 4.4. Assume that new activities `plan blueprint` and `prepare presentations` (with one data dependency between them) shall be added to process type schema S. This change can be easily accomplished in a correct and consistent manner at the process type level. Assume that instance I is described by (data–consistent) execution history Π_I^S shown in Figure 4.4b. Following Criterion 7 the intended change could not be propagated to I since no history entries for `plan blueprint` and `prepare presentations` have been written within the first (completed) iteration of the external and the internal loop within Π_I^S. Hence, I is considered as non–compliant with the new schema. Consequently, only process instances, which are in the first iteration of both – the internal and the external loop – could be adequately treated in this case.

From a practical viewpoint, however, in most cases it will be too restrictive to prohibit change propagation for in-progress or future loop iterations only because their previous execution is not compliant with the new schema. Think of, for example, medical treatment cycles running for months or years. Every PMS which does not allow propagating such schema changes (e.g., due

Process Type Level:

a) Process Type Schema S:



Process Instance Level:

b) Execution History Π_I^S of Process Instance I:

<S(meet customer, 1st it), END(meet customer),
START(identify requirements, 1st it),
END(identify requirements(requirements, "MAIL ADVERTISING")),
START(develop blueprint(requirements, "MAIL ADVERTISING")), END(develop requirements),
START(present internally), END(present internally, optimization required = FALSE),
START(present externally), END(present externally, further requests = TRUE),
START(meet customer, 2nd it), END(meet customer),
START(identify requirements, 2nd it),
END(identify requirements(requirements, "TV advertising")>

c) Reduced Execution History $\Pi_{I\,red}^S$ of Process Instance I:

<START(meet customer, 2nd it), END(meet customer),
START(identify requirements, 2nd it),
END(identify requirements(requirements, "TV advertising")>

Figure 4.4: Process Type Change and Effects on Running Process Instances (Example)

to the development of a new medical drug) to already running instances (e.g., related to patients expecting an optimal treatment) would not be accepted by the medical staff. Therefore it is stringently necessary to improve the data–consistent representation of Π_I^S in order to exterminate its current restrictiveness in conjunction with loops. The key to solution is to come into the position to differentiate between completed and future executions of loop iterations. From a formal point of view there are two possibilities. The first approach (*linearization approach*) is to logically treat loop structures as being equivalent to respective linear sequences. Doing so allows us to apply Criterion 7 (with full history information). However, this approach has an essential drawback – the explosion of the graph size. An example for this problem is depicted in Figure 4.5a. Here the linearization of a small loop construct (containing a sequence with three activities) already leads to a fairly swelled process graph. The problem becomes even worse when considering nested loop constructs as depicted in the second example in Figure 4.5b. For constructing the linearized graph we have to know the number of iterations for inner loop constructs which can be only done by consulting history information.

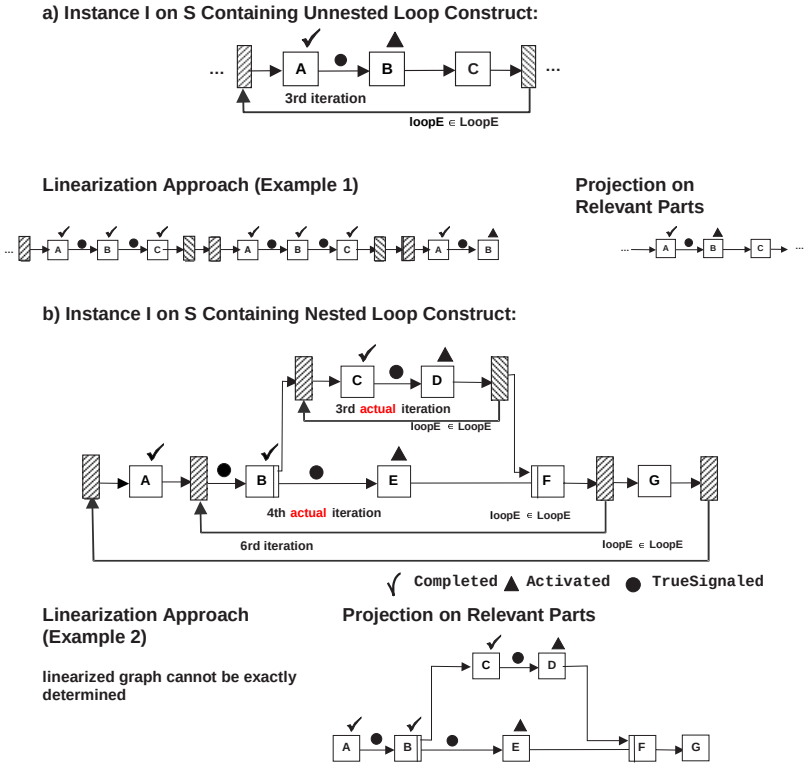


Figure 4.5: Linearization And Projection Approaches

Due to these drawbacks we have adopted another approach which works on a *projection on relevant history information*, i.e., it maintains the loop construct but restricts the necessary evaluation to the relevant parts of the execution history. What does projection on relevant

information exactly mean? Relevant information in conjunction with loops and compliance checking concerns the actual marking of a loop body hiding all information about previous loop iterations. Note that relevant information also includes the last marking of already finished loops. Two examples for maintaining the actual marking of a loop body as relevant information are depicted in Figure 4.5. The challenging question is how to determine the desired projection on the actual marking of loop bodies. A solution is to logically discard all those entries from the execution history which have been produced by another loop iteration than the actual one (if the loop is still executed) or the last on (if the loop execution has been already finished). We call this logical view on the execution history the *reduced execution history*. It can be formally defined as follows:

Definition 7 (Reduced Execution History $\Pi^S_{I\,red}$) *Let S = (N, D, ...) be a process type schema. Let further I be a process instance running according to S with (data–consistent) execution history $\Pi^S_I =< e_0, \ldots, e_k >$ with*

$e_i \in \{\texttt{Start}^{(d^i_1,v^i_1),\ldots,(d^i_n,v^i_n)}\ \texttt{(<act>, <It>)}, \texttt{End}^{(d^i_1,v^i_1),\ldots,(d^i_m,v^i_m)}\texttt{(<act>, <sc|lc>)}\}$
$(i = 0, \ldots, k,\ act \in N).$

(In conjunction with loop executions there may be several entries for one activity.) – The reduced execution history $\Pi^S_{I\,red}$ is obtained as follows:

1. *In the absence of loops $\Pi^S_{I\,red}$ is identical to Π^S_I.*
2. *Otherwise, it is derived from Π^S_I by discarding all history entries related to other loop iterations than the last one (completed loop) or the actual iteration (running loop).*

Formally:

$\Pi^S_{I\,red} := \Pi^S_I$;

forall *loop constructs (L_S, L_E) in S* **do**

if $\exists e_x \in \Pi^S_{I\,red}$ *with* $e_x = \texttt{Start}(L_S,\ i),\ i > 0 \wedge$
$\nexists e_y \in \Pi^S_{I\,red}$ *with* $e_y = \texttt{Start}(L_S,\ i+1)$

then discard all entries $e = \texttt{Start}(n, \mu)$ *and* $E = \texttt{End}(n, \mu)$ *respectively from* $\Pi^S_{I\,red}$ *with:*

$n \in$ (c_succ*(S, L_S) $\cap$ c_pred*(S, L_E)) $\cup \{L_S, L_E\} \wedge \mu < i$

Example 4.4.c (Reduced Execution History): Figure 4.4c depicts the reduced execution history derived from the execution history shown in Figure 4.4b. Since the inner as well as the outer loop are actually executed we have to discard all entries produced by already finished iterations according to Definition 7. More precisely, for the given instance this means to logically discard all entries produced by the first iteration of the outer loop construct.

In summary, applying Criterion 7 based on the reduced representation form of an execution history $\Pi^S_{I\,red}$ yields the desired behavior: it does not needlessly exclude instances from

migrating to a changed process type schema, guarantees consistency in conjunction with data flow changes, and exterminates the dangling states problem. Furthermore, in conjunction with reduced execution histories Criterion 7 is valid for all process execution models which store information about previous execution of process instances, i.e., process meta models with True/False semantics. Examples include Activity Nets as used by WebSphere MQ Workflow [73], and the process meta models applied in BREEZE [104] and WASA$_2$ [136]. A transfer of the compliance framework for WSM–Nets to a compliance framework for Activity Nets can be found in [91].

4.3 On Efficient Compliance Checking

Let again S be a (correct) process type schema and $I_1, \ldots, I_m$ process instances (with execution histories $\Pi^S_{I_1}, \ldots, \Pi^S_{I_m}$) running according to S. Let further Δ_T be a process type schema which transforms S into another (correct) type schema S'.

4.3.1 Motivation

In the previous section we provided a logical correctness criterion (cf. Criterion 6) to be able to decide on whether instances $I_1, \ldots, I_m$ are compliant with S' or not. The challenging question we want to answer in this section is how to ensure this criterion for a possibly large number of running process instances (for example in a medical environment or the financial sector $m > 10.000$ may easily hold [57]). At first glance, an obvious solution is to replay the execution histories of all running process instances on the changed process type schema and – if this is possible – to check afterwards whether the resulting instance marking is consistent (cf. Definition 4). However, doing is so is by far too restrictive for the following reasons:

1. Already the volume of a single execution history $\Pi^S_{I_k}$ may be large. The reason is that besides `Start` and `End` events for started and completed activities further information is logged within $\Pi^S_{I_k}$ like, for example, actor assignments and time stamps (see e.g., [62, 82, 121]). In ADEPT, for example, a single history entry typically comprises, at present, information like event type (`Short, 2 Bytes`), activity identifier (`Long, 8 Bytes`), actor assignment (`Long, 8 Bytes`), time stamp (`Long, 8 Bytes`), iteration counter (`Short, 2 Bytes`), and selection code (`Short, 2 Bytes`). Therewith each history entry has a size of `30 Bytes`. Consider now a process type schema comprising 20 activities with 10 activities being embedded into a loop construct. In case that this loop construct is executed 5 times in the average, 10.000 running process instances would produce an execution history size of approximately 35 MB.

2. In most PMS execution histories are not kept in main memory but are only written to external storage (cf. Figure 4.6). Then they are processed later on by other tools, like, e.g., performance analysis tools. Only in case of crash recovery or if "semantic rollback" (by performing compensation activities [72, 103]) is to be done, the PMS will access these

data again. As these cases occur rather seldom, re–reading the data from external storage does not cause any serious harm. – This picture would change very much if execution history information is used for compliance checking. Then for every process instance an access on external storage becomes necessary which may cause performance problems.

One of the few approaches to think about performance in conjunction with replaying whole execution histories is TRAMs [67] (cf. Section 2). Although this approach is restricted in several directions a very important suggestion is to exploit the semantics of the applied change operations in order to achieve more efficient compliance checks.

Example (Compliance Checking in TRAMs [67]): Let S be a (correct) process type schema and I a process instance running according to S with execution history Π_I^S. Let further Δ_T be a process type change which deletes activity X from S and thereby transforms S into another (correct) type schema S'. Then TRAMs identifies I as compliant with S' if there is no `Start` entry for activity X in Π_I^S.

This example shows that dependent on which change operation is performed only a special part of the (respective) execution history must be considered for compliance checks. We want to incorporate this idea of *exploiting the semantics of the applied change operations* in our approach. However, compliance checking in TRAMs is still based on execution history scans. As discussed above accesses on external storage may cause performance problems, especially in conjunction with a a large number of running process instances. Therefore, we develop methods which avoid expensive history scans by basing compliance checking on the model–inherent markings of our process meta model (cf. Section 3.1.2). Furthermore, our approach is comprehensive, i.e., for every applicable change operation a respective compliance condition is provided.

As mentioned above, we want to avoid access on history data if possible. Instead it would be very desirable to find a compact representation for execution histories. Fortunately, our process meta model offers a memory–based *marking approach* where activity markings represent a consolidated and compact view on the execution history of a particular process instance. At this, the process instance markings represent the actual state of the respective instance execution and therefore perfectly correspond to the reduced form of the underlying execution history.

Example 4.6 (Consolidated View on Execution History) Figure 4.6 depicts the two possibilities to represent the execution state of process instance I_3, i.e., the execution history and the consolidated instance markings. Please note that the instance markings represent the actual state of I_3 and therefore the unnecessary information about previous loop iterations contained in the execution history of I_3 are not present.

Altogether, our compliance checking approach, the so called *compliance conditions on instance markings approach*, is based on two pillars – exploiting the semantics of the applied change operations and process instance markings representing a consolidated view on history data. By doing so, we are able to provide an approach which in the average requires only two comparison operations in contrast to complexity of $O(n)$ for replaying or scanning the whole

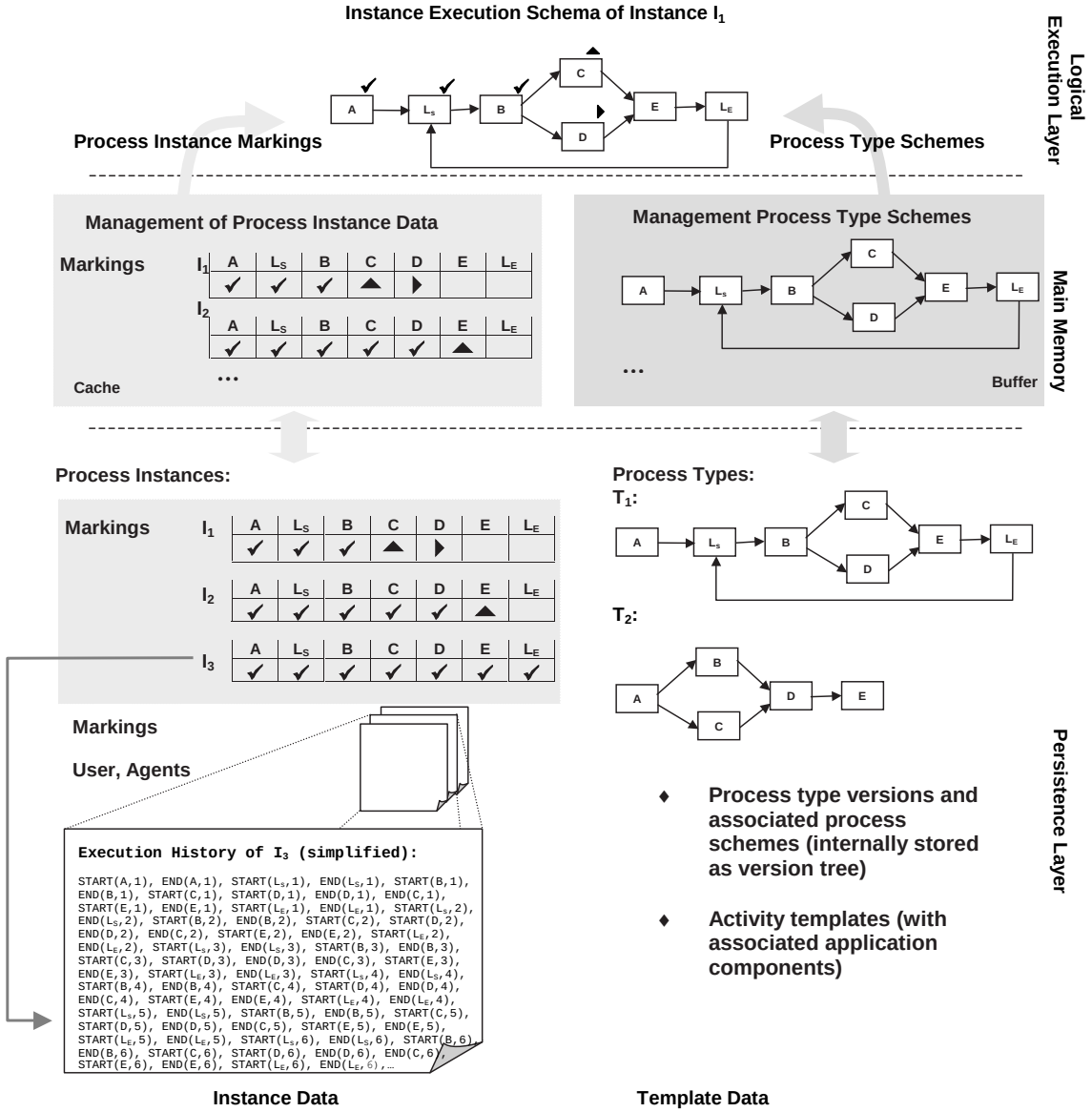


Figure 4.6: Process Type Schema and Instance Management (Simplified)

execution history information (cf. Table 2.2). In addition, the necessary marking information is kept in main memory what offers additional advantages with respect to an efficient implementation. The reason is that doing so results in significantly less accesses on secondary memory. Futhermore, for all provided compliance conditions we formally show that they obey Criterion 7 based on the reduced representation form of execution histories.

4.3.2 Compliance Conditions for Control and Data Flow Changes

The ability to efficiently check compliance is indispensable for the flexible and efficient support of processes by a PMS. Regarding existing approaches, as shown in Section 2, it remains pretty vague how and at which costs compliance can be checked in conjunction with a large number of running process instances. Thus, in our approach we provide efficient and precise conditions on instance markings to ensure Criterion 7 for all possible kinds of changes operations.

4.3.2.1 Compliance Conditions for Additive Change Operations

We start with compliance conditions for the insertion of new activities, control edges, and sync edges. (Note that the addition of a new activity is always accompanied by the insertion of associated control or sync edges, which embed this activity into the process schema context.)

Theorem 1 (Compliance Conditions When Inserting Activities) *Let S = (N, D, NT, CtrlE, SyncE, LoopE, DataE) be a correct process type schema (represented by a WSM–Net) and I be a process instance on S with reduced execution history* $\Pi^S_{I\,red}$ *and with marking* $M^S = (NS^S, ES^S)$*. Assume further that change operation* Δ *transforms S into a correct process type schema S' = (N', D', NT', CtrlE', SyncE', LoopE', DataE') by inserting an activity* n_{insert} *(with associated control and sync edges) into S,*

i.e., $\Delta = [serial|parallel|branch]InsertActivity(S, n_{insert}, \ldots)$ *(cf. Table 3.2).*
Then:

I is compliant with S' ⇔
$\forall\ n \in \{x \in N \mid (n_{insert}, x) \in (CtrlE' \cup SyncE')\}$*:*
NS(n) $\in$ {`NotActivated, Activated, Skipped`} $\vee$
n_{insert} *is inserted into an already skipped branch of an XOR-branching*

A formal proof of Theorem 1 is given in Appendix C (cf. Proof C.1). Informally, for adding activities, compliance can be always guaranteed if all (direct) successors of the newly inserted activity $\mathtt{n}_{insert}$ are actually marked as `Activated`, `NotActivated`, or `Skipped`. In this case they have not yet written any entry into the execution history. Interestingly, the same applies when inserting activities into already skipped branches.

Example 4.7 (Compliance Conditions for Insertion of New Activities) A new activity X is serially inserted between actvities C and F into the process type schema depicted in Figure 4.7. Instance

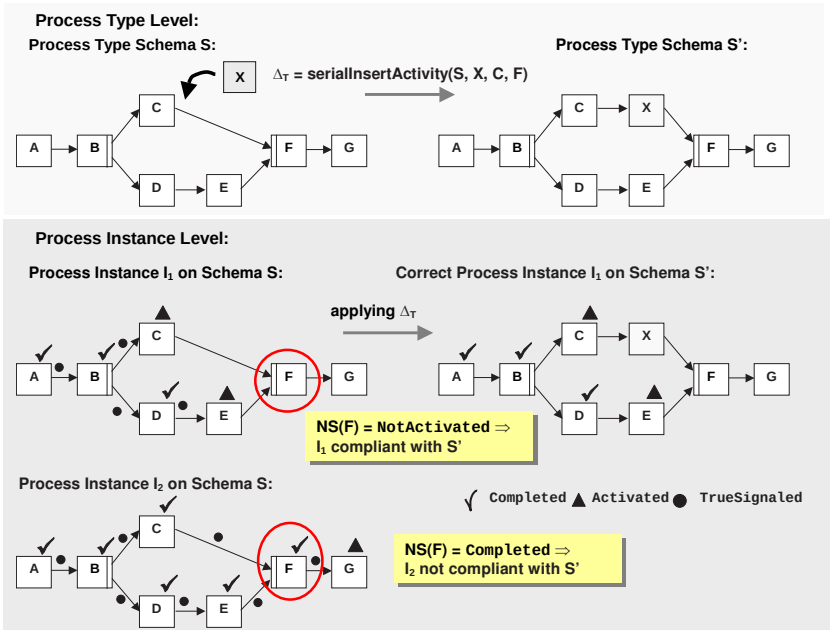


Figure 4.7: Compliance Checks for Insertion of Activities (Abstract Example)

I_1 is compliant with the changed process type schema S' according to Theorem 1 since the direct successor of X in S' (namely F) has state `NotActivated`. In contrast, instance I_2 is not compliant with S' since F is already in execution state `Completed`.

Similar to the insertion of single activities we can state compliance conditions for the insertion of whole control blocks:

Theorem 2 (Compliance Conditions When Inserting Control / Loop Blocks) *Let S = (N, D, NT, CtrlE, SyncE, LoopE, DataE) be a correct process type schema (represented by a WSM–Net) and I be a process instance on S with reduced execution history $\Pi^S_{I\,red}$ and with marking $M^S = (NS^S, ES^S)$. Assume further that change operation Δ transforms S into a correct process type schema S' = (N', D', NT', CtrlE', SyncE', LoopE', DataE') by inserting a control (loop) block $cBlock_{insert} = (b, e)$ ($lBlock_{insert} = (b, e)$) (with associated control and sync edges) into S,*

i.e., $\Delta \in \{[serial|parallel|branch]InsertBlock(S, cBlock_{insert}, \ldots),$
$[serial|parallel|branch]InsertLoopBlock(S, lCond, \ldots)\}$ (cf. Table 3.2).

Then:

I is compliant with S' $\Leftrightarrow$
$\forall\ n \in \{x \in N \mid e \rightarrow x \in (CtrlE' \cup SyncE')\}$:
$NS(n) \in$ `{NotActivated, Activated, Skipped}` $\vee$
$cBlock_{insert}$ ($lBlock_{insert}$) is inserted into an already skipped branch of an XOR-branching

Though no application program or manual action is associated with a loop start or loop end node these nodes write entries into $\Pi^S_{I\,red}$, e.g., loop start nodes write the iteration counter and loop end nodes the result of evaluating the loop condition. These effects on $\Pi^S_{I\,red}$ are reflected by the compliance conditions given in Theorem 2.

Theorem 2 can be proven in an analogous manner as Theorem 1. The reason is that we can logically transform newly inserted control (loop) block $cBlock_{insert}$ ($lBlock_{insert}$) into a complex activity (for example by applying the `nest`–operation; cf. Table 3.2). Note that the compliance condition is satisfied for all activities in $cBlock_{insert}$ ($lBlock_{insert}$) if it is fulfilled for end activity e.

Theorem 3 (Compliance Conditions When Inserting Control And Sync Edges) *Let S = (N, D, NT, CtrlE, SyncE, LoopE, DataE) be a correct process type schema (represented by a WSM–Net) and I be a process instance on S with reduced execution history* $\Pi^S_{I\,red}$ *and with marking* $M^S = (NS^S, ES^S)$. *Assume further that change operation* Δ *transforms S into a correct process type schema S' = (N', D', NT', CtrlE', SyncE', LoopE', DataE').*

(a) Δ *inserts a control edge* $ctrlE = \mathtt{n}_{src} \rightarrow \mathtt{n}_{dest}$ *into S,*
i.e, $\Delta = addCtrlEdge(S, ctrlE)$ *(cf. Table 3.1). Then:*

I is compliant with S' $\Leftrightarrow$

$NS(n_{dest}) \in \{$`NotActivated, Activated, Skipped`$\} \vee$
$[NS(n_{src}) =$ `Completed` $\wedge\ NS(n_{dest}) \in \{$`Running, Completed`$\}$ *with*
$(\exists e_i =$ `End`$(n_{src}),\ e_j =$ `Start`$(n_{dest}) \in \Pi^S_{I\,red} \wedge i < j)]$

(b) Δ *inserts a sync edge* $syncE = \mathtt{n}_{src} \rightarrow \mathtt{n}_{dest}$ *into S (*n_{src} *and* n_{dest} *ordered parallel so far),*
i.e., $\Delta = insertSyncEdge(S, syncE)$ *(cf. Table 3.2). Then:*

I is compliant with S' $\Leftrightarrow$

$[NS(n_{dest}) \in \{$`NotActivated, Activated, Skipped`$\}] \vee$
$[NS(n_{src}) =$ `Completed` $\wedge\ NS(n_{dest}) \in \{$`Running, Completed`$\}$ *with*
$\exists e_i =$ `End`$(n_{src}),\ e_j =$ `Start`$(n_{dest}) \in \Pi^S_{I\,red} \wedge i < j))] \vee$
$[NS(n_{src}) =$ `Skipped` $\wedge\ NS(n_{dest}) \in \{$`Running, Completed`$\})$ *with*
$\forall\ n \in N_{critical}$ *with* $NS(n) \neq$ `Skipped`*:*
$\exists e_i =$ `Start`$(\mathtt{n}_{dest}),\ e_j =$ `End(n)` $\in \Pi^S_{I\,red}$ *with* $j < i)$,
where $N_{critical} = (c\_pred^*(S, n_{src}) \neg\ c\_pred^*(S, n_{dest}))$
and $c\_pred^*(S, n)$ *denotes all direct/indirect predecessors of n in S concerning control edges*]

For a formal proof of Theorem 3 we refer to Proof C.2 in Appendix C. Concerning the insertion of a single control or sync edge, compliance can be always ensured if the target node of the respective edge has not been started yet. This is a sufficient condition for guaranteeing compliance, but it is not always necessary. In a few cases additional information from the

reduced execution history may be required to decide on compliance.

Example 4.8 (Compliance Conditions for Insertion of New Sync Egdes) As an example take process type schema S from Figure 4.8. Assume that sync edge $D \rightarrow F$ is inserted into S. Obviously, instance I_1 is compliant with S' since destination node F of the newly inserted sync edge has node state `Activated`. Regarding process instance I_2 we see that the source node D is skipped and the target node F is completed. According to Theorem 3, in this situation, I_2 is only compliant with the new schema if and only if B has written its end entry before the start entry of F into the execution history ($N_{critical} = \{B\} \wedge \text{NS}(B) \neq$ `Skipped`). This condition cannot be verified by analyzing the marking of I_2. In turn, the information about the execution order between B and F for I_2 has to be extracted from the execution history Π_{I_2} of I_2. We can see from Π_{I_2} depicted in Figure 4.8 that F was started before B. Consequently, the insertion of sync edge $D \rightarrow F$ cannot be propagated to I_2.

Inserting "normal" control edges seems to make not much sense at the moment since inserting of activities is already accompanied by the insertion of respective "embedding" control edges. Nevertheless, insertion and deletion of control edges is important when order–changing operations take place (cf. Section 4.3.2.3). The respective compliance conditions are given in Theorem 6.

4.3.2.2 Compliance Conditions for Subtractive Change Operations

Of course, delete operations are very important for practical purposes as well, e.g., activities may have to be skipped (and therefore the associated control and sync edges embedding the respective activity into the process context be deleted). Thus we provide Theorem 4 which summarizes the compliance conditions for delete operations:

Theorem 4 (Deletion of Activities/Control Edges/Sync Edges) *Let S = (N, D, NT, CtrlE, SyncE, LoopE, DataE) be a correct process type schema (represented by a WSM–Net) and I be a process instance on S with reduced execution history $\Pi^S_{I\,red}$ and with marking $M^S = (NS^S, ES^S)$. Assume further that change operation Δ transforms S into a correct process type schema S' = (N', D', NT', CtrlE', SyncE', LoopE', DataE').*

(a) Δ deletes an activity n_{delete} from S (including the re-linking of control edges), i.e., $\Delta = deleteActivity(S, n_{delete})$ (cf. Table 3.2). Then:

I is compliant with S' $\Leftrightarrow$
$NS(n_{delete}) \in$ {`NotActivated, Activated, Skipped`}

(b) Δ deletes a control or sync edge edge = (n_{src}, n_{dest}) from S, i.e., $\Delta = delete[Ctrl|Sync]Edges(S, edge)$ (cf. Table 3.1 + 3.2). Then:

I is compliant with S'

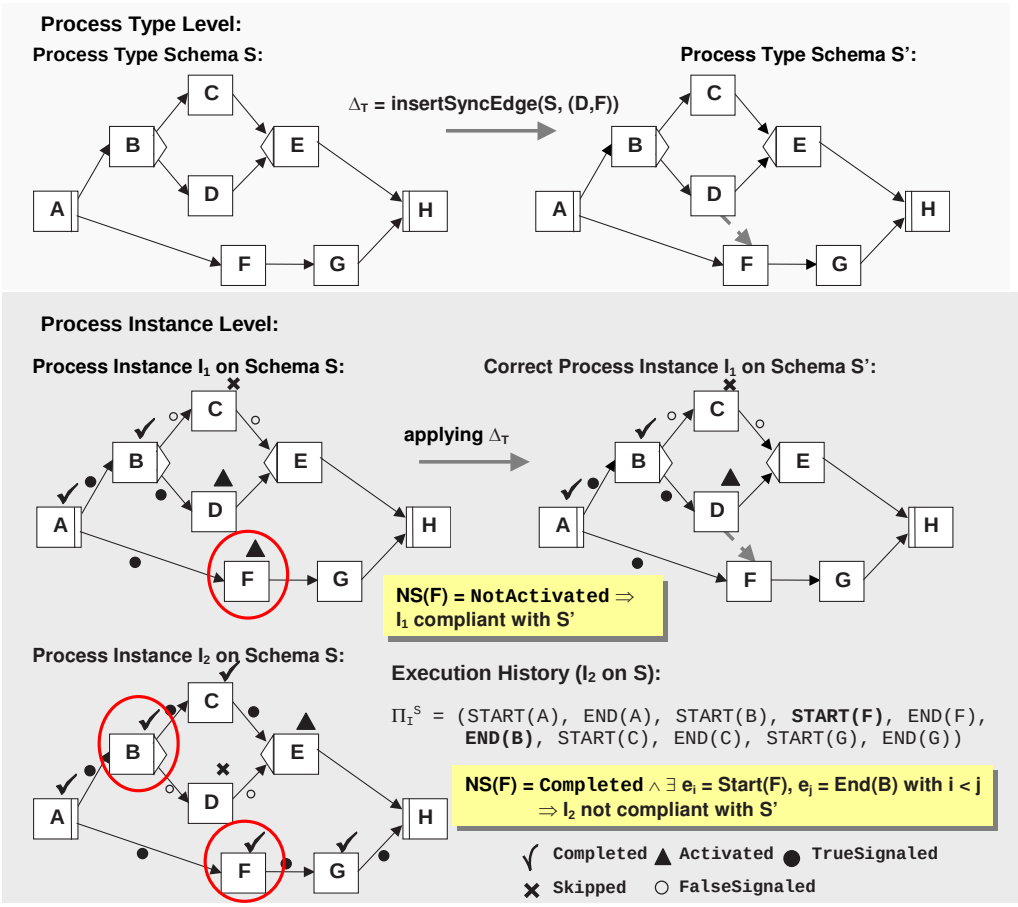


Figure 4.8: Compliance Checks for Insertion of Sync Edges (Abstract Example)

A formal proof for Theorem 4 is given in Appendix C (cf. Proof C.3). For delete operations compliance checks can be always performed solely on basis of activity markings. Intuitively, only those activities of a process instance I can be dynamically deleted which have not yet written any entry into the execution history. This is the case if the node marking of the activity to be deleted is `NotActivated, Activated`, or `Skipped`. Concerning control / sync edges their deletion is uncritical with respect to the compliance of process instances with the resulting process schema. Note that order relations between the source and end activity nodes of deleted edges are abolished. Therefore the previous execution can be replayed on the changed schema.

The following theorem comprises the compliance conditions for deleting control blocks.

Theorem 5 (Deletion of Control Blocks) *Let $S = (N, D, NT, CtrlE, SyncE, LoopE, DataE)$ be a correct process type schema (represented by a WSM–Net) and I be a process instance on S with reduced execution history $\Pi^S_{I\,red}$ and with marking $M^S = (NS^S, ES^S)$. Assume further that*

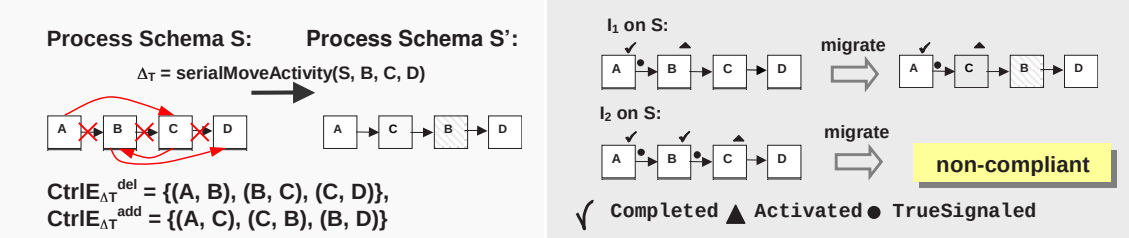


Figure 4.9: Order–Changing Operation (Example)

change operation Δ transforms S into a correct process type schema S' = (N', D', NT', CtrlE', SyncE', LoopE', DataE) by deleting control block $cBlock_{delete} = (begin, end)$ from S, i.e., $\Delta = deleteBlock(S, cBlock_{delete})$ (cf. Table 3.2). Then:

I is compliant with S' $\Leftrightarrow$
NS(begin) $\in$ {`NotActivated, Activated, Skipped`}

Similar to the insertion of control blocks for the deletion of control blocks we can argue that the respective proof can be mapped onto the proof for deleting activities if $cBlock_{delete}$ is nested to an activity. In contrast, the compliance conditions are only fulfilled for all actvities of $cBlock_{delete}$ if they are satisfied for activity *begin*.

4.3.2.3 Compliance Conditions for Order–Changing Operations

Order–changing operations like moving an activity n_{move} from its current position to position $(src, dest)$ are carried out by applying a sequence of edge insertion and deletion operations. Note that n_{move} is not deleted (and then inserted at its new position). Therefore the data context of n_{move} is preserved. More precisely, control or sync edges are deleted to unhinge n_{move} from its current position in S and to break open the new position. Afterwards n_{move} has to be woven into the process context again. Here new control edges are inserted to close the gap of the previous position of n_{move} and to embed n_{move} between src and $dest$.

Example 4.9.a (Order–Changing Operation) In Figure 4.9 activity B is moved to position between C and D. This order–changing operation Δ_T is realized by deleting a set of existing control edges $CtrlE^{del}_{\Delta_T} = \{(A, B), (B, C), (C, D)\}$ and by inserting a set of new control edges $CtrlE^{add}_{\Delta_T} = \{(A, C), (C, B), (B, D)\}$.

Since a move operation actually consists of the composed application of deleting and inserting edges, intuitively, the compliance condition of the respective move operation is fulfilled if the compliance conditions of the single edge operations are fulfilled (cf. Theorems 3 + 4). Since deletion of sync / control edges has no impact on compliance (cf. Theorem 4) only compliance

conditions for inserting control edges have to be aggregated to find the desired compliance condition, formally:

Theorem 6 (Moving Activities) *Let S = (N, D, NT, CtrlE, SyncE, LoopE, DataE) be a correct process type schema (represented by a WSM–Net) and I be a process instance on S with reduced execution history* $\Pi^S_{I\,red}$ *and with marking* $M^S = (NS^S, ES^S)$*. Assume further that change operation* Δ *transforms S into a correct process type schema S' = (N', D', NT', CtrlE', SyncE', LoopE', DataE) by moving activity* n_{move} *from its current position to its new position within S'* [3]*. Thereby* Δ *adds set of control edges*

$CtrlE^{add}_{\Delta_T} = \{(n_1, n_2) | n_1 \in c\_pred(S, n_{move}),\ n_2 \in c\_succ(S, n_{move})\} \cup$
$\{(n_1, n_{move}) | n_1 \in c\_pred(S', n_{move})\} \cup$
$\{(n_{move}, n_2) | n_2 \in c\_sucee(S', n_{move})\}$
i.e., $\Delta = [serial|parallel|branch]moveActivity(S, n_{move}, \ldots)$ *(cf. Table 3.3). Then:*

I is compliant with S' $\Leftrightarrow$
$\forall (n_{src}, n_{dest}) \in CtrlE^{add}_{\Delta_T}$:
$NS(n_{dest}) \in \{\texttt{NotActivated, Activated, Skipped}\} \vee$
$[NS(n_{src}) = \texttt{Completed} \wedge NS(n_{dest}) \in \{\texttt{Running, Completed}\}$ *with*
$(\exists e_i = \texttt{End}(n_{src}),\ e_j = \texttt{Start}(n_{dest}) \in \Pi^S_{I\,red} \wedge i < j)]$

A formal proof of Theorem 6 can be found in Appendix C (cf. Proof C.4). For insertion of control edges into $CtrlE^{add}_{\Delta_T}$ we claim that each destination activity must be `NotActivated`, `Activated`, or `Skipped`. In order to move an activity n_{move}, at first, we have to fill the "gap" resulting from unhinging n_{move} from its former context, i.e., we insert a control edge between the direct predecessor(s) and successor(s) of n_{move} in S. Secondly, n_{move} has to be embedded into the process context by inserting new control edges between all direct predecessor of n_{move} within S' and n_{move} as well as between n_{move} and all direct successors of n_{move} in S'.

Example 4.9b (Compliance Condition When Moving Activities) Consider Figure 4.9. For instances I_1 and I_2 the activity markings of activities B, C and D have to be checked according to Theorem 6. Instance I_1 is compliant with S' since B is `Activated` and C and D are still in state `NotActivated`. In contrast, instance I_2 is not compliant with S' since B has been already completed.

ADEPT offers additional operations to move whole control blocks (cf. Table 3.3). Obviously, moving blocks could be also achieved by individually moving each single activity of the respective block. However, the $[serial|parallel|branch]MoveBlock(S, \ldots)$ operations often better suit to the users' intention since they may lose overview when moving each single activity, especially for large control blocks.

Operations $[serial|parallel|branch]MoveBlock(S, block, \ldots)$ work like the respective operations for moving activities: First, control block $block = (b, e)$ is unhinged from its current context

[3] Note that the static correctness of process schema S' is guaranteed by the post–conditions of the applied move operation (cf. Table 3.2)

by deleting all incoming edges of start activity b and all outgoing edges of end activity e. Then (b, e) is reembedded at its new position by inserting control edges between all direct predecessors of start activity b and b as well as between end activity e and all of its direct successors.

Theorem 7 (Moving Blocks) *Let S = (N, D, NT, CtrlE, SyncE, LoopE, DataE) be a correct process type schema (represented by a WSM–Net) and I be a process instance on S with reduced execution history* $\Pi^S_{I\,red}$ *and with marking* $M^S = (NS^S, ES^S)$*. Assume further that change operation* Δ *transforms S into a correct process type schema S' = (N', D', NT', CtrlE', SyncE', LoopE', DataE) by moving control / loop block block* $= (b, e)$ *from current position to its new position within* S' *by adding set of control edges*

$CtrlE^{add}_{\Delta_T} = \{(n_1, n_2) | n_1 \in$ *c_pred(S, b),* $n_2 \in$ *c_succ(S, e)*$\} \cup$
$\{(n_1, b) | n_1 \in$ *c_pred(S', b)*$\} \cup \{(e, n_2) | n_2 \in$ *c_succ(S, e)*$\}$
i.e., $\Delta = [serial|parallel|branch]MoveBlock(S, block, \ldots)$ *(cf. Table 3.3). Then:*

I is compliant with S' $\Leftrightarrow$
$\forall (n_{src}, n_{dest}) \in CtrlE^{add}_{\Delta_T}$*:*
NS(n_{dest}*)* $\in$ {`NotActivated, Activated, Skipped`} $\vee$
[*NS(*n_{src}*)* $=$ `Completed` $\wedge$ *NS(*n_{dest}*)* $\in$ {`Running, Completed`} *with*
$(\exists e_i =$ `End`*(*n_{src}*),* $e_j =$ `Start`*(*n_{dest}*)* $\in \Pi^S_{I\,red} \wedge i < j)$]

We abstain from a formal proof for Theorem 7. Again this operation can be mapped to the respective operation for moving single activities by nesting control / loop block (b, e) to a complex activity (which can then be moved to the respective position within S'). Note that for moving control / loop blocks start activity b as well as end activity e have to be taken into account.

4.3.2.4 Compliance Conditions for Data–Flow Changes

Changes of the data flow may become necessary in conjunction with control flow schema changes (e.g., removing associated data edges of an activity to be deleted) or may have to be applied independently in order to re-link data edges or data elements (e.g., if errors in the modeled flow of data have to be corrected). To modify the data flow schema, our approach offers operations for adding and deleting data elements as well as data edges.

Taking the compliance property from Criterion 7, all conditions set out for control flow changes (cf. Theorem 1 – 6) must be further fulfilled. Additionally, it is required that each started or finished activity (of the respective process instance) would have read and each finished activity would have written the same data element values also on the new schema. The compliance of a process instance in case of data flow changes can be easily checked based on the following conditions.

Theorem 8 (Compliance Conditions For Data Flow Changes) *Let S = (N, D, NT, CtrlE, SyncE, LoopE, DataE) be a correct process type schema (represented by a WSM–Net) and I be a*

process instance on S with reduced execution history $\Pi^S_{I\,red}$ *and with marking* $M^S = (NS^S, ES^S)$. *Assume further that change operation* Δ *transforms S into a correct process type schema S' = (N', D', NT', CtrlE', SyncE', LoopE', DataE).*

(a) Δ *inserts a data element d into S, i.e.,* Δ = *addDataElements(S,* $\{d\}$, ...*). Then:*

I is compliant with S'.

(b) Δ *deletes a data element d from S, i.e.,* Δ = *deleteDataElements(S,* $\{d\}$, ...*). Then:*

I is compliant with S' $\Leftrightarrow$
No read or write access on d by an activity with state `Running` *or* `Completed`

(c) Δ *inserts or deletes a read edge (d, n, read),*
i.e., $\Delta \in$ {*addDataEdges(S,* {*(d, n, read)*}*), deleteDataEdges(S,* {*(d, n, read)*}}*. Then:*

I is compliant with S' $\Leftrightarrow$ *NS(n)* $\in$ {`NotActivated, Activated, Skipped`}

(d) Δ *inserts or deletes a write edge (d, n, write),*
i.e., $\Delta \in$ {*addDataEdges(S,* {*(d, n, write)*}*), deleteDataEdges(S,* {*(d, n, write)*}}*. Then:*

I is compliant with S' $\Leftrightarrow$ *NS(n)* $\neq$ `Completed`

(for all data flow operations see Table 3.3)

A formal proof can be found in Appendix C (cf. Proof C.5). To explain how Theorem 8 works we provide the following example:

Example 4.3.c (Compliance Conditions for Data Flow Changes) Consider Figure 4.3 where type change Δ_T cannot be applied to instance I according to Theorem 8. The reason is that activity C is `Running` and therefore already has read data element d_1. Consequently re–linking the data access of C to d_2 would be prohibited what complies to the desired behavior in this case.

As already mentioned, data flow adaptations also become necessary in conjunction with the insertion and deletion of activities. In this case, the conditions of Theorem 8 are already met if the state conditions of the according node insertion or deletion operations are fulfilled (cf. Theorem 1 and 4). Similar to the aggregated compliance conditions for order–changing operations concerning data flow changes, the conditions for using complex operations arise from the aggregation of the conditions of basic change operations (cf. Section 4.3.2.5).

4.3.2.5 Further Changes Operations

For the following kinds of change operations like attribute–changing, nesting, and high-level (complex) changes we informally provide their particular compliance conditions but abstain from presenting formal proofs. The reason is that we want to achieve completeness of compliance conditions regarding possible kinds of changes on the one hand but avoid unnecessary

recapitulation of already presented concepts and proofs on the other hand. Again let S be a (correct) process type schema and I an unbiased process instance with reduced execution history $\Pi^S_{I\,red}$ and with marking $M^S = (NS^S, ES^S)$ on S. Let further Δ_T be a process type change transforming S into another (correct) schema S'.

Attribute–Changing Operations: Basically, we distinguish between changes of activity attributes, i.e., working assignments and edge attributes like selection codes or priorities [87]. Let first Δ_T change attribute *attr* of activity X to value *val*. Then the running instance I is compliant with S' if and only if $NS(X) \in \{$`NotActivated, Activated, Skipped`$\}$ holds (cf. Table 4.2). Exemplarily, let activity X be a patient preparation step and the respective actor assignment be *"role = Nurse"*. Then we can change this actor assignment until activity X is selected by a nurse and therefore its state changes to `Running`. As a consequence X writes a `Start` entry with actor assignment *"role = Nurse"* into $\Pi^S_{I\,red}$ which would be not producible on S' if the actor assignment attribute is changed. Please note that it is possible to change attributes like actor assignments even if the respective activity is activated and already distributed into work lists. The reason is that based on our marking adaptation algorithm presented in the following section we "flag" the respective activity as newly to evaluate and re–distribute it to worklists according to the new working assignment. If Δ_T changes an attribute of $edge = (n_{src}, n_{dest})$ we claim that $n_{dest} \in \{$`NotActivated, Activated, Skipped`$\}$. This leads to a very interesting marking adaptation process (cf. Algorithm 2): if we change the selection code of an outgoing edge of an `XOr-Split` we eventually have to check whether the completed deadpath elimination is still correct or has to be adjusted.

Nest and Unnest Operations: Propagating nest or unnest operations to running instances is always uncritical regarding compliance. The reason is that we only change the view on the process not the control or data flow themselves. Consequently, logically, there is always a common flat representation of the respective process type and instance schemes based on which the respective execution histories can be arbitrarily transferred.

High–Level (Complex) Change Operations: Generally the compliance conditions of high–level or complex change operations are fulfilled if the compliance conditions of the applied basic or high–level change primitives are fulfilled. Of course, these conditions may overlap such that several optimizations are conceivable.

As an example, in Table 4.2 we provide the compliance conditions for complex change operation $insertBetweenNodeSets(S, X, M_{bef}, M_{aft})$ which inserts activity X between activity sets M_{bef} and M_{aft}. Informally this change works by first inserting X into parallel position to a minimal control block containing M_{bef} and M_{aft}. Afterwards the desired order dependencies between activities in M_{bef} and X as well as between X and activities in M_{aft} are set out by inserting respective sync edges. Altogether, we apply a parallel insertion operation and several insertions of sync edges. If we look at the compliance condition given in Table 4.2 we see that the insertion operations demands that all direct successors via control and sync edges in S' have to be in states `NotActivated, Activated`, or `Skipped`. This conditions already comprise the conditions for inserting sync edges between X and activities in M_{aft}. For inserting sync edges

Table 4.2: *Compliance Conditions for Attribute–Changing, Nesting and Complex Changes*

Change Operation Δ	**Compliance Conditions**
	Attribute Changing Operations
$changeActivityAttribute(S, X, attr, val)$	I is compliant with $S \Longleftrightarrow$ $NS(X) \in$ {`NotActivated, Activated, Skipped`}
$changeEdgeAttribute(S, edge, attr, val)$	I is compliant with $S \Longleftrightarrow$ $NS(n_{dest}) \in$ {`NotActivated, Activated, Skipped`} where $edge = (n_{src}, n_{dest})$
	Nest and Unnest Operations
$nestBlock(S, ctrlBlock, X)$	I is compliant with S
$unnestBlock(S, X)$	I is compliant with S
	High–Level Change Operations
$insertBetweenNodeSets(S, X, M_{bef}, M_{aft})$	I is compliant with $S \Longleftrightarrow$ $\forall$ n $\in$ {x $\in$ N $\mid$ $X \rightarrow$ x $\in$ (CtrlE' $\cup$ SyncE')}: NS(n) $\in$ {`NotActivated, Activated, Skipped`}
$insertLoopEdge(S, (A, B), lCond)$	I is compliant with $S \Longleftrightarrow$ NS(A) $\in$ {`NotActivated, Activated, Skipped`}

between activities in M_{bef} and X we have to claim teh conditions of Theorem 3 for X. However new activities are always inserted with state `NotActivated` (and are re–evaluated if need be) by what the conditions of Theorem 3 for X are already fulfilled.

For high–level operation $insertLoopEdge(S, \ldots)$ the compliance conditions are dominated by the compliance conditions for inserting the empty loop block (L_S, L_E) (cf. Section 3.2). The reason is that the respective condition claims that NS(A) $\in$ {`NotActivated, Activated, Skipped`}. For moving (A, B) between L_S anbd L_E we get that the destination activities of newly added control edges contained in $CtrlE_{\Delta}^{add} = \{(L_S, A), (B, L_E), (L_E, c\_succ(S, B))\}$ should all have states in {`NotActivated, Activated, Skipped`} or – if they are already running or finished and the respective source activity is also completed – they should have written their history entries after the history entries of the source nodes (cf. Theorem 7). These claims are already fulfilled by claim NS(A) $\in$ {`NotActivated, Activated, Skipped`}: L_E is inserted with state `NotActivated` and since NS(A) $\in$ {`NotActivated, Activated, Skipped`} the same must hold for the successors of A (cf. Definition 4), in particular for B and its direct successors.

4.4 Adapting Process Instance Markings After Migration

At this stage we are able to quickly and efficiently decide whether a running process instance is compliant with a changed process type schema or not. As we have already discussed in Section

2.4 it is also indispensable to provide methods to automatically adapt instance markings after migration to the changed schema. As we have also shown there are actually approaches dealing with adaptive processes which burden the user with adapting instance markings. An example is the *Synthetic Cut Over Changes* proposed by Ellis et al [39] which forces the user to generate and distribute instance markings within a very complex Petri Net (cf. Figure 2.13). Obviously, doing so cannot be expected of the user in practice.

Therefore in our approach we have developed an algorithm to automatically adapt instance markings after migration to the changed schema. This algorithm must obey two fundamental claims: First, it has to output correct instance markings on the changed type schema according to Definition 4. We formally prove that this correctness claim is properly fulfilled by our algorithm (cf. Theorem 9). Second the marking adaptation algorithm has to work efficiently. The reason is that based on the particular situation the necessary marking adaptations may turn out very extensive. Thereby, generally, the effort of instance marking adaptations depends on the kind and scope of the change. Except initialization of newly inserted nodes and edges, no adaptations will become necessary if execution of has not yet entered the change region.

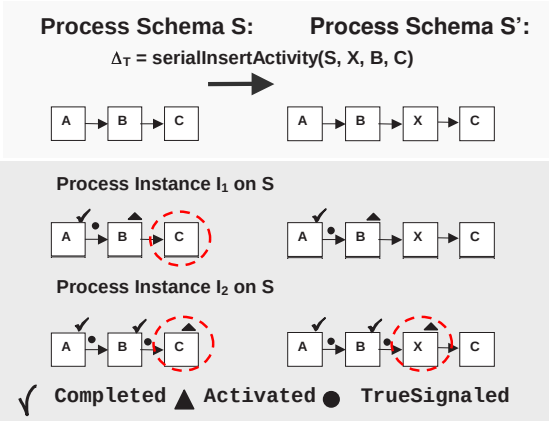


Figure 4.10: Markings Adaptations (Example)

An example is depicted in Figure 4.10 where instance I_1 has not yet entered the process area where activity X is inserted into. In other cases more extensive marking adaptations may be required. An activity X, for example, may have to be deactivated if new control edges are inserted with X as target activity. This is the case for instance I_2 in Figure 4.10 where activity C has to be reset from state `Activated` to state `NotActivated` end therefore has to be removed from respective work lists. Conversely, newly inserted activity X is evaluated to `Activated` and immediately offered in work lists.

Even more complex marking adaptation scenarios arise if activities are, for example, inserted into already skipped branches. The most effort has to be spent if the selection codes of edges with an `XOr-Split` as source activity are changed (cf. Section 4.3.2.5). As a result whole branches have to be reset and other branches may undergo further deadpath–eliminations. An example is shown in the following section.

We now describe how markings can be automatically and efficiently adapted when migrating compliant instances. Initially, we can restrict marking evaluations to those nodes and edges, which constitute the context of a change region. We sketch how these sets can be determined for selected change primitives as well as for complex changes. Based on this, we present an algorithm which correctly calculates new markings for compliant instances.

For the following considerations again let S be a (correct) process type schema and I an unbi-

ased process instance with reduced execution history $\Pi^S_{I\,red}$ and with marking $M^S = (NS^S, ES^S)$ on S. Let further Δ_T be a process type change transforming S into another (correct) schema S' and I be compliant with S' (cf. Criterion 7 and compliance conditions of Section 4.3.2).

4.4.1 Initial Determination Of Newly To Evaluate Activities And Edges

Generally, when migrating a process instance I to a changed process schema S' only those activities and edges should be re–evaluated for which state adaptations are possibly necessary, i.e., we avoid to analyze activities and edges which are not involved into type change Δ_T at all. More precisely, we have to determine activity set N_{check} and edge set E_{check} for which the state has to be newly evaluated on S'. We call these sets the *inital activity and edge sets*. Table 4.3 summarizes the intial activity and edge sets for the particular change operations.

When inserting new activity X (including context edges) we have to re–evaluate the states of all direct successors of X and all incoming control edges of X to avoid inconsistent markings in the following (cf. Table 4.3). A re–evaluation of X is only necessary if one of the incoming edges of X can be marked as `TrueSignaled` or `FalseSignaled`. Similarly when inserting a new control block (b, e) we intially re–evaluate the states of all incoming edges of start node b. It is possible that further activities of control block (b, e) have to be re–marked, for example, if the whole control block has to be skipped. Similarly to insertion of activities, we have to consider the direct successors of end node e of the control block what is reflected in $N_{check}(\Delta_T)$. When inserting a new control edge (n_{src}, n_{dest}) again the edge itself and its destination node n_{dest} have to be newly evaluated. Evaluating n_{dest} is also necessary if the control edge itself is marked as `NotSignaled`. It this case possibly the activation of n_{dest} has to be reset due to an insertion of another edge.

If we delete an activity X we first remove all incoming and outgoing edges of X and then bypass the arising "hole" with a respective number of control edges. With this consideration $N_{check}(\Delta_T)$ and $E_{check}(\Delta_T)$ for deleting activities directly follow (cf. Table 4.3). (Analogously, $N_{check}(\Delta_T)$ and $E_{check}(\Delta_T)$ result for the deletion of control blocks.)

Moving an activity X from its current position to its new position within S' can be either seen as a sequence of edge change operations (cf. Section 4.3.2.3) or – logically – as the combined application of deleting X and the afterwards insertion of X at the respective position within S'. For both cases we have to check the states of all direct successors of X for the original and the resulting position ($N_{check}(\Delta_T)$). Furthermore, it is necessary to take all incoming edges of X at the new position and the edges bypassing the "hole" at the original position into account ($E_{check}(\Delta_T)$).

The most "tricky" re–evaluations may result for attribute–changing operations. For changes on attributes of activity X it may be necessary to re–distribute X to work lists if working assignments are changed (cf. Table 4.3). In this case we "flag" $N_{check}(\Delta_T)$ with the instruction to check necessity of re–distribution. If we change the selection code of edges it may be necessary to re–activate already skipped branches of an alternative branching and to skip other branches

Table 4.3: *Initial Activity And Edge Sets for Re–Evaluation of Instance Markings*

Applied Change Operation Δ	**Initial Activity And Edge Sets**
Inserting Activity X (including context edges)	$N_{check}(\Delta) := \{n \in N \mid (X,n) \in E'\}$ $E_{check}(\Delta) := \{(n_{src}, n_{dest}) \in E' \mid n_{dest} = X\}$
Inserting Control Block (b,e) with start node b and end node e	$N_{check}(\Delta) := \{n \in N' \mid (e,n) \in E'\}$ $E_{check}(\Delta) := \{(n_{src}, n_{dest}) \in E' \mid n_{dest} = b\}$
Inserting Control or Sync Edge (n_{src}, n_{dest})	$N_{check}(\Delta) := \{n_{dest}\}$ $E_{check}(\Delta) := \{(n_{src}, n_{dest})\}$
Deleting Activity X (including deletion and insertion of context edges)	$N_{check}(\Delta) := \{n \in N \mid (X,n) \in E\}$ $E_{check}(\Delta) := E_{add}$ $(E_{add} := E' \setminus E)$
Deleting Control or Sync Edge (n_{src}, n_{dest})	$N_{check}(\Delta) := \{n_{dest}\}$
Deleting Control Block (b,e) with start node b and end node e (including deletion and insertion of context edges)	$N_{check}(\Delta) := \{n \in N \mid (e,n) \in E\}$ $E_{check}(\Delta) := E_{add}$ $(E_{add} := E' \setminus E)$
Moving Activity X to new position in S' (including context edges)	$N_{check}(\Delta) := \{n \in N' \mid (X,n) \in E'\} \cup$ $\{n \in N \mid (X,n) \in E\}$ $E_{check}(\Delta) := \{(n_{src}, n_{dest}) \in E' \mid n_{dest} = X\} \cup$ $\{(n_1,n_2) \in E' \mid (n_1,X), (X,n_2) \in E\}$
Moving Block $block = (b,e)$ to new position in S' (including context edges)	$N_{check}(\Delta) := \{n \in N' \mid (e,n) \in E'\} \cup$ $\{n \in N \mid (e,n) \in E\}$ $E_{check}(\Delta) := \{(n_{src}, n_{dest}) \in E' \mid n_{dest} = b\} \cup$ $\{(n_1,n_2) \in E' \mid (n_1,b), (e,n_2) \in E\}$
Changing Attribute attr of Activity X to new value $nVal$ **if** $attr = wAss$ with $wAss$ is working assignment attribute of X **else**	$N_{check}(\Delta) := \{(X, "r")\}$ $E_{check}(\Delta) := \emptyset$ $N_{check}(\Delta) := \emptyset$ $E_{check}(\Delta) := \emptyset$
Changing Attribute attr of edge (n_{src}, n_{dest}) to new value $nVal$ **if** $attr = sc$ and $n_{src} = split$ with sc selection code and $split$ `XOr-Split` of an alternative branching **else**	$N_{check}(\Delta) := \emptyset$ $E_{check}(\Delta) := \{(n_{src}, n) \in$ alternative branch in $E'\}$ $N_{check}(\Delta) := \emptyset$, $E_{check}(\Delta) := \emptyset$
Inserting New Loop Block (n_1, n_2) with start node n_1 and end node n_2	$N_{check}(\Delta) := \{n \in N' \mid (n_2,n) \in E'\}$ $E_{check}(\Delta) := \{(n_{src}, n_{dest}) \in E' \mid n_{dest} = n_1\}$

Algorithm 1 (Determining $N_{check}(\Delta)$ and $E_{check}(\Delta)$ For Change Transaction Δ)

input
$\Delta := (op_1, \ldots, op_n)$: `Applied change transaction Δ consisting of change operations` $op_1, \ldots, op_n$`;`
output
$N_{check}(\Delta)$ `and` $E_{check}(\Delta)$: `Sets of newly to evaluate activities / edges;`
begin
$E_{check}(\Delta) = \emptyset$; $N_{check}(\Delta) = \emptyset$; `//Initialization`
for `i:=1 to n` **do** `//Aggregation`
$N_{check}(\Delta)$`:=`$N_{check}(\Delta) \cup N_{check}(op_i)$;
$E_{check}(\Delta)$`:=`$E_{check}(\Delta) \cup E_{check}(op_i)$;
done
$E_{check}(\Delta)$`:=`$E_{check}(\Delta) \cap$ `E';`
$N_{check}(\Delta)$ `:=` $N_{check}(\Delta) \cap$ `N;`
end

in contrast. To be able to do so we add all outgoing edges of the respective `XOr-Split` to $E_{check}(\Delta_T)$ as starting point for re–evaluation.

How can we determine $N_{check}(\Delta_T)$ and $E_{check}(\Delta_T)$ for a change transaction $\Delta_T := (op_1, \ldots, op_n)$? One would guess that we get the desired sets by unifying $N_{check}(op_i)$ and $E_{check}(op_i)$ for all applied change operations $op_i (i = 1, \ldots, n)$. However this is not always the case since the particular change operations op_i can be based on each other. In particular, effects of op_i may be (partially) overrided by precedent change operations $op_1, \ldots, op_{i-1}$. Algorithm 1 incorporates this aspect as follows: $E_{check}(\Delta_T)$ is determined by unifying $E_{check}(op_i)$ for all applied change operations whereby those edges are removed which are not present in the new schema S'. In fact these edges have been produced but have been also removed again by subsequent operations. Regarding $N_{check}(\Delta_T)$ only those actitivites have to be taken into account which were present in original schema S. (The following algorithm works by only evaluating newly inserted activities if an incoming edge has been also newly marked.) Altogether, Algorithm 1 only determines the necessary activity and edge sets to be re–evaluated and therefore the input of Algorithm 2 is kept minimal.

4.4.2 Marking Adaptation Algorithm

Let $S = (N, D, CtrlE, SyncE, LoopE, \ldots)$ be a (correct) process type schema which is transformed into another (correct) process schema S' by applying change transaction $\Delta_T := (op_1, \ldots, op_n)$. Let further $N_{check}(\Delta_T)$ and $E_{check}(\Delta_T)$ be as determined by Algorithm 1. Then Algorithm 2 determines the new activity and edge markings $\mathrm{M}^{S'} = (\mathrm{NS}^{S'}, \mathrm{ES}^{S'})$ of a compliant process instance I on S after its migration to S'. In essence, this algorithm is based on the ADEPT marking and execution rules (cf. Figure 3.3d) and Rules 1). Starting point are

the inital sets of newly to evaluate activities and edges (cf. Table 4.3 and Algorithm 1). If need be, Algorithm 2 carries out cascading re–evaluations, e.g., if a deadpath elimination has to be executed or already skipped branches have to be reset.

How Algorithm 2 works is illustrated by the following examples:

Example 4.10 (Marking Adaptations When Inserting An Activity): For type change Δ_T as depicted in Figure 4.10 we get $N_{check}(\Delta_T) = \{C\}$ and $E_{check}(\Delta_T) = \{(B, X)\}$ according to Table 4.3. Starting from this for instance I_2 Algorithm 2 first re–marks (B, X) to `TrueSignaled` and adds X to $N_{check}(\Delta_T)$. Then C is taken from $N_{check}(\Delta_T)$ and reset to `NotActivated` (doing so is accompanied by removing entries of C from work lists). Finally, the state of X is set to `Activated` (and entries of X are distributed to work lists) since incoming edge (B, X) is `TrueSignaled`. Then Algorithm 2 breaks with a correct marking of I_2 on S' as depicted in Figure 4.10.

Example 4.11 (Marking Adaptations When Changing A Selection Code): Type Change Δ_T changes the selection code of edge (B, D) from former transition condition $sc1 : x > 5$ to $sc1 : x > 8$ (cf. Figure 4.11). Consider running instance I: Activity A has already written value 6 to data element x. Therefore when completing `XOr-Split` B transition condition $sc1 : x > 5$ has been evaluated to true and therefore edge (B, D) is marked with `TrueSignaled`. In contrast the default branch is skipped. Regarding compliance conditions of Table 4.2 I is compliant with S' since $\text{NS}(D) =$ `Activated`. Therefore we are allowed to apply Δ_T to I. But how to adapt markings of I on S'? First of all we determine $N_{check}(\Delta_T) = \emptyset$ and $E_{check}(\Delta_T) = \{(B, C), (B, D)\}$ according to Table 4.3. Based on this Algorithm 2 works like follows: We first fetch (B, C) from $E_{check}(\Delta_T)$ and find that $\text{ES}^{S'}(B, C)$ has to be re–evaluated to `TrueSignaled`. Therefore we add destination activity C to $N_{check}(\Delta_T)$. Then we take (B, D) from $E_{check}(\Delta_T)$ and reset its marking to `FalseSignaled` applying the ADEPT marking and execution rules. In turn we add destination activity D to $N_{check}(\Delta_T)$. Afterwards we analyze activity C which has to be re–marked as `Activated`. As a consequence we add outgoing edge (C, E) to $E_{check}(\Delta_T)$. In contrast, D has to be skipped and also outgoing edge (D, E) is added to $E_{check}(\Delta_T)$. Now we have to turn back to $E_{check}(\Delta_T)$. (C, E) has to be reset to `NotActivated` applying the ADEPT marking rules whereas (D, E) has to be marked as `FalseSignaled`. Since states of both edges have changed we add their destination activity D to $N_{check}(\Delta_T)$. For D no re–evaluation is necessary and finally Algorithm 2 finishes with correct marking of I on S' as depicted in Figure 4.11.

Using Algorithms 1 and 2 reduces complexity of marking adaptations compared to approaches using reachability analysis or – in most cases – compared with replaying the whole execution history. Algorithm 1 has to be applied only one time per change operation whereas Algorithm 2 has to be excuted for each compliant process instance. Complexity of Algorithm 2 is restricted by $O(n)$ whereby n is the number of activities in S'. In addition, compliance checks need at most $O(n)$ for each running instance. However, in most cases we can restrict compliance checks and re–evaluations of markings to a small subset of activities and edges of S

Algorithm 2 (Re–Evaluation of Process Instance Markings After Migration)

```
input
    NS^S, ES^S:  Current Activity and Edge Markings of I on S;
    N_check(Δ_T), E_check(Δ_T):  Sets of newly to evaluate activites and edges;
output
    NS^S', ES^S':  Newly calculated Activity and Edge Markings of I on S';
begin
  //Initialization of ES^S';
  forall e ∈ E' do
    if e ∈ E then //edge has been already present in E:=(CtrlE ∪ SyncE ∪ LoopE);
      ES^S'(e) := ES^S(e);
    else //initialize edge with NotSignaled;
      ES^S'(e) := NotSignaled;
    done
    // Initialization of NS^S'
    forall n ∈ N' do
      if n ∈ N then //activity has been already present in N;
        NS^S'(n) := NS^S(n);
       else // initialize activity with NotActivated;
        N^S' := NotActivated;
    done
    // Re-Evaluation of Activities in N_check(Δ_T) and Edges in E_check(Δ_T)
    repeat
      while E_check(Δ_T) ≠ ∅ do
        Fetch edge e = (n_src, n_dest) from E_check(Δ_T);
        Check whether ES^S'(e) = TrueSignaled or ES^S'(e) = FalseSignaled by
        applying ADEPT marking rules on e - if so adapt edge marking respectively;
        if marking of e has been changed then
          N_check(Δ_T) := N_check(Δ_T) ∪ {n_dest};
        endif
      done
      while N_check(Δ_T) ≠ ∅ do
        Fetch element n from N_check(Δ_T);
        if n = (n̄, "r") do //working assignment has been changed (cf.  Table 4.3)
          re-distribute activity n̄ to work lists;
        else //n is "normal" activity
         if NS^S'(n) ∈ {Activated, NotActivated} then
           check whether n should be marked with NS^S'(n) ∈ {NotActivated, Activated, Skipped}
           by applying ADEPT execution rules on n - if so adapt marking respectively;
           if N^S'(n) has been set to Skipped then
             E_check(Δ_T) := E_check(Δ_T) ∪ {e = (n_src, n_dest) ∈ E'|n_src = n}
           endif
         endif
         if NS^S'(n) = Skipped then
           check whether n should be marked with NS^S'(n) ∈ {NotActivated, Activated}
           by applying ADEPT execution rules on n - if so adapt marking respectively;
           if N^S'(n) has been set to {NotActivated, Activated} then
             E_check(Δ_T) := E_check(Δ_T) ∪ {e = (n_src, n_dest) ∈ E'|n_src = n}
           endif
         endif
      done
    until E_check(Δ_T) = ∅ and N_check(Δ_T) = ∅
end
```

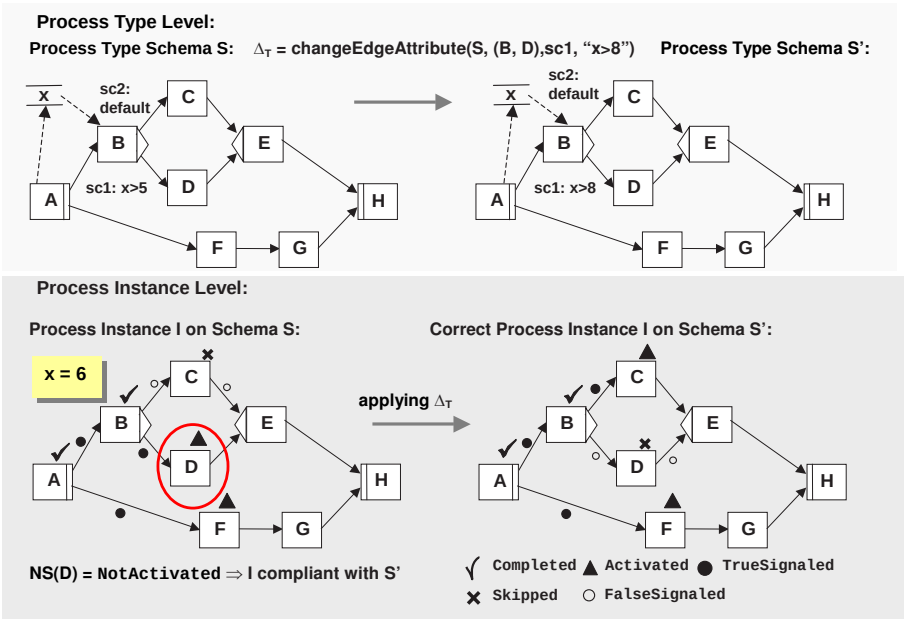


Figure 4.11: Marking Adaptations When Changing A Selection Code (Example)

or S' respectively. This is even the case if changes work on different parts of the process schema what is an immense improvement to approaches like [39] for which the analysis regions comprise the unified region of all applied changes (cf. Figure 2.13).

Finally, it can be formally shown that the application of Algorithm 2 on compliant instances results in a correct instance marking according to Definition 4. The reason is that the marking resulting from applying Algorithms 1 and 2 is equal to the marking which results from replaying the (reduced) execution history of I on S'. The latter is a correct marking on S' according to Criterion 7. Formally:

Theorem 9 (Correctness of Marking Adaptation Approach) *Let S be a process type schema and I an unbiased process instance running on S with instance marking $M^S = (NS^S, ES^S)$ and execution history Π_I^S. Let further Δ_T be a process type change which transforms S into another (correct) process schema S' and let I be compliant with S'. Then the instance markings resulting from replaying Π_I^S on S', i.e., $M^{S'}_{replay} = (NS^{S'}_{replay}, ES^{S'}_{replay})$ conincides with the marking resulting from applying Algorithms 1 + 2, i.e., $M^{S'}_{adapt} = (NS^{S'}_{adapt}, ES^{S'}_{adapt})$. Formally:*

$$NS^{S'}_{replay} = NS^{S'}_{adapt} \wedge ES^{S'}_{replay} = ES^{S'}_{adapt}$$

For a formal proof of Theorem 9 we refer to Proof C.6 in the appendix.

4.5 Coping with Non-Compliant Instances

Now we are able to decide whether a process instance is compliant with a changed process type schema or not. If so we can automatically adapt instance markings. However the interesting question remains how to deal with *non–compliant* process instances. Of course, at minimum it is required that non–compliant instances can finish their execution according to the original process schema they were started on or migrated to. However, there are approaches which try to bring non–compliant instances back to a compliant form.

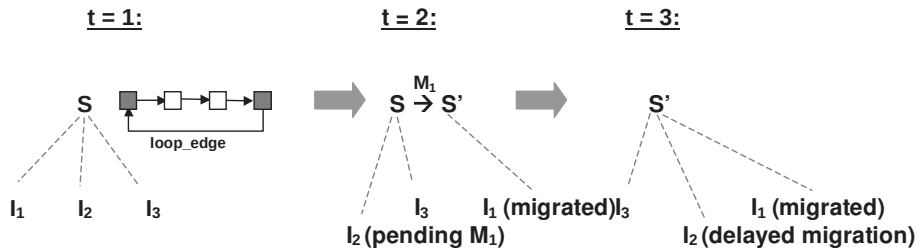
Breeze [104, 106] first groups process instances with respect to their *compliance* with the changed schema. For non-compliant instances the *compliance graph* is constructed which serves to migrate these instances to the changed schema as well. The compliance graph consists of three parts: The first parts corresponds to the subgraph of the respective schema which reflects the execution state up to which compliance is given; e.g., when deleting an activity X the first part results as the subgraph "until" X. The second part acts as a bridge between the original and the changed process schema, i.e., it consists of compensation activities for those activities which have been executed "too far" for compliance. In this case all running or completed successors activities of X would be rolled back. The last part of the compliance graph consists of that subgraph of the changed schema remaining after removing the first part and the compensated activities. Altogether, in this approach a non–compliant instance is partially rolled back into a compliant state and then executed according to the changed schema. An obvious drawback of this approach is that it is not always possible to find compensating activities, i.e., to adequately roll back non–compliant instances.

Bichler et al. [21] propose to extend the set of non–compliant process instances by a partial rollback as well. This approach raises similar problems as discussed for Breeze.

An alternative approach to deal with non–compliant instances is not to bring these instances back to a compliant form but to wait until they are compliant again. More precisely, we distinguish non–compliant instances into instances, which can never be migrated (*"never-more-compliant instances"*) and others, which only fail because the current execution of a loop iteration has proceeded too far. The latter instances then become a candidate for migration when the loop enters its next iteration (*"re-compliant instances"*).

Re–compliant instances. In particular, the marking of a loop is reset if a loop back takes place such that Criterion 7 will be satisfied with delay. Thus, process instances which are not compliant according to their actual loop iteration may become re–compliant when another loop iteration takes place and therefore can be migrated to the new schema with delay (*delayed migration*). As shown in Figure 4.12, re-compliant instances can be held as "pending to migration" until the loop condition is evaluated.

The treatment of re-compliant instances, which is especially important in conjuntion with long-running processes, is not as trivial as it looks like at first glance. At first, if an instance contains nested loops there can be several events (loop backs) to trigger the execution of a previously delayed migration. Furthermore, the interesting question remains how to deal with



Event	Condition	Action
Loop_Back	ES(loop_edge) = TRUE_SIGNALED	migrate I_2 to S'

Figure 4.12: Principle Of Delayed Migration

pending instances if further schema changes take place.

An alternative approach supporting *delayed migrations* of non-compliant instances is offered by *Flow Nets* [39]. Even if an instance I on S is not compliant with S' within the actual iteration of a loop, a delayed migration of I to the new change region is possible when another loop iteration takes place. As an example consider Figure 2.13(2). In case of a loop back the token is passed to the initial place of the loop. If then the process type change is applied to I the resulting firing sequence can be also a firing sequence on S'. Consequently, I is said to be compliant with S' again.

4.6 Summary

The approach presented in this chapter fulfills all challenges set out for the migration of unbiased process instances to a changed process type schema as depicted in Figure 4.13. We have presented a compliance criterion based on which it can be decided whether a running process instance can be migrated to a changed process type schema or not. We have discussed that the quality of Criterion 7 depends on the kind of execution history used. To achieve a maximum of quality in our approach we use the so called reduced representation for execution histories. Doing so, we exterminate the dangling states problem introduced in Section 2.4.1 and any data inconsistenices. Furthermore, we are not too restrictive in conjunction with loops. To efficiently check the compliance criterion we have provided precise and easy to check compliance conditions based on which complexity of compliance checks can be dramatically reduced. Finally we have presented an algorithm which automatically adapts instance markings after migration to the changed schema.

A very important additional remark is that the results presented in this section can be easily transferred to other process meta models as well. In [91] we have presented a complete framework for *Activity Nets* as used for example in Websphere MQ Workflow [73]. Activity Nets also use

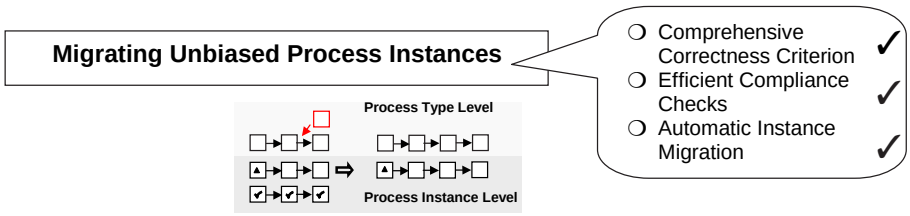


Figure 4.13: Migrating Unbiased Process Instances

model–inherent markings and consequently belong to approaches with True/False-Semantics (cf. Figure 2.1). The only adaptations compared to the compliance conditions in this work have been made due to the fact that Activity Nets use only one kind of edges, namely control edges, and have to be acyclic (what reduces complexity on the one hand but restricts the expressiveness of the process models on the other hand).

Chapter 5

Migrating Biased Process Instances

To provide a fully flexible PMS it is indispensable to support changes at both the process type level and the level of single process instances. As discussed in Section 2 both kinds of changes have been an important research topic in the process management literature [118, 1, 26, 37, 39, 67, 104, 88, 96, 136] for several years. However, there are only few adaptive PMS which support both kinds of changes in one system [65, 136]. All of them have in common that once an instance has been individually modified (i.e., it possesses an instance-specific process schema), it cannot longer benefit from process type changes; i.e., changes of the type schema they were originally created from. In [65], changes of process type schemes as well as changes of single process instances are supported but they are not considered in interplay. In $WASA_2$ [136], for example, a change of a single instance is carried out by deriving a new schema version to which the instance is migrated. Afterwards, this instance is excluded from further adaptations of its original schema version at the process type level. However, doing so is not sufficient in many cases, especially in conjunction with long-running processes as, for example, patient treatment processes in a clinical environment. In such environments ad hoc modifications of a single instances become necessary, e.g., due to a life–threading situation. However, it is also usual that the treatment process itself has to be adapted (on type level), e.g., if new documentation guidelines come into effect. In fact, it must be possible to propagate such process schema changes at the type level to biased instances as well. Therefore, in this chapter, we establish a (formal) framework which enables us to adequately support the **interplay** between *concurrent*[1] process type and instance changes.

The remainder of this chapter is organized as follows: We start with a discussion of challenges arising in conjunction with concurrent process changes in Section 5.1. In Section 5.2 we provide a formal framework which enables us to differentiate between disjoint and overlapping changes. This is followed by providing a compliance criterion for process instances with disjoint bias (cf.

[1]The considered process instance changes take place before the process type change occurs. However, we denote these process type and process instance changes as *concurrent* since they work on the same underlying process schema.

Section 5.3), structural conflicts test (cf. Section 5.4) and a discussion of migrating process instances with disjoint bias to a changed process type schema (cf. Section 5.5). We close this chapter with a summary of the presented results (cf. Section 5.6).

5.1 Challenges for Migrating Biased Process Instances

To give an idea which challenges arise in conjunction with concurrent process type and process instance changes we first provide some illustrating examples.

Example 5.1 (Basic Scenario: Migrating Unbiased And Biased Instances): Consider the example from Figure 5.1 where type change Δ_T serially inserts activity X between subsequent activities C and D. Assume that we want to propagate Δ_T to running process instances I_1 and I_2. I_1 is an unbiased instance whereas I_2 is a biased instance with instance–specific change (bias) Δ_{I_2}. Δ_{I_2} has inserted activity Y between D and E resulting in instance schema $S_{I_2} := S + \Delta_{I_2}$. For both instances it is necessary to check the state–related compliance conditions stated in Section 4.3.2. For unbiased instances performing these state–related compliance checks are already sufficient. For biased instances (like I_2), additionally we have to ensure that the resulting instance–specific schema $(S + \Delta_{I_2}) + \Delta_T$ obeys the correctness constraints set out for WSM Nets (cf. Definition 2, Section 3.1.1). Intuitively, if the resulting instance–specific schema after applying process type and process instance changes is structurally correct no structural conflicts between these changes have been occured. An example for such structural conflicts are deadlock–causing cycles which may appear if sync edges are inserted by both, process type and process instance changes in an uncontrolled manner.

The migration of unbiased instance I_2 is depicted in Figure 5.1. It can be done without any problems because neither structural nor state–related conflicts have occured. However, such unproblematic instance migrations are rather scarce. Often conflicts between process type and process instance changes occur as the following example shows:

Example 5.2.a (Overlapping Process Type and Process Instance Changes) Consider Figure 5.2.a where type change Δ_T and instance change Δ_I both delete same activity D from original process schema S. In general, we call such changes having the same effects on the original process schema *overlapping changes*. But what is the challenge when dealing with overlapping process changes? Remember that, in general, our goal is to propagate process type changes to as many running process instances as possible. Therefore we also want to propagate Δ_T to process instance I with overlapping instance–specific change Δ_T (bias) as depicted in Figure 5.2.a. However, doing so is not possible in the given case since activity D, which is to be deleted by Δ_T is no longer present in $S_I := S + \Delta_I$. Consequently, Δ_T cannot be applied to S_I.

We give a second example to show which further conflicts may arise in conjunction with concurrent process type and process instance changes.

Example 5.2.b (Deadlock Causing Cycle As Structural Conflict) Have a look at Figure 5.2.b where this time type change Δ_T inserts two new activities X and Y into different branches of a

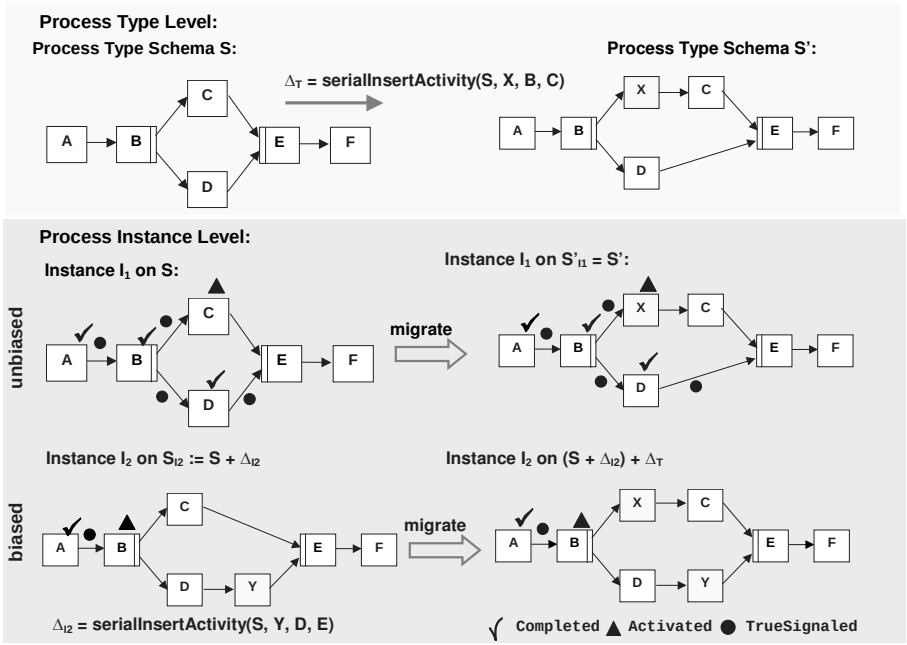


Figure 5.1: Migrating Biased Process Instances (Example)

parallel branching and a sync link (X, Y) between them. Prior to Δ_T instance–specific change Δ_I has inserted a sync link (F, C) between activities F and C into S. Obviously, Δ_T and Δ_I work on different elements of the original process type schema S and therefore have totally different effects on it. For such *disjoint* process type and process instance changes it is always possible to apply Δ_T to S_I resulting in instance–specific schema $(S + \Delta_I) + \Delta_T$ for I on S'. However, $(S + \Delta_I) + \Delta_T$ may not be a correct WSM Net (cf. Definition 2) as it is the case for the given example. The reason is that $(S + \Delta_I) + \Delta_T$ contains a structural inconsistency, namely the deadlock–causing cycle $X \rightarrow Y \rightarrow E \rightarrow F \rightarrow C \rightarrow D \rightarrow X$.

As illustrated by the above examples different conflicts between concurrent process type and process instance changes may occur. Either they (partially) have the same effects on the underlying process schema or they cause structural inconsistencies within the resulting instance–specific schema. In detail we have to deal with the following challenges:

1. **Disjoint $\longleftrightarrow$ Overlapping Changes:** As it can be seen from Examples 5.2.a and 5.2.b there are process type and instance changes which *overlap* (i.e., which have (partially) the same effects on original schema S) and changes which are *disjoint* (i.e., which have totally different effects on original schema S). Both kinds of concurrent changes, overlapping and disjoint changes, require a totally different handling. The reason is that for overlapping changes it may be even not possible to apply type change Δ_T to instance–specific schema $S_I = S + \Delta_I$ (cf. Example 5.2.a) whereas for disjoint changes this is always possible (cf.

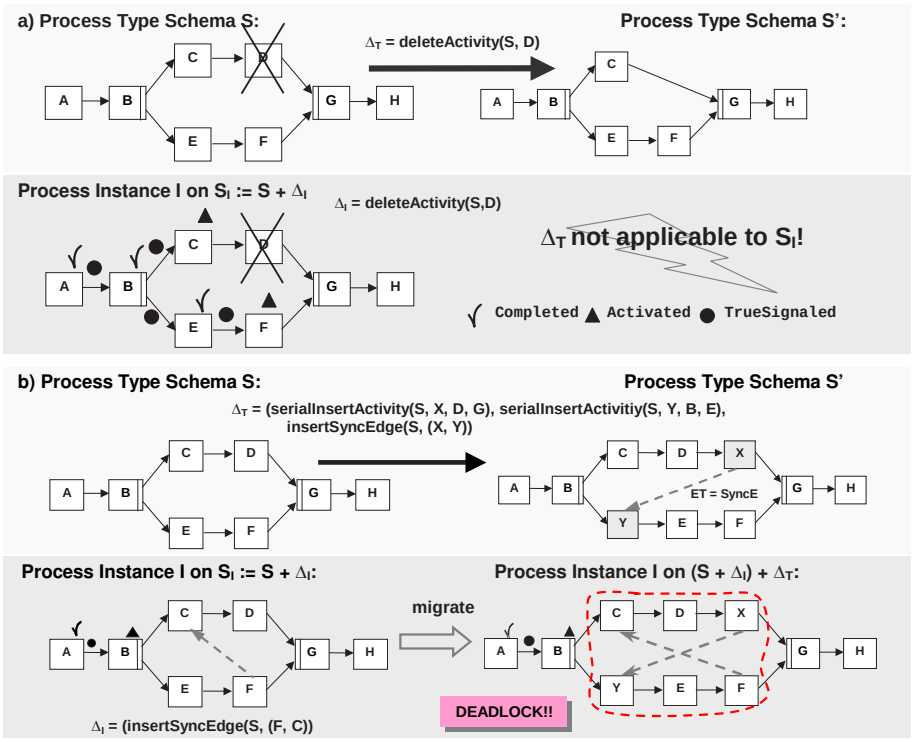


Figure 5.2: Concurrent Process Type and Instance Changes (Example)

Criterion 8, Section 5.3) but the resulting schema $(S + \Delta_I) + \Delta_I$ may contain structural errors like deadlock–causing cycles (cf. Example 5.2.b).

In order to chose the right *migration strategy* for dealing with disjoint changes on the one hand and overlapping changes on the other hand, we first must be able to detect whether process type and process instance changes are disjoint or overlap. For disjoint changes, for example, we apply the so called *standard migration strategy* whereas for overlapping changes we choose one of the *advanced migration strategies* (cf. Figure 5.3). To be able to detect whether concurrent changes are disjoint or overlap it is necessary to find adequate formal definitions. These definitions, in turn, provide an adequate formal underpinning for further considerations.

2. **Correctness:** As shown by Example 5.1.b, propagating process type change Δ_T to instances with (disjoint) instance–specific change may lead to structural inconcistencies within the resulting instance–specific schemes. Examples are deadlock causing cycles (cf. Example 5.2.b) or missing input data for activities executed in the sequel. Therefore, it is indispensable to find a correctness criterion which maintains the state–related claims of Criterion 7 but also incorporates structural correctness of resulting instance–specific schemes after propagating a process type change. In the following, therefore we will pro-

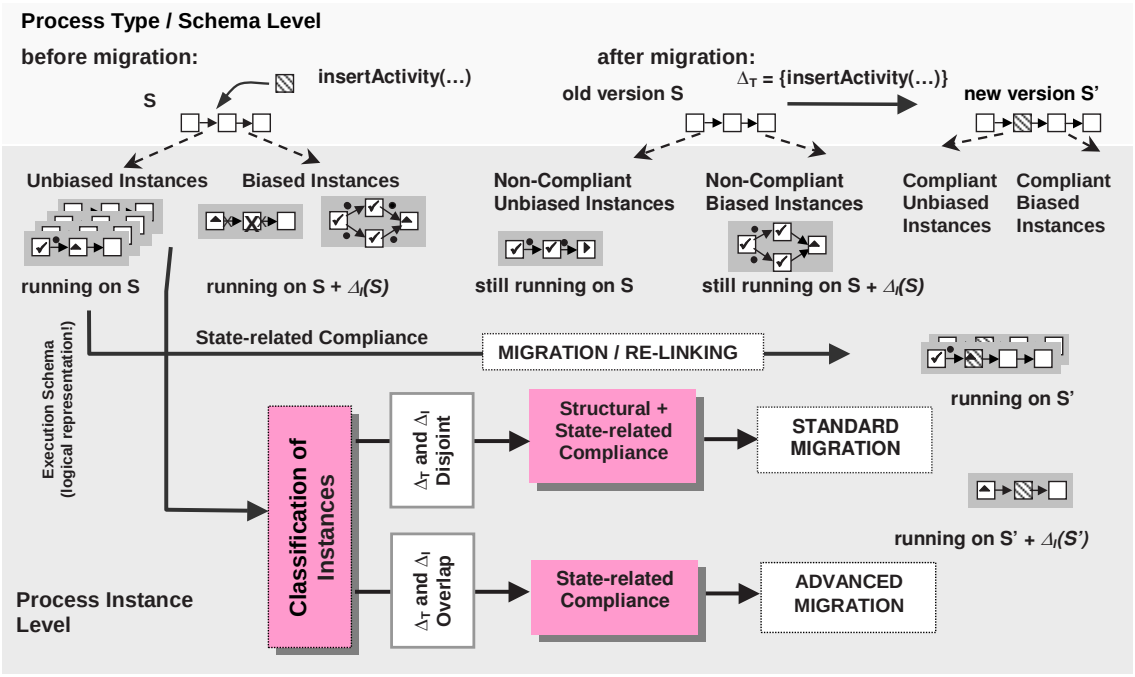


Figure 5.3: Migration Process at a Glance

vide an appropriate generalization of Criterion 7 (cf. Section 4.2).

3. **Efficient Structural Compliance Checks:** In Section 4.3.2 precise conditions to efficiently decide on state–related compliance have been elaborated. For biased process instances an appropriate compliance criterion must ensure state–related correctness further on but must also incorporate structural correctness. Since checking structural correctness increases complexity of ensuring compliance we urgently need extended checking routines to stay efficient for biased process instances as well. Therefore in the following we present respective *structural conflict tests* which quickly indicate whether there will be structural inconsistencies within resulting instance–specific schemes after propagation of a process type change or not.

4. **Instance Adaptations:** If compliant unbiased process instances are migrated to changed process type schema S' marking adaptations become necessary (cf. Section 4.4). Of course, respective state adaptations are required for compliant, biased process instances I as well when migrating them to changed type schema $S' = S + \Delta_T$. In addition, we actually have to "produce" instance–specific schema $(S + \Delta_I) + \Delta_T$ for each compliant and biased process instance I in order to apply Δ_T on $S_I = S + \Delta_I$. However, doing so may turn out to be very expensive, especially at the presence of a large number of biased process instances. Furthermore, from $(S + \Delta_I) + \Delta_T$ we cannot derive which instance–specific change $\Delta_I(S')$ remains for I on S', i.e., what the bias of I on S' is. Therefore, we introduce

commutativity as an important property of concurrent process changes. Commutativity allows us to (logically) transform instance–specific schema $(S + \Delta_I) + \Delta_T$ into instance–specific schema $S'_I := (S + \Delta_T) + \Delta_I = S' + \Delta_I$. Doing so saves us from applying Δ_T to S_I and clearly shows that new instance–specific change $\Delta_I(S')$ of I on S' remains unalteredly the same as $\Delta_I(S)$ for I on S. These results will be summarized within an appropriate migration approach, the so called *standard migration* (cf. Figure 5.3).

5.2 A Formal Framework for Disjoint and Overlapping Changes

In the previous section we have informally introduced the notions of *disjoint* and *overlapping* process changes. In this section, we give formal definitions of these concepts which serve as theoretical underpinning for the following considerations. First of all, we abstract from whether changes are carried out at the type or at the instance level. More precisely, we base our considerations on two arbitrary changes Δ_1 and Δ_2 concurrently applied on the same schema S.

In the following, let S be a (correct) process schema and Δ_1 and Δ_2 two change transactions which transform S into another (correct) process schema S_1 and S_2 respectively (notation: $S_1 := S + \Delta_1$ and $S_2 := S + \Delta_2$). Generally, disjointness and overlapping are two special relations between two changes of the same schema. The challenging question is how to relate changes to each other. This can be either done

1. by directly comparing Δ_1 and Δ_2 or by
2. by correlating their effects on the original schema S. Effects of Δ_1 and Δ_2 on S, in turn, are reflected by the resulting process schemes S_1 and S_2. Consequently, a way to find a relation between changes Δ_1 and Δ_2 is to find a relation between resulting schemes S_1 and S_2.

In the workflow literature several (equivalence) relations between process schemes have been discussed [117, 118, 59, 128]. Aalst and Basten [118], for example, use *branching bisimilarity* as equivalence relation. In this context, Basten has proven that branching bisimiliarity is an equivalence relation indeed [12]. There are several other notions of equivalence between process schemes as pointed out in [59]. In [113], Glabbeek and Goltz provide a very nice classification of semantic equivalences based on the basic notions of *bisimulation* and *trace equivalence*. Another approach to provide semantic equivalence of process schemes is described in [43]. This work supports semantic–preserving transformations to maintain the semantical meaning of a process schema before and after the change.

The challenging question is which equivalence relation from the research area is best suited for determining the relation between concurrent changes on the same process schema. Note that so far the equivalence relations mentioned above have been used to compare process schemes. In this thesis we use them in order to compare the changes leading to different process schemes.

Since it is sufficient to compare the behavior of process schemes we base our further considerations on *trace equivalence* [113, 59, 98]. Claiming graph equivalence would be too restrictive since we want to abstract from silent/null activities within our approach.

Definition 8 (Trace Equivalence Between WSM Nets) *Let S_1 and S_2 be two WSM Nets. S_1 and S_2 are equivalent with respect to their possible traces (formally: $S_1 \equiv_{trace} S_2$) if and only if each execution history Π^{S_1} (cf. Definition 3.1.2) producible on S_1 can be generated on S_2 as well and vice versa.*

Intuitively, two process schemes S_1 and S_2 are trace equivalent if each possible behavior of S_1 (represented by respective execution histories) can be simulated by process schema S_2 and vice versa.

Based on trace equivalence we now introduce an adequate definition for overlapping and disjoint change transactions. Intuitively, two change transactions Δ_1 and Δ_2 overlap if they have (partially) the same effects on the underlying process schema S. This is the case if Δ_1 and Δ_2 manipulate the same – already existing – elements of S or insert the same activities into S. Overlapping effects on already existing elements of a process schema may result from subtractive, order–changing, or attribute–changing operations (cf. Table 3.3). Subtractive changes that overlap may affect the applicability of Δ_1 on S_2 and vice versa (cf. Figure 5.2.a). Overlapping order–changing and attribute–changing operations may mutually *override* the effects of each other.

Example 5.4.a (Overlapping Process Changes Overriding Their Effects) Consider Figure 5.4.a where instance change Δ_I has moved activity E from its current postion to position between C and F and, in contrast, Δ_T moves activity E to position between F and G. Propagating Δ_T on I results in instance–specific schema $(S + \Delta_I) + \Delta_T$ which does not reflect instance–specific change Δ_I any longer. Therefore Δ_T has overrided the effects of Δ_I.

The challenging question is how to deal with the problems arising in conjuntion with two arbitrary overlapping changes Δ_1 and Δ_2 on existing elements of a process schema S; namely (a) destroyed applicability of Δ_1 on S_2 or Δ_2 on S_1 respectively and (b) overriding effects of Δ_1 by Δ_2 or vice versa. These problems can be avoided if change transactions Δ_1 and Δ_2 are *commutative*. More precisely, changes Δ_1 and Δ_2 are commutative if applying Δ_2 on S_1 leads to process schema $S_1 + \Delta_2$ which is trace equivalent to process schema $S_2 + \Delta_1$ resulting from applying Δ_1 on S_2. Assuming commutativity, applicability of Δ_1 on S_2 and applicability of Δ_2 on S_1 are implicitly ensured (claim (a)). Furthermore, if $S_1 + \Delta_2$ and $S_2 + \Delta_1$ must be trace equivalent Δ_2 cannot override the effects of Δ_1 on S and vice versa. Formally, commutativity of change transactions can be defined as follows:

Definition 9 (Commutativity of Changes) *Let S be a (correct) schema and Δ_1 and Δ_2 be two changes transforming S into (correct) schema S_1 and S_2 respectively. We call Δ_1 and Δ_2 commutative if the application of Δ_1 to S_2 and the application of Δ_2 to S_1 result in trace equivalent schemes. Formally:*

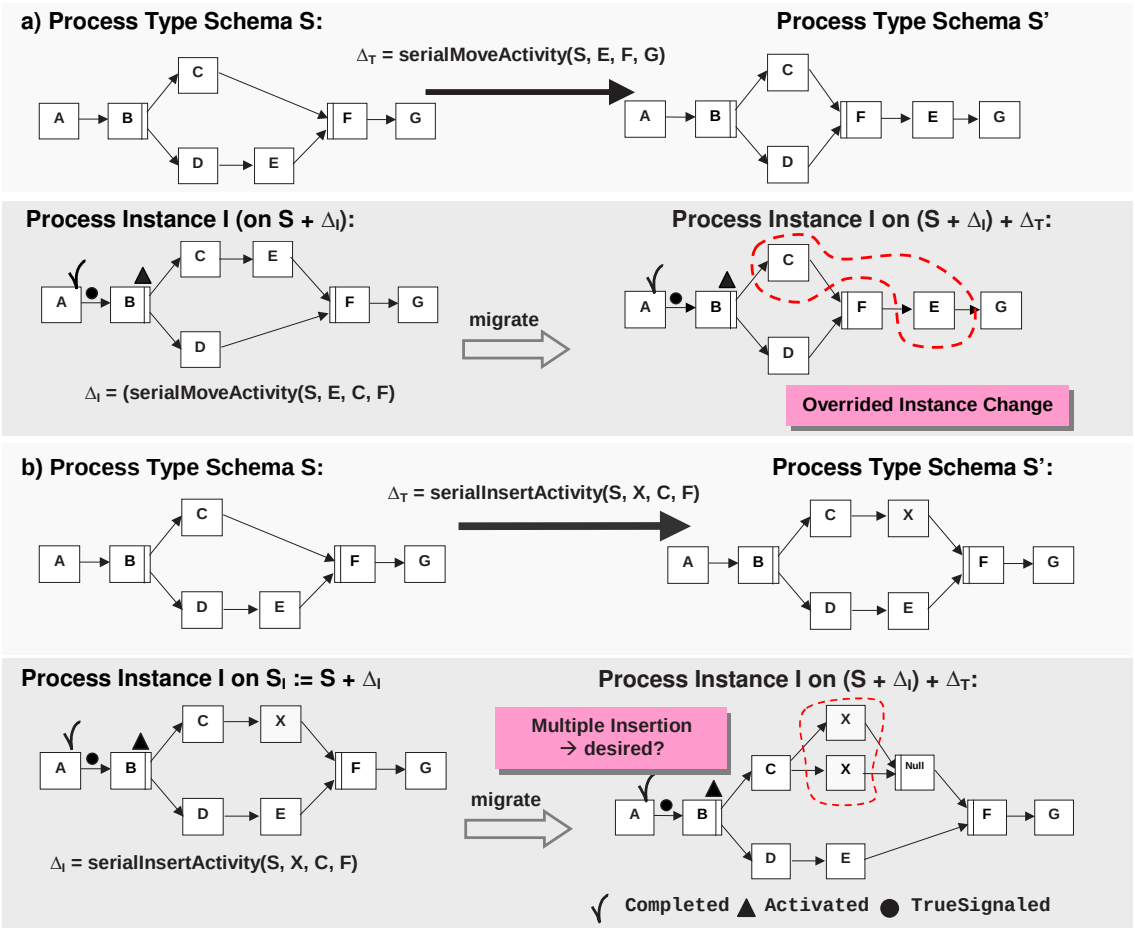


Figure 5.4: Concurrent Changes With Different Overlap Effects

$$\Delta_1, \Delta_2 \textit{ commutative} \iff (S + \Delta_1) + \Delta_2 \equiv_{trace} (S + \Delta_2) + \Delta_1$$

In general, commutativity is an important property in the context of concurrent changes in cooperative applications. In [7], operations commute if the state changes on an object as well as the values returned by the operations are independent of the order in which they are executed. Wäsch and Klas claim that concurrent changes can be correctly applied to complex objects if they are commutative. These changes are then carried out by merging their histories [131].

However, the commutativity property is not strong enough to also cover disjointness of additive changes as, for example, insertions of new activities. In particular, commutativity does not exclude the (undesired) multiple insertion of the same activity as the following example shows:

Example 5.4.b (Multiple Insertion of Activities) Consider Figure 5.4.b where process instance

change Δ_I as well as process type change Δ_T both insert the same new activity X between activities C and F into process schema S. Propagating Δ_T to instance–specific schema S_I results in new instance–specific schema $(S + \Delta_I) + \Delta_T$ which contains newly inserted activity X twice. We call this phenomenon *multiple insertion of activities*. Usually, this would not correspond to the user's intention.

Regarding multiple insertion of activities one can ask under which conditions two activities X_1 and X_2 are considered as equal (otherwise there would be no multiple insertion problem ever). At this point, we abstract from realization details. Informally, two activities X_1 and X_2 are equal (notation: $X_1 = X_2$) if and only if they have the same activity templates and the same semantic identifier. Doing so corresponds to practical scenarios where often a set of prefabricated activities is offered to the user to be plugged into a process template.

Altogether, disjoint changes have to be commutative and their sets of newly inserted activities have to be disjoint. Formally:

Definition 10 (Disjoint and Overlapping Changes) *Let S = (N, D, CtrlE, SyncE, ...) be a WSM Net and Δ_1 and Δ_2 be two changes which transform S into WSM Nets S_1 and S_2 with $S_1 = (N_1, D_1, CtrlE_1, SyncE_1, DataE_1, ...)$ and $S_2 = (N_2, D_2, CtrlE_2, SyncE_2, DataE_2, ...)$. Then:*

1. *We denote Δ_1 and Δ_2 as disjoint (notation: $\Delta_1 \cap \Delta_2 = \emptyset$) iff the following properties hold:*
   - *(a) Δ_1 and Δ_2 are commutative (cf. Def. 9)*
   - *(b) $(N_1 \setminus N) \cap (N_2 \setminus N) = \emptyset$*
2. *We denote Δ_1 and Δ_2 as overlapping (notion: $\Delta_1 \cap \Delta_2 \neq \emptyset$) if they are not disjoint.*

Regarding disjoint changes, Definition 10 explicitly excludes the multiple insertion of same activities. At this point, one may ask how we deal with the multiple insertion of edges. However, this problem is not existent since multiple insertion of same edges is (implicitly) forbidden by Property (1). More precisely, commutativity of changes implies that Δ_1 is applicable to S_2 and vice versa. Applicability of basic and high–level change primitives inserting new control, sync, or data edges (cf. Tables 3.1 + 3.2) already includes the pre–condition that newly inserted edges are not present in original schema S so far [87]. We illustrate this by applying Definition 10 to a concrete scenario where process type change Δ_T and a process instance change Δ_I concurrently work on the same original schema S:

Example 5.5 (Insertion of Same Control Edges:) Consider Figure 5.5 where both, process type change Δ_T and process instance change Δ_I move activity B to position between activities C and D. In the course of this change operation a number of control edges are deleted and inserted. As it can be seen from Figure 5.5, some control edges have been inserted on both process type and instance level. However, in this case, Δ_T cannot be applied to instance–specific schema

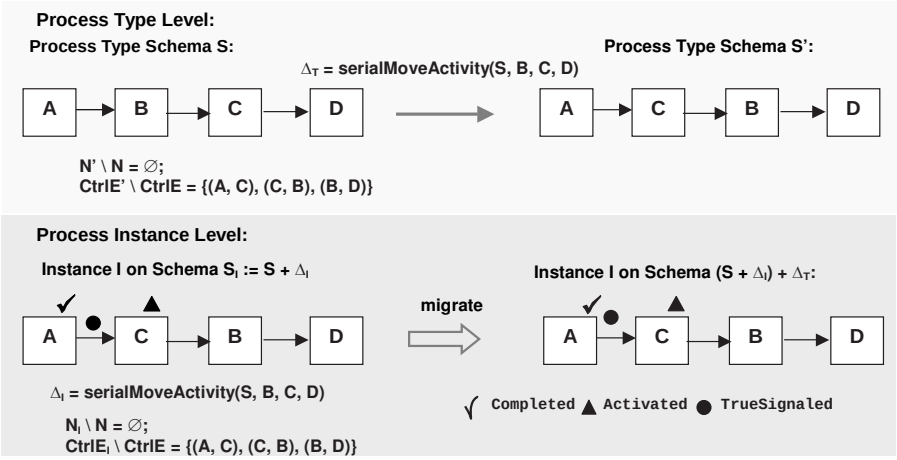


Figure 5.5: Moving Same Activity To Same Target Position (Example)

S_I since, for example, control edge (C, B) is already present in S_I. Consequently, Δ_T and Δ_I are detected as being overlapping changes (cf. Definition 10) what corresponds to the intuitive comprehension that Δ_T and Δ_I have overlapping effects on S.

As summarized in Figure 5.6 we have established a formal framework which enables us to distinguish between disjoint and overlapping process type and process instance changes what is essential to find adequate migration strategies in the following. The challenging question is how to verify Definition 10 in order to quickly group the set of running biased process instances into such instances with disjoint instance–specific change and such instances for which their instance–specific change overlaps the respective type change. However, in anticipation of further results, the distinction into process instances with disjoint and overlapping bias is still not sufficient. In fact, we have to divide process instances with overlapping bias into subclasses along their particular *degree of overlap*. Therefore we put approaches to decide on Definition 10 and to carry out further classifications off. We present all approaches together in Chapter 6.

In the following, the standard migration case, i.e., migrating process instances with disjoint instance–specific change to changed process schema S, with all its facets is presented. For the remainder of this chapter, therefore we assume that process type and process instance changes are disjoint (cf. Figure 5.6).

5.3 A General Correctness Criterion

Let S be a process type schema. Let further I be a process instance which has been started based on S and which has been biased by instance–specific change Δ_I resulting in instance–specific schema $S_I := S + \Delta_I$. Assume that process type change Δ_T transforms S into another

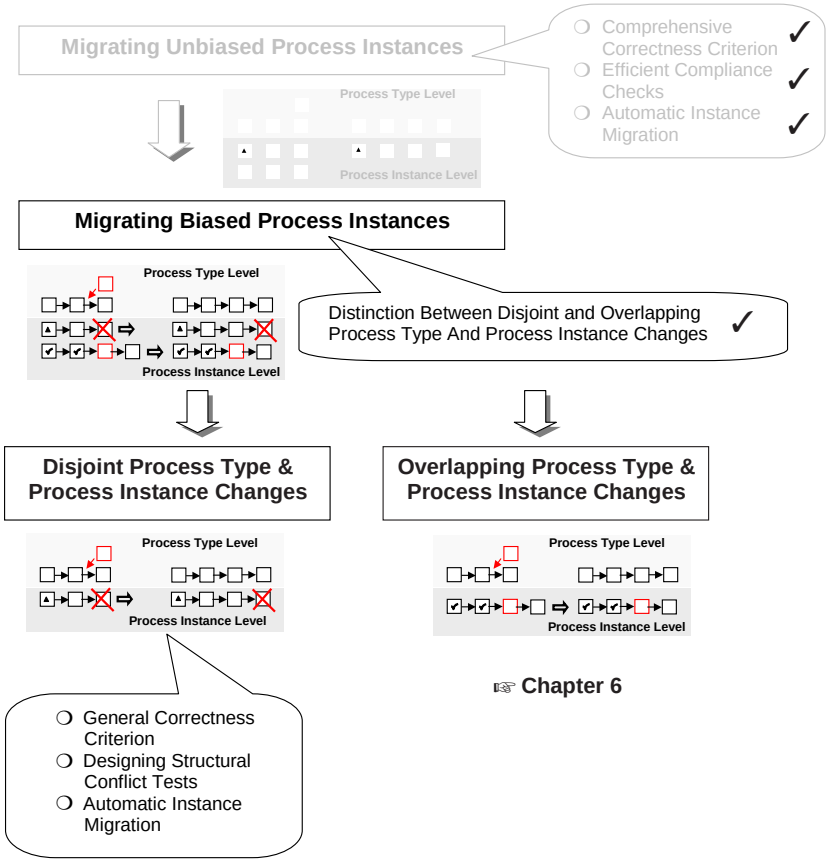


Figure 5.6: Migrating Process Instances with Disjoint Bias

process type schema S' and let Δ_T and Δ_I be disjoint changes according to Definition 10, i.e., $\Delta_T \cap \Delta_I = \emptyset$. The question is how to decide for arbitrary S, Δ_T, and Δ_I whether I is *compliant* with S' or not.

At first, we need an appropriate correctness criterion based on which the above question can be decided. As discussed in Section 5.1 an adequate criterion must incorporate the state–related claims of Correctness Criterion 7 (cf. Section 4.2). However it must also capture the *structural correctness* of the instance–specific schema $(S + \Delta_I) + \Delta_T$ (i.e., the schema resulting when propagating Δ_T to S_I) in order to avoid structural inconcistencies like deadlock–causing cycles or incomplete input data for activity executions in the sequel. Correctness Criterion 8 presented in the following provides an adequate generalization of Correctness Criterion 7 [101, 91]. It comprises an *application–neutral* part (cf. claims 1 and 2 in Correctness Criterion 8) and an *application–dependent* part (cf. claim 3 in Correctness Criterion 8). At this, the application–neutral part is the fundament for being able to manage process type and process instance changes within a PMS and can be decided on without any further information about

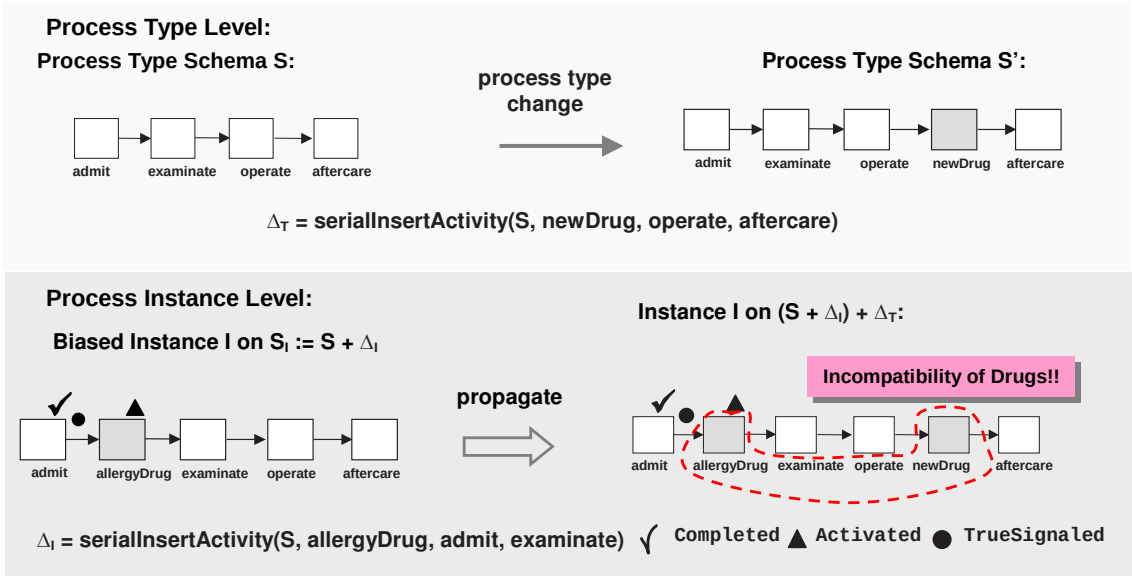


Figure 5.7: Incompatibility of Drugs (Example)

the concerned applications. The application–dependent part requires the deposit of further application information within the PMS as the following example shows:

Example 5.7 (Incompatibility Of Drugs): Look at the process depicted in Figure 5.7. Due to an anaphylactic shock of the respective patient instance I was individually modified by inserting activity `allergyDrug` between activities `admit` and `examinate`. After the introduction of a medical drug `newDrug` the underlying process type schema S is changed by inserting activity `newDrug` between activities `operate` and `aftercare`. Assume that this activity is medically not compatible with `allergyDrug` due to an undesired drug interaction. Propagating this process type change to the already modified instance I ($\Delta_T \cap \Delta_I = \emptyset$) would cause no problems regarding structural and state related conflicts as it can be easily seen from Figure 5.7. But from a semantical point of view, both process type and process instance change are non–compliant due to the mentioned medical incompatibility between the two drugs.

Obviously, to solve problems of this kind we need additional semantical knowledge about the changes to be applied. A short discussion about how to ensure semantical correctness based on application information is given in Chapter 8. The main focus of this work, however, is put on adequate handling of application–neutral aspects (since it provides the basis for any other considerations on adaptive process management).

Correctness Criterion 8 (Compliance For Biased Process Instances) *Let S be a correct process type schema and let $I = (S, \Delta_I, \ldots)$ be a (biased) process instance with current instance execution schema $S_I := S + \Delta_I$. Let further Δ_T be a process type change which trans-*

forms S into another correct process type schema S'. Assume that Δ_T and Δ_I are disjoint changes according to Definition 10, i.e., $\Delta_T \cap \Delta_I = \emptyset$. Then:

I is compliant with S' $\Longleftrightarrow$

1. **Structural Correctness:** *$(S + \Delta_I) + \Delta_T$ is a correct schema according to the structural correctness constraints set out by the used process meta model (cf. Definition 2 for WSM Nets); i.e., Δ_T can be correctly applied to $S_I = (S + \Delta_I)$.*
2. **State-Related Correctness:** *I is compliant with $(S+\Delta_I)+\Delta_T$ according to Correctness Criterion 7; i.e., the execution history $\Pi^S_{I\,red}$ of I can be produced on $(S + \Delta_I) + \Delta_T$ as well.*
3. **Semantical Correctness:** *Δ_T and Δ_I are semantically conflict–free.*

Correctness Criterion 8 is valid for unbiased as well as for biased process instances. More precisely, it handles unbiased instances as a special case. For an unbiased process instance we obtain $(S + \Delta_I) + \Delta_T = (S + \emptyset) + \Delta_T = S'$, whereby S' is correct according to the assumption of Correctness Criterion 8 and, trivially, Δ_T and $\Delta_I = \emptyset$ are semantically conflict–free. Consequently, only state-related correctness has to be checked what exactly corresponds to Correctness Criterion 7 (cf. Section 4.2), i.e., an unbiased instance I is compliant with a changed schema S' if its previous execution trace on S (with eliminated log entries of non–relevant loop iterations) is also a possible execution trace on S' (cf. Requirement 2 in Correctness Criterion 8).

Regarding Requirement 1 of Correctness Criterion 8 we first have to ensure that Δ_T is actually applicable to instance-specific execution schema $S_I := S + \Delta_I$. Obviously, this pre–condition is fulfilled since Δ_T and Δ_I are disjoint according to the assumption of Correctness Criterion 8. The intuitive explanation is that all elements which are necessary to carry out type change Δ_T are still present in S_I. The formal reason lies within Definition 9 since commutativity of Δ_T and Δ_I implies that Δ_T is applicable to S_I.

Therefore target schema $(S + \Delta_I) + \Delta_T$ can be produced. However, the resulting instance–specific schema $(S+\Delta_I)+\Delta_T$ may still contain control and data flow errors like deadlock-causing cycles or missing input data (cf. Example 5.2.b). We therefore must analyze $(S+\Delta_I)+\Delta_T$ with respect to its structural correctness properties like, e.g., the absence of cycles (except loops) (cf. Definition 2).

In the following we want to efficiently test and ensure that $(S+\Delta_I)+\Delta_T$ does not contain any control or data flow errors, i.e., to efficiently test that there are no structural conflicts between process schema and process instance changes (cf. Requirement 2 of Correctness Criterion 8). Obviously, an appropriate approach for this problem has to work for a large number of biased process instances as well. A naive solution would be to simulate process type change Δ_T on each instance–specific schema S_I; i.e., to materialize instance–specific schema $(S + \Delta_I) + \Delta_T$ for each (biased) instance I and then to apply respective correctness checks on $(S + \Delta_I) + \Delta_T$. However, this may result in a serious performance penalty caused by the expensive materialization of

$(S + \Delta_I) + \Delta_T$ on the one hand and the subsequent complex control and data flow correctness checks on $(S + \Delta_I) + \Delta_T$ on the other hand. Again note that these two steps would have to be applied to each biased instance to be migrated.

Therefore, in the following section, we show how expensive correctness tests (based on materialized schemes $(S + \Delta_I) + \Delta_T$ for each biased instance I) can be avoided. The key idea behind our approach is to detect potential control and data flow errors in $S + (\Delta_I) + \Delta_T$ solely based on the applied changes Δ_T and Δ_I, and the original schema S. More precisely, we elaborate quickly checkable conflict tests by exploiting the semantics of the applied changes Δ_T and Δ_I. Respective conflict tests either yield that there would be definitely no control or data flow error in schema $(S + \Delta_I) + \Delta_T$ or they indicate that a possible structural conflict between Δ_T and Δ_I (potentially leading to such an error) may occur.

To check state–related correctness (cf. Requirement 2 of Correctness Criterion 8) the precise compliance conditions developed in Section 4.3.2 can be used further on. More precisely, they can be applied for checking state-related compliance of unbiased as well as of biased process instances with a modified process type schema. The reason is that the application of the compliance conditions is independent of the particular instance–specific schemes.

5.4 On Designing Structural Conflict Tests

In this section, we develop simple but effective tests for detecting potential conflicts between concurrently applied control and/or data flow changes in order to satisfy Requirement 2 of Criterion 8. At this, a first important result is that the number and kind of possible structural conflicts between process type and process instance changes (which may result in incorrect WSM Nets; cf. Definition 2) depend on the kind of applied change operations. Generally, the definition of changes should be based on a set of change operations with precise semantics. In principle, there are different possibilities for the design of this set of change operations. They range from simple node/edge operations (change primitives) to change operations on a high semantical level. In this section we show how the definition of concurrently applied changes influences the handling of structural conflicts between them (cf. Table 5.1).

In the following sections, we present efficient *conflict tests* which rule out the different possible structural conflicts between type change Δ_T and instance change Δ_I concurrently applied to a process schema S. The motivation behind is the following: To exclude structural conflicts between Δ_T and Δ_I reflected in structural inconsistencies within schema $(S + \Delta_I) + \Delta_T$ a solution would be to materialize $(S + \Delta_I) + \Delta_T$ and to carry out possibly expensive correctness checks. The other possibility is to provide conflict tests based on the existing information, i.e., based on changes Δ_T and Δ_I and process schema S. Doing so we prohibit materialization of $(S + \Delta_I) + \Delta_T$ and expensive correctness checks.

The remainder of this section is organized as follows: We start with control and data flow conflicts in conjunction with the application of change primitives (cf. Section 5.4.1). Section

Table 5.1: Possible Structural Conflicts Between Concurrently Applied Changes

Semantical Level Of Applied Changes	**Possible Structural Conflicts** (Acc. To Contraints For WSM–Nets (cf. Def. 2))
1) Change Primitives op_1, op_2 (cf. Table 3.1)	
• $op_i \in$ {addCtrlEdges(S, ...), deleteCtrlEdges(S, ...) addNodes(S, ...)}	→ Isolated Activities (cf. Constraint 2) → Overlapping Control Blocks (cf. Constraint 2)
• $op_i \in$ {addDataEdges(S, ...), deleteDataEdges(S, ...)}	→ Missing Input Data For Activity Execution (cf. Constraint 2) → Lost Updates on Data Elements (cf. Constraint 2)
2) Basic Change Operations op_1, op_2 (cf. Table 3.2)	
• $op_i \in$ {insertSyncEdge(S, ...)}	→ Deadlock–Causing Cycles (cf. Constraint 2)
3) Change Transactions Δ_1, Δ_2	
• $\Delta_i \in$ {insertSyncEdge(S, ...), serialInsertActivity(S, ...))}	→ Deadlock–Causing Cycles (cf. Constraint 2)
• Δ_i = insertBetweenNodeSets(S, ...)	→ Deadlock–Causing Cycles (cf. Constraint 2)
• Δ_i = insertLoopEdge(S, ...)	→ Overlapping Loop Blocks (cf. Constraint 2) → Sync Links Crossing Loop Block Boundaries (cf. Constraint 2)
(• Δ_i arbitrarily choosen	→ Combination Of Possible Structural Conflicts)

5.4.2 presents structural conflict tests for the application of basic change operations and Section 5.4.3 provides tests for the application of high–level change operations and change transactions.

5.4.1 Structural Conflicts When Applying Change Primitives

The concurrent application of change primitives (e.g., to insert/delete control and data egdes) to the same process schema may lead to several structural conflicts. Basically we can divide them into control flow conflicts (e.g., isolated activities) and data flow conflicts (e.g., missing input data for activity executions) (cf. Table 5.1).

The application of change primitives to a process schema does not (automatically) ensure correctness of the changed schema (cf. Section 3.2). Therefore change primitives are applied and afterwards schema correctness is checked whereas change operations guarantee schema correctness by formal pre–conditions. When applying, for example, change operation $serialInsertActivity(S, X, src, dest)$ neither newly inserted activity X nor context activities src and $dest$ can be isolated by this change since the applied high–level primitive cares for the correct embedding of X between src and $dest$. We will illustrate later on that this may not be

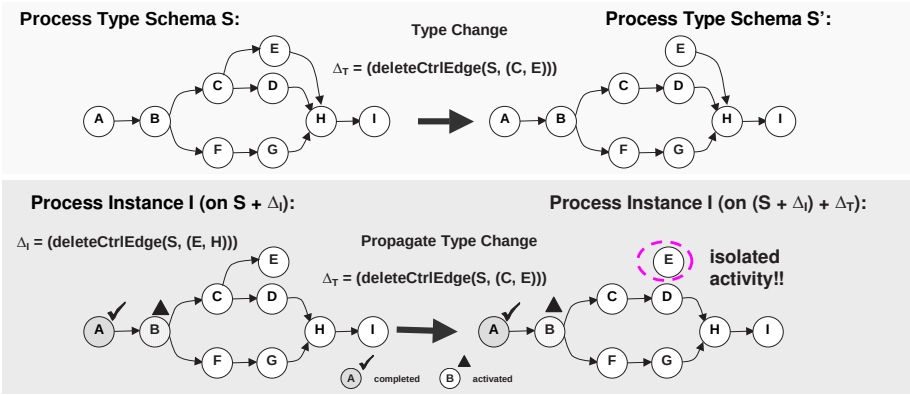


Figure 5.8: Activity Net Containing Isolated Activity Node

the case when executing the same change by applying change primitives. Therefore control flow primitives like adding or deleting control edges are often not (directly) offered to the user in order to design or change process schemes. Instead change operations with formal pre– and post–conditions are provided (e.g., in WIDE [26], TRAMs [67], and ADEPT [87, 88]). Due to this fact it is not stringently necessary to provide explicit conflict tests in conjunction with control flow primitives in the following. However, we present some illustrating examples to alert possible control flow conflicts to the reader. In particular, to establish a bridge to other process meta models we present an example for change primitives applied on Activity Nets [73] leading to isolated activity nodes [101]. It is a reasonable constraint for Activity Nets to forbid isolated activity nodes due to their totally unclear execution semantics.

Control Flow Primitives:[2] As it can be seen from Table 5.1 structural conflicts like isolated activities and overlapping control blocks may occur after propagating Δ_T to instance–specific schema S_I in an uncontrolled manner. Our first example shows the presence of an isolated activity node within an Activity Net:

Example 5.8 (Activity Net Containing Isolated Activity Node): An example for an Activity Net containing an isolated activity node after an uncontrolled application of concurrent process type and process instance changes is depicted in Fig. 5.8: Δ_T deletes control link (C, E) whereas Δ_I has already deleted control link (E, H) for instance I. The uncontrolled propagation of Δ_T to instance-specific schema $S_I = S + \Delta_I$ results in target schema $(S + \Delta_I) + \Delta_T$ which contains isolated activity node E.

Our next example again refers to WSM Nets (cf. Section 3.1.1). It shows overlapping control blocks resulting when propagating type change Δ_T to instance–specific schema S_I in an uncontrolled manner.

Example 5.9 (Overlapping Control Blocks): Consider Figure 5.9.a where process type change

[2] $\Delta_T, \Delta_I \in$ {addCtrlEdges(S, ...), deleteCtrlEdges(S, ...), addNodes(S, ...)}

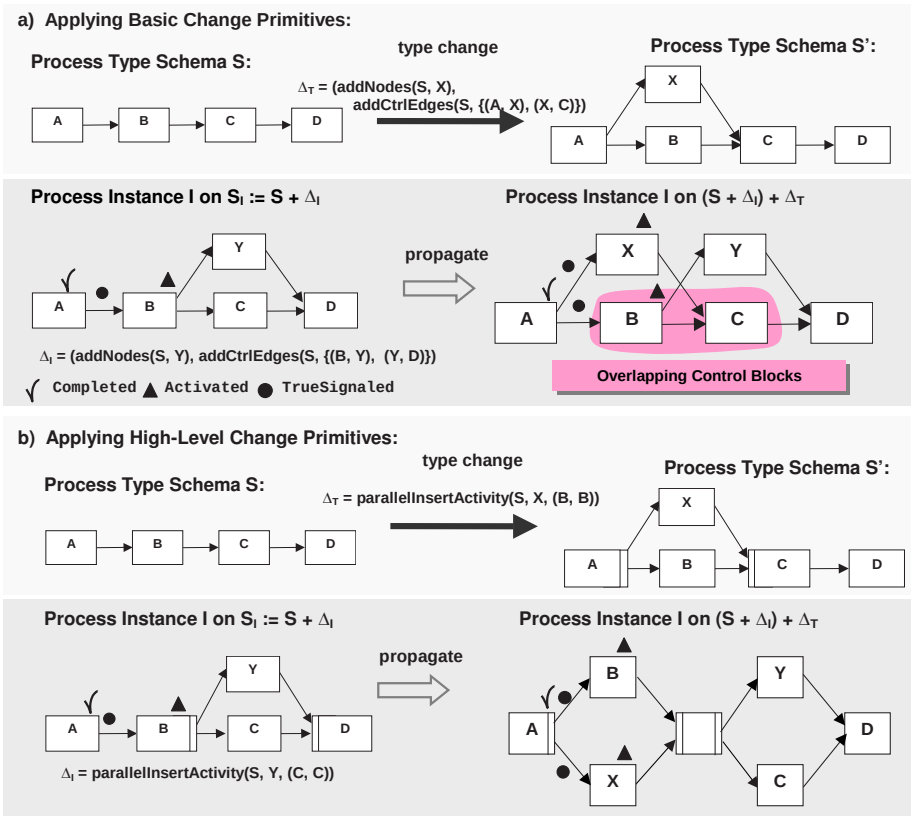


Figure 5.9: Conflicting Control Flow Primitives

Δ_T and process instance change Δ_I both insert new activities X and Y respectively by applying change primitives. Propagating Δ_T to instance–specific schema S_I results in overlapping control blocks (A, C) and (B, D) what offends against the block–structure constraint for WSM Nets (cf. Definition 2). In contrast, if Δ_T and Δ_I execute the insertions by applying basic change operation $parallelInsertActivity(S, \ldots)$ (cf. Section 3.2) no overlapping control blocks are contained in resulting instance–specific schema $(S + \Delta_I) + \Delta_T$ (cf. Figure 5.9.b).

Example 5.9 shows that control flow conflicts in conjunction with change primitives can be prohibited when using the respective change operations instead.

Data Flow Primitives:[3] Though, in most cases, data flow changes are only carried out accompanying control flow changes within a change transaction (cf. Section 4.3.2.4) ADEPT allows the seperated application of data flow primitives in order to, for example, correct data flow modeling errors. Therefore, we do not restrict our considerations to illustrating examples but, furthermore, present easy to check data flow conflict tests in the following. Note that data

[3] $\Delta_T, \Delta_I \in$ {addDataEdges(S, ...), deleteDataEdges(S, ...)}

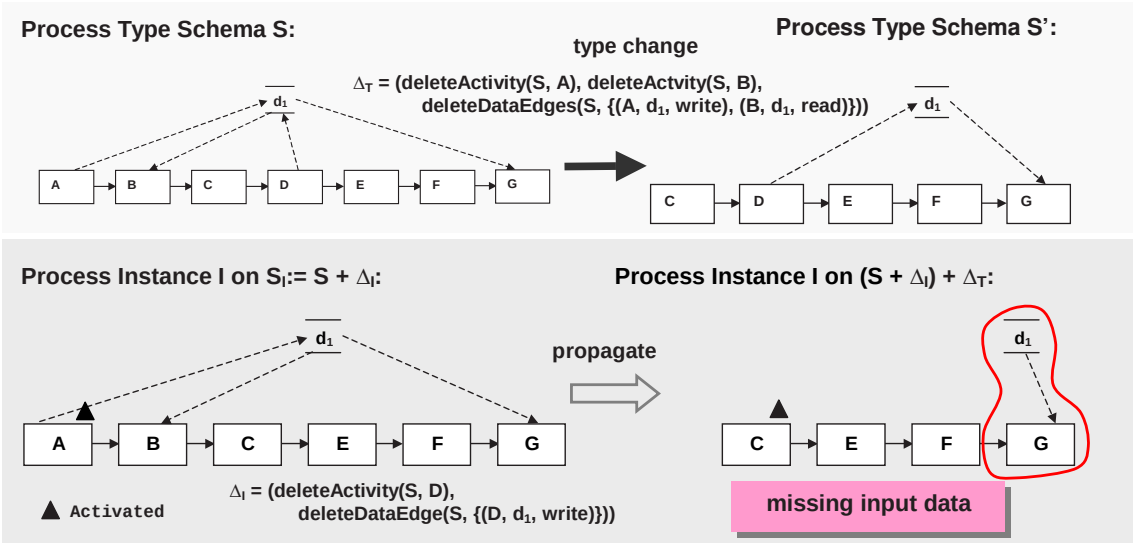


Figure 5.10: Deleting All Necessary Write Accesses on Instance Data (Example)

flow conflict tests have to be carried out in order to avoid data flow inconsistencies after applying concurrent change transactions.

Our data flow constraints from Definition 2 forbid activities with missing input data and lost updates on data elements. Respective inconsistencies may occur within $(S + \Delta_I) + \Delta_T$ if process instance and process type change both delete write data links on the same data element read by other activities in the sequel as the following example shows:

Example 5.10 (Missing Input Data): An example is depicted in Figure 5.10 where Δ_T and Δ_I delete write data links related to the same data element d_1 which causes missing input data of activity G in $(S + \Delta_I) + \Delta_T$.

However, materialization of $(S+\Delta_I)+\Delta_T$ should be avoided. Note that the detection of data flow conflicts based on materialized schema $(S+\Delta_I)+\Delta_T$ may result in exponential complexity [87]. Therefore, in the following we provide a formal proposition to exclude data flow errors for $(S + \Delta_I) + \Delta_T$ but without materializing $(S + \Delta_I) + \Delta_T$. This proposition allows to exclude conflicts for a magnitude of instances solely on basis of Δ_T and Δ_I.

Proposition 1 (Avoiding Missing Input Data and Lost Updates) *Let S be a WSM Net and I be a biased instance with starting schema S and current execution schema $S_I := S + \Delta_I = (N_I, D_I, NT_I, CtrlE_I, SyncE_I, \ldots)$. Assume that type change Δ_T transforms S into a correct schema S' = (N', D', NT', CtrlE', SyncE', ...).*
Then: Propagating Δ_T to S_I neither results in missing input data nor in lost updates if

$\forall\ mDL_1 = (d_1,\ mode_1,\ [\text{"add"}|\text{"delete"}]) \in \mathcal{AD}(S,\ \Delta_T) \cup \mathcal{DD}(S,\ \Delta_T),$
$\forall\ mDL_2 = (d_2,\ mode_2,\ [\text{"add"}|\text{"delete"}]) \in \mathcal{AD}(S,\ \Delta_I) \cup \mathcal{DD}(S,\ \Delta_I)$
$\quad with\ mode_i \in \{read,\ write\}\ (i = 1,\ 2):$
$\quad\quad d_1 \neq d_2 \vee$
$\quad\quad mode1 = mode2 = read \vee$
$\quad\quad mDL_1 = (d_1,\ \text{"read"},\ \text{"delete"}) \vee mDL_2 = (d_2,\ read,\ \text{"delete"})$ (♣)

whereas

- $\mathcal{AD}(S,\ \Delta_T) := \{(d, mode, \text{"add"})|\exists(X, d, mode) \in DataE' \setminus DataE,\ X \in N,\ mode \in \{read,\ write\}\}$
- $\mathcal{DD}(S,\ \Delta_T) := \{(d, mode, \text{"delete"})\exists(X, d, mode) \in DataE \setminus DataE',\ X \in N,\ mode \in \{read,\ write\}\}$
- $\mathcal{AD}(S,\ \Delta_I) := \{(d, mode, \text{"add"})|\exists(X, d, mode) \in DataE_I \setminus DataE,\ X \in N,\ mode \in \{read,\ write\}\}$
- $\mathcal{DD}(S,\ \Delta_I) := \{(d, mode, \text{"delete"})|\exists(X, d, mode) \in DataE \setminus DataE_I,\ X \in N,\ mode \in \{read,\ write\}\}$

A conflict test based on Proposition 1 checks whether for each pair of data edges newly inserted or deleted by Δ_T and Δ_I (resulting in $(S+\Delta_I)+\Delta_T$) either different data elements are affected or only read data edges are manipulated, or at least one read data edge is deleted. If only data accesses on different data elements are inserted into or deleted from S by Δ_T and Δ_I the correctness of WSM–Nets S' and $(S + \Delta_I) + \Delta_T$ implies the correctness of $(S + \Delta_I) + \Delta_T$. (For further details see Proof C.7 in Appendix C).

Example 5.11 shows two data flow changes which can be concurrently applied to process schema S without causing data flow inconsistencies in the sequel. This result is correctly indicated by evaluating condition (♣) of Proposition 1.

Example 5.11 (Concurrent Data Flow Changes Resulting In Correct Schema): Consider Figure 5.11 where both type change Δ_T and instance change Δ_I insert new activities X and Y respectively with a read data link on data element d. Propagating Δ_T to S_I does not result in a data flow inconsistency witin schema $(S+\Delta_I)+\Delta_T$. This result is correctly reported by a respective test based on Proposition 1 since Δ_T and Δ_I insert a data link on same data element d but both data links have mode *read*.

Now we show an example where the concurrent application of data flow changes Δ_T and Δ_I leads to a data flow inconsistency within resulting schema $(S + \Delta_I) + \Delta_I$:

Example 5.12 (Missing Input Data): In Figure 5.12, Δ_T deletes activities B and F together with data edges *(B, d_2, write)* and *(F, d_2, read)*. At the instance level, Δ_I serially inserts activity Y between activities D and E with a read data link connected to data element d_2 ($\Delta_I = \{addDataEdge(S,\ (Y,\ d_2,\ read))\}$). Obviously, propagating Δ_T to S_I leads to the problem of missing input data regarding newly inserted activity Y. Condition ♣ from Proposition 1 indicates this conflict since both process type and process instance change work on the same data element d_2 by deleting write data links and inserting new read data links for this data element. Such critical instances can be easily detected by a test derived from Proposition 1. Otherwise expensive data flow analyses on $(S + \Delta_I) + \Delta_T$ would become necessary for all cases.

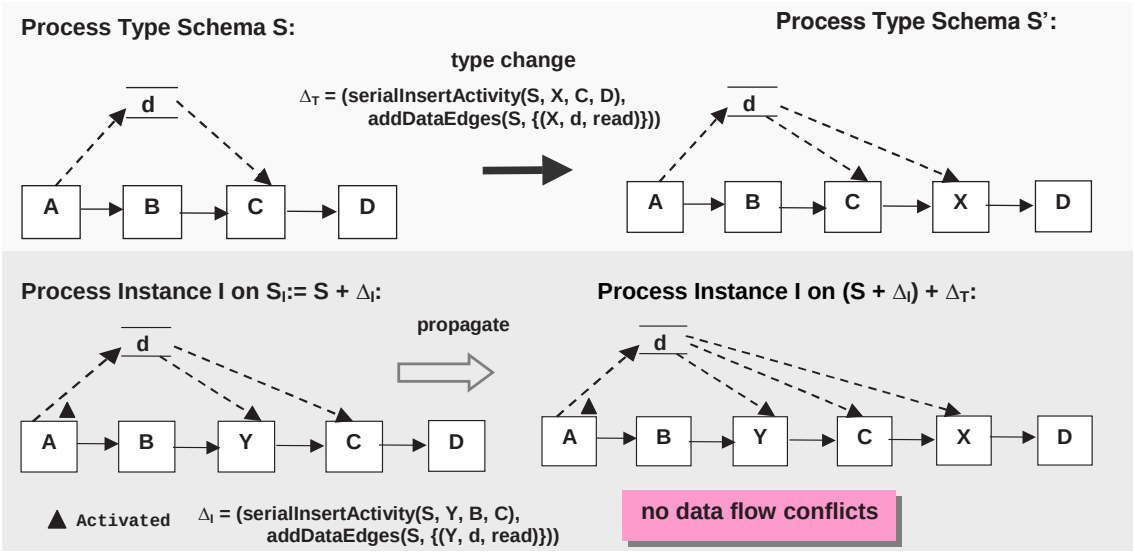


Figure 5.11: Concurrent Data Flow Changes Resulting in Correct Schema

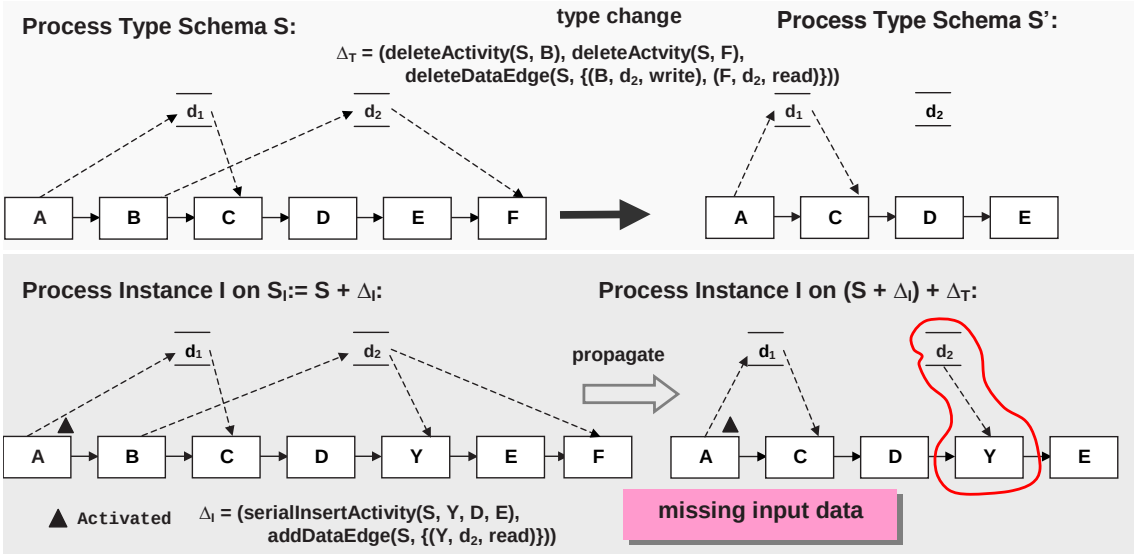


Figure 5.12: Deleting Write Accesses on Data Read by Newly Inserted Activity (Example)

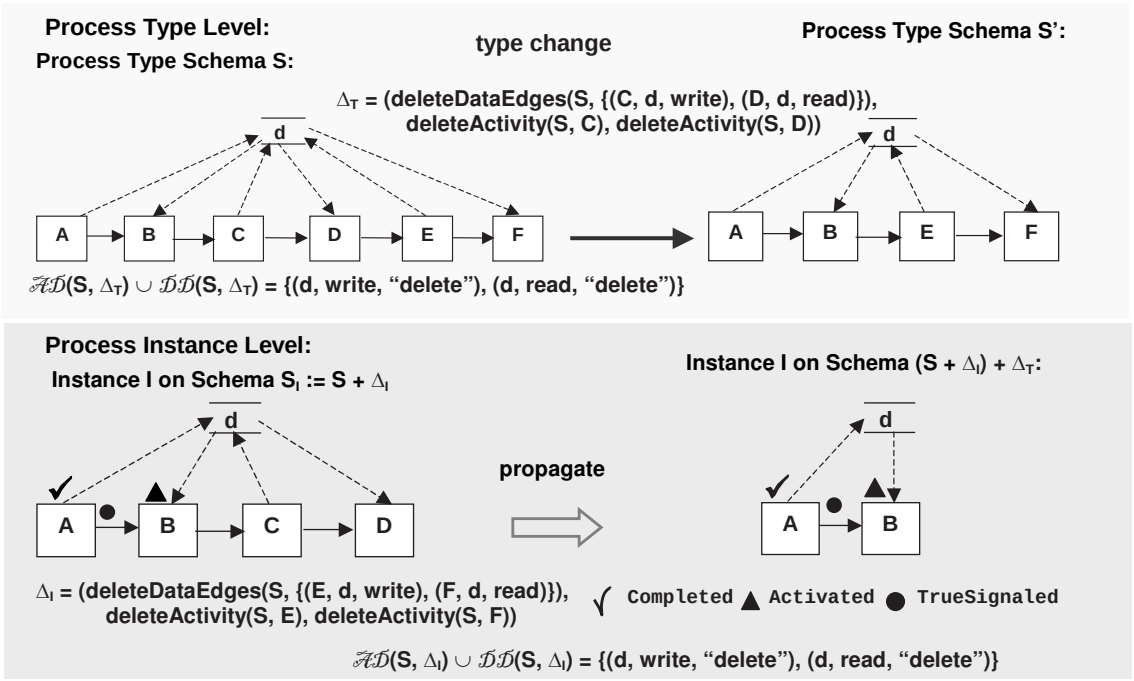


Figure 5.13: Indication of a Potential Data Flow Conflict

A conflict test based on Proposition 1 discovers potential data flow conflicts as, e.g., described by Example 5.12. If a potential data flow conflict is indicated further checks become necessary as the following example shows.

Example 5.13 (Indication Of A Potential Data Flow Conflict): Consider Figure 5.13 where Δ_T and Δ_I both delete a write access on data element d. Due to condition (♣) of Proposition 1 a potential data flow conflict is indicated when propagating Δ_T to instance–specific schema S_I. However, resulting instance–specific schema $(S + \Delta_I) + \Delta_T$ would not contain a data flow inconsistency according to the rules defined in Section 3.1.1.

If a potential data flow conflict is indicated though actually there is none the reason may be that the number of write accesses on a data element d in original process schema S is larger than two. Condition (♣) namely checks whether two write accesses on the same data element are deleted when propagating Δ_T to S_I. Therefore, further checks become necessary. One possibility is to materialize schema $(S + \Delta_I) + \Delta_T$ and to apply the data flow correctness checks presented in [87]. Nevertheless, Proposition 1 and the respective tests work fine in most cases in order to quickly and efficiently detect data flow conflicts concurrent changes at the type and instance level.

5.4.2 Structural Conflicts When Applying Basic Change Operations

According to Table 5.1 a serious problem which may arise from the uncontrolled propagation of a process type change Δ_T to a biased instance (on instance-specific schema $S_I := S + \Delta_I$)[4] is the occurence of deadlock-causing cycles (for an example see Fig. 5.2.b). As mentioned before, a naive solution would be to first materialize target schema $(S+\Delta_I)+\Delta_T$ and then to carry out respective cycle checks on $(S+\Delta_I)+\Delta_T$. Since these materialization and validation steps would have to be applied for each biased instance I, this approach would cause severe performance problems. Thus, our ambition is to perform an appropriate deadlock test based on information given by the process type and instance changes themselves and the original process schema S. A deadlock test satisfying these claims is given in Proposition 2 [101, 91]:

Proposition 2 (Basic Deadlock Prevention) *Let S be a WSM Net and I be a biased instance with starting schema S and execution schema $S_I := S+\Delta_I = (N_I, D_I, CtrlE_I, SyncE_I, \ldots)$. Assume that type change Δ_T transforms S into a correct schema S' = (N', D', NT', CtrlE', SyncE', ...).*

Then: $(S + \Delta_I) + \Delta_T$ does not contain deadlock-causing cycles if the following condition holds:

$$\forall\ (s_1, d_1) \in \mathcal{AS}(S,\ \Delta_T),\ \forall\ (s_2, d_2)\ \mathcal{AS}(S,\ \Delta_I):$$
$$d_1 \notin (pred^*(S,\ s_2) \cup \{s_2\}) \vee d_2 \notin (pred^*(S,\ s_1) \cup \{s_1\})\ (\Psi)$$

whereas

- $\mathcal{AS}(S,\ \Delta_T) := SyncE' \setminus SyncE$
- $\mathcal{AS}(S,\ \Delta_I) := SyncE_I \setminus SyncE$

For a formal proof of Proposition 2 see Proof C.8 in Appendix C.

By simply applying condition Ψ from Proposition 2 we can exclude deadlocks when propagating a type change to a biased instance. Note that condition Ψ is based on the original process type schema S. Consequently, an easy conflict test can be derived which avoids the materialization of any other schema (S_I or $(S + \Delta_I) + \Delta_T$). Based on simple graph algorithms the respective test has complexity $O(n)$.

Example 5.14 (Simple Deadlock–Test): Consider Figure 5.14 where type change Δ_T inserts a sync edge between activities C and D and Δ_I inserts a sync edge between activities E and B. Obviously, propagating Δ_T to instance–specific schema S_I results in incorrect target schema $(S + \Delta_I) + \Delta_T$ containing the deadlock–causing cycle $E \rightarrow B \rightarrow C \rightarrow D \rightarrow E$. Applying Proposition 2 before propagating Δ_T detects this cycle: The set of newly inserted sync edges for Δ_T and Δ_I are $\mathcal{AS}$(S, Δ_T) = {(C, D)} and $\mathcal{AS}$(S, Δ_I) = {(E, B)} respectively. Deploying these sets to condition Ψ yields $D \notin \{A, D, E\} \vee B \notin \{A, B, C\}$ what results in contradiction.

[4] $\Delta_T, \Delta_I \in \{insertSyncEdge(S, \ldots)\}$

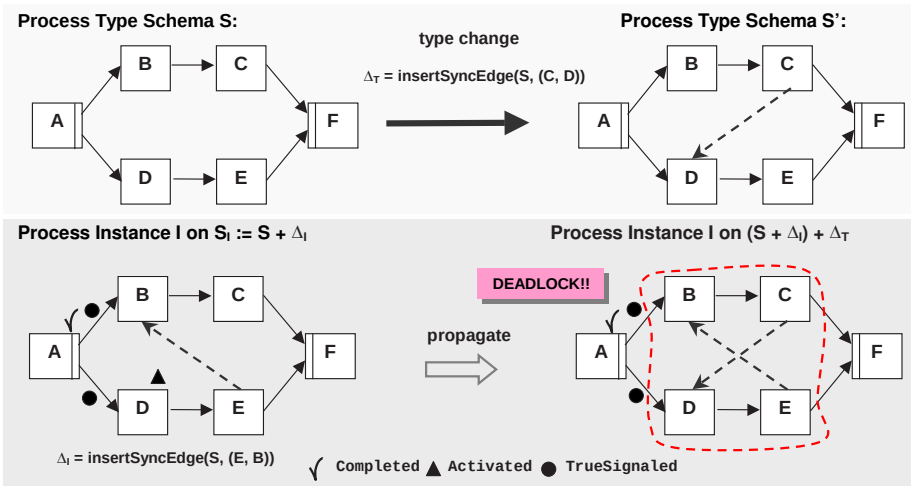


Figure 5.14: Instance–Specific Schema Containing Deadlock–Causing Cycle (Example)

Note that a conflict test based on Proposition 2 would still not work fine if sync links are inserted between also newly inserted activities. The reason is that the newly inserted activities are "unknown" in original schema S and therefore condition (Ψ) of Proposition 2 cannot be evaluated. We provide respective methods to overcome this problem in Section 5.4.3 since the insertion of sync links between newly inserted activities is carried out within a change transaction.

5.4.3 Structural Conflicts When Applying Change Transactions

A change transaction is an ordered series of change primitives and (basic/high–level) change operations. Let, for example, $\Delta_T := (op_1^T, \ldots, op_n^T)$ and $\Delta_I := (op_1^I, \ldots, op_k^I)$ be a process type and a process instance change transaction applied to process schema S. In conjunction with concurrent process type and instance change transactions we precede in three steps:

1. The single operations within a change transaction may be based on each other, i.e. operations are based on elements manipulated by precedent operations. An example is the insertion of a new sync link between activities previously inserted by the same change transaction. We call such operations based on effects of previously applied operations *context–dependent* changes. These context–dependent changes raise totally new and interesting challenges for which we will provide solution approaches in Chapter 6. At this point, our main goal is again to restrict necessary conflict analysis to information given by Δ_T, Δ_I and S even if Δ_T and Δ_I contain context–dependent change operations.

2. We provide tests for structural conflicts arising in conjunction with a special subset of

change transactions – high–level change operations. Strictly speaking, high–level change operations like the insertion between activity sets or the insertion of embracing loop blocks (cf. Table 3.3) are change transactions as well. The only difference to arbitrary change transactions is that high–level change operations are offered to the user in a pre–defined way whereas arbitrary change transactions must be composed by the user.

3. For arbitrary change transactions arbitrary combinations of structural conflicts may arise. We present an illustrating example and shortly explain how to deal with concurrently applied (arbitrary) change transactions.

5.4.3.1 Context–Dependent Change Transactions

Proposition 2 detecting deadlock–causing cycles works fine as long as only basic change operations inserting sync edges are applied. For this case, a respective test would be rated high according to its efficient application to concurrent process type and process instance changes. However, another important quality factor is the number of "uncritical" instances I for which conflicts between process type and instance changes can be definitely excluded. The deadlock test derived from Proposition 2 is a "good" test with respect to efficiency. However, it still scores lower regarding the second quality factor. The reason is that for particular instance changes Δ_I this test indicates conflicts with type change Δ_T although target schema $(S + \Delta_I) + \Delta_T$ will not contain any deadlock causing cycle as the following example shows:

Example 5.15.a (Inserting Sync Edges Not Leading to Deadlock–Causing Cycles): An example is depicted in Figure 5.15: Instance change Δ_I inserts a sync edge between activities C and F (already contained in S) whereas type change Δ_T inserts a sync edge between also newly inserted activites X and Y. From the applied changes we derive $\mathcal{AS}(\mathrm{S}, \Delta_T) = \{(\mathrm{X}, \mathrm{Y})\}$ and $\mathcal{AS}(\mathrm{S}, \Delta_I) = \{(\mathrm{C}, \mathrm{F})\}$ (cf. Proposition 2). The expression yielding from applying condition Ψ (cf. Proposition 2) to these sets cannot be evaluated due to the absence of activites X and Y in S. Consequently, the respective conflict test is unable to exclude the occurence of a deadlock-causing cycle in S although in fact there is none.

At first glance, it seems that we must materialize and validate target schema $(S + \Delta_I) + \Delta_T$ in order to overcome this problem. This approach, however, offends against the efficiency quality factor. Fortunately, there is another solution avoiding materialization of $(S + \Delta_I) + \Delta_T$ and excluding deadlock conflicts for "uncritical" instances.

Example 5.15.b (Transitive Order Relations Regarding Newly Inserted Sync Edges): Consider again the example given in Figure 5.15: Here we cannot evaluate condition Ψ based on sync edge (X, Y) since its source and destination activities have been newly inserted by Δ_T as well. However, the insertion of sync edge (X, Y) does not only set out the direct order relation "X before Y" but also, for example, the transitive order relation "D before E". Since D and E are present in S we are able to verify condition Ψ for a respective sync edge (D, E). Based on this consideration we try to virtually re-link the actual sync edge (X, Y) to the virtual sync edge (D, E). The challenge is to determine the virtual sync edge(s) based on which condition Ψ can

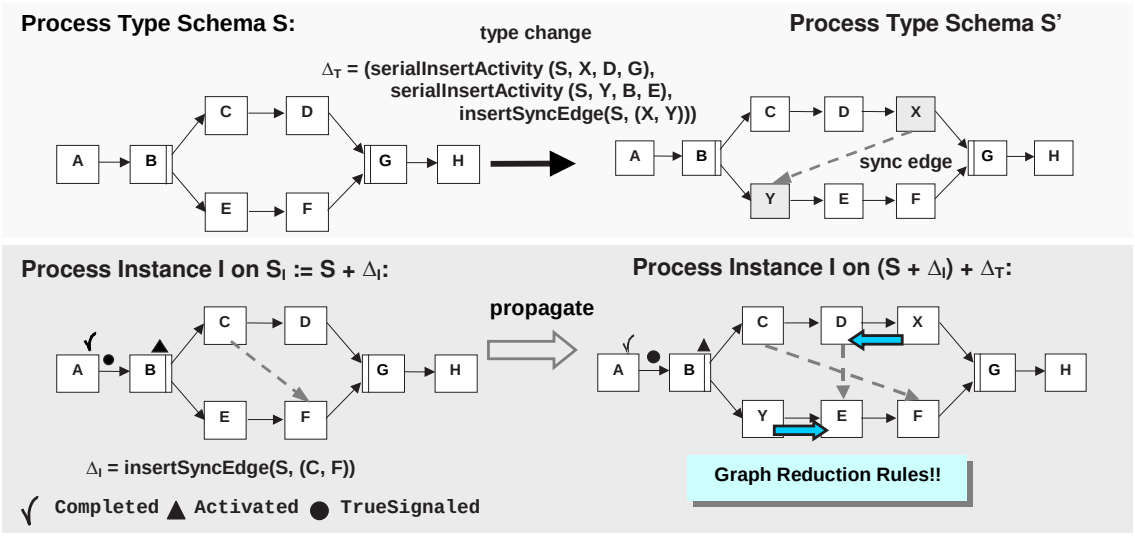


Figure 5.15: Insertion of Sync Edges on Process Type and Instance Level

be evaluated on S. Then – solely based on S – we can determine whether $(S + \Delta_I) + \Delta_T$ will contain a deadlock-causing cycle or not. From Δ_T we know which activities have been inserted and into which context they have been embedded (*insertion context*). For serially inserted activities, for example, the insertion context includes the direct predecessor and successor of the newly inserted activity. For the newly inserted activity X in Fig. 5.15, for example, insertion context (D, G) includes the direct predecessor D of X in S' and for the newly inserted activity Y its insertion context (B, E) includes the direct successor E of Y in S'. Altogether, this is the information we need for determining the virtual sync edges between activities present in S. In our example (cf. Fig. 5.15) we get the virtual sync edge (D, E) instead of (X, Y).

Thus, the idea behind is to first transfer the order relations set out by the newly inserted sync edges to starting schema S by applying "virtual" graph reduction rules and then to apply condition Ψ of Proposition 2 to the reduced graph. The respective graph reduction approach applicable in connection with the composed insertion of activities and sync edges is given in Algorithm 3:

Algorithm 3 (Graph Reduction Rules (Deadlock Prevention)) *Let S = (N, D, NT, CtrlE, SyncE, ...) be a WSM Net and Δ be a change which transforms S into a correct schema S' = (N', D', NT', CtrlE', SyncE', ...). Let further*

- $\mathcal{AS}(S, \Delta) := SyncE' \setminus SyncE$ *and*
- $\mathcal{AA}(S, \Delta) := \{(X, (src, dest)) \mid X \in N', src, dest \in N, X$ *serially inserted between src and dest by* $\Delta\}$.

GraphRed(N, $\mathcal{AS}$(S, Δ), $\mathcal{AA}$(S, Δ)) ⟶ ($\mathcal{AS}_{red}$(S, Δ))

$\mathcal{AS}_{red}$(S, Δ):=∅
forall (src, dest) ∈ $\mathcal{AS}$(S, Δ) **do**
  while src ∉ N **do**
    find (src, (pSrc, sSrc)) ∈ $\mathcal{AA}$(S, Δ);
    src := pSrc;
  done
  while dest ∉ N **do**
    find (dest, (pDest, sDest)) ∈ $\mathcal{AA}$(S, Δ);
    dest := sDest;
  done
  $\mathcal{AS}_{red}$(S, Δ) := $\mathcal{AS}_{red}$(S, Δ) ∪ {(src, dest)}
done

Algorithm 3 works by replacing the source (destination) nodes of the newly inserted sync edges by their direct predecessors (successors) if these nodes have not been present in the original schema S. If several activities are inserted in a row, Algorithm 3 iteratively replaces them by their direct predecessors/successors until we find an adequate predecessor/successor also present in S. In the following Proposition 3, condition Ψ of Proposition 2 is applied based on the graph reduction of Algorithm 3. A deadlock test derived from this proposition fulfills both desired quality factors: It is efficiently applicable based on original schema S and it does not indicate deadlocks for target schema $(S + \Delta_I) + \Delta_T$, if $(S + \Delta_I) + \Delta_T$ is actually deadlock-free.

Proposition 3 (Deadlock Prevention (2)) *Let S be a WSM Net and I be a biased instance with starting schema S and execution schema $S_I := S+\Delta_I = (N_I, D_I, NT, CtrlE_I, SyncE_I, \ldots)$. Assume that type change Δ_T transforms S into another correct schema S' = (N', D', NT', CtrlE', SyncE', ...). Let further $\mathcal{AS}_{red}$(S, Δ_T) and $\mathcal{AS}_{red}$(S, Δ_I) be the sync edge reductions after applying Algorithm 3.*

Then: $(S+\Delta_I)$ + Δ_S does not contain deadlock-causing cycles iff the following condition holds:

$$\forall\ (s_1, d_1) \in \mathcal{AS}_{red}(S, \Delta_T),\ \forall\ (s_2, d_2) \in \mathcal{AS}_{red}(S, \Delta_I):$$
$$d_1 \notin (pred^*(S, s_2) \cup \{s_2\}) \vee d_2 \notin (pred^*(S, s_1) \cup \{s_1\})\ (\Psi)$$

As already mentioned, the reduction rules of Algorithm 3 are necessary in order to transfer the order relations set out by the newly inserted sync edges to the original schema S. As decribed in Proposition 3, we apply Algorithm 3 to the sync edges and activities newly inserted by Δ_T and Δ_I. Based on the resulting sets $\mathcal{AS}_{red}$(S, Δ_T) and $\mathcal{AS}_{red}$(S, Δ_I) condition Ψ from Proposition 2 can be applied to S. Doing so saves us from expensive checks on $(S + \Delta_I) + \Delta_T$.

In general, graph reduction techniques must be applied in order to correctly detect conflicts between context–dependent changes. Further examples are newly inserted read and write ac-

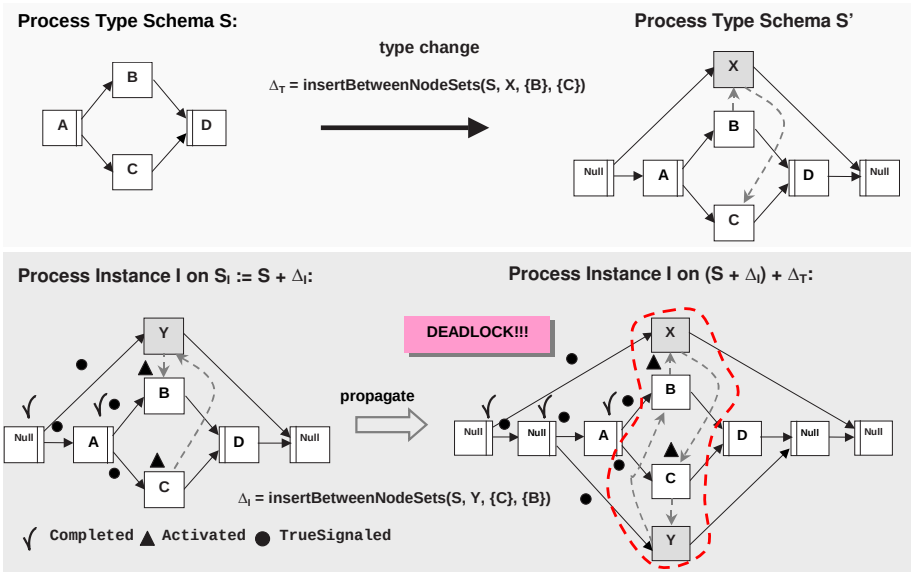


Figure 5.16: Deadlock When Concurrently Applying *insertBetweenNodeSets* Operation

cesses of also newly inserted activities. For these examples similar techniques as Algorithm 3 can be developed in order to to cope with context–dependent data flow changes as well.

We do not discuss structural conflicts arising in conjunction with applying a predefined high–level change operation (cf. Table 3.2).

5.4.3.2 Concurrent Application Of High–Level Change Operations

High–level operation $insertBetweenNodeSets(S, X, M_{before}, M_{after})$ consists of the parallel insertion of activity X and several insertions of sync links. These sync links are necessary to set out the desired order between all activities from M_{before} and X as well as between X and all activities from M_{after}. If several activities are concurrently inserted at process type and process instance level by this operation, cycles between them (via the respective sync links) may emerge as Example 5.16 shows. Note that this is no ADEPT specific conflict. Similar problems may also arise for Activity Nets [73] as, for example, used in IBM Websphere MQ Workflow.

Example 5.16.a (Cycles Due To The Insertion Of Activities Between Node Sets:) Consider Figure 5.16 where type change Δ_T insert activity X between activities B and C whereas instance change Δ_I insert activity Y between activities C and B. Propagating Δ_T to instance–specific schema S_I leads to instance–specific $(S + \Delta_I) + \Delta_T$ which contains the deadlock–causing cycle $X \longrightarrow C \longrightarrow Y \longrightarrow B \longrightarrow X$ via the newly inserted sync egdes.

To detect such deadlock–causing cycles when concurrently applying

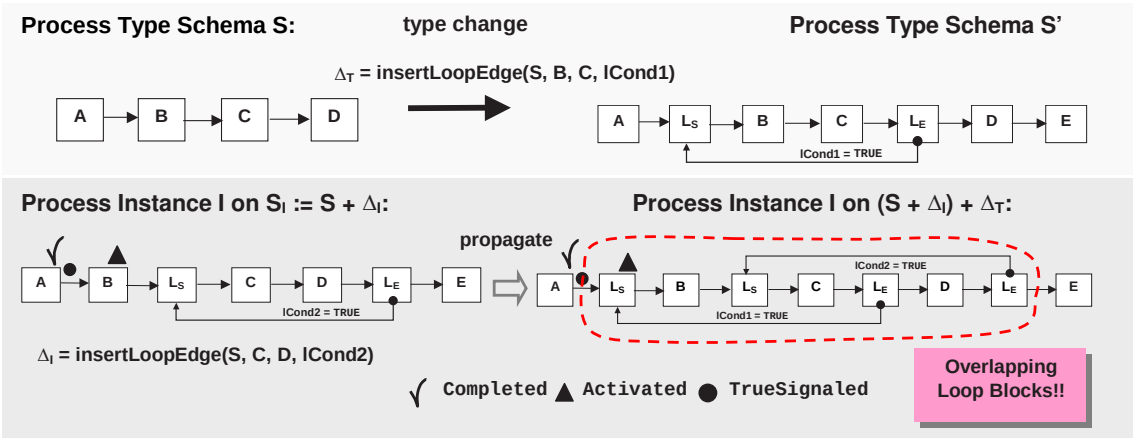


Figure 5.17: Overlapping Loops Blocks

$insertBetweenNodeSet(S, \ldots)$ operations again Algorithm 3 and Proposition 3 can be used based on the set of newly inserted sync edges:

Example 5.16.b (Detecting Deadlock–Causing Cycles): For Δ_T we obtain $\{(B, X), (X, C)\}$ and for Δ_I we obtain $\{(C, Y), (Y, B)\}$ as the set of newly inserted sync egdes. First we apply Algorithm 3 in order to determine the reduced set of (virtual) sync edges based on S. Algorithm 3 yields reduced sets $\{(B, C), (B, C)\}$ for Δ_T and $\{(C, B), (C, B)\}$ for Δ_I. Putting these sets into condition (Ψ) of Proposition 3 it can be easily seen that $B \in \{A, B\}$ and therefore the deadlock–causing cycle within $(S + \Delta_I) + \Delta_T$ is detected.

As opposed to overlapping control blocks which may occur if basic change primitives are concurrently applied in an uncontrolled manner (cf. Section 5.4.1) overlapping loop blocks may also emerge if high–level change operation $insertLoopEdge(S, \ldots)$ is concurrently applied at process type and process instance level. Overlapping control blocks are caused due to the fact that no pre–conditions are checked if the respective basic control flow primitives are applied. Overlapping loop blocks, however, can be considered as "pure" structural conflicts which actually could be not detected until $(S + \Delta_I) + \Delta_T$ is materialized (except we find an appropriate conflict test based on Δ_T, Δ_I and S in the following).

Example 5.17.a (Overlapping Loop Blocks): Please look at Figure 5.17 where Δ_T and Δ_I both insert an embracing loop block inclusive loop backward edge around existing blocks (B, C) and (C, D) respectively. Propagating Δ_T to S_I results in instance–specific schema $(S + \Delta_I) + \Delta_T$ which contains overlapping loop blocks. Obviously, this offends against the correctness constraints for WSM Nets (cf. Definition 2).

How can we determine such overlapping loop blocks within instance–specific schema $(S + \Delta_I) + \Delta_T$ without need to materialize it? Again key to success is to exploit the information given by changes Δ_T and Δ_I. More precisely, based on the information given by Δ_T and Δ_I we can

determine which control blocks shall be newly embraced by loop constructs. Based on S then it can be easily checked whether these newly embraced loop blocks overlap or not. Formally:

Proposition 4 (Overlapping Loop Blocks) *Let S be a WSM Net and I be a biased instance with starting schema S and execution schema $S_I := S+\Delta_I = (N_I, D_I, NT, CtrlE_I, SyncE_I, \ldots)$. Assume that type change Δ_T transforms S into another correct process schema S' = (N', D', NT', CtrlE', SyncE', ...).*

Then: $(S+\Delta_I)+\Delta_S$ does not contain overlapping loop blocks iff the following condition holds:

$$\forall\ (begin_{\Delta_I},\ end_{\Delta_I}) \in EmbracedLoopBlocks(S,\ \Delta_I),\ \forall\ (begin_{\Delta_T},\ end_{\Delta_T}) \in EmbracedLoopBlocks(S,\ \Delta_T):$$
$$begin_{\Delta_T} \in (begin_{\Delta_I},\ end_{\Delta_I}) \Longleftrightarrow end_{\Delta_T} \in (begin_{\Delta_I},\ end_{\Delta_I})^5\ (\bowtie)$$

$$whereas\ EmbracedLoopBlocks(S,\Delta) := \{\{begin_\Delta, end_\Delta\} \cup (succ^*(S, begin_\Delta) \cap pred^*(S, end_\Delta)) \mid \exists insertLoopEdge(S, begin_\Delta, end_\Delta, lC_\Delta) \in \Delta_{\{\}}\}^6$$

To see how a respective conflict test based on Proposition 4 works we provide the following example:

Example 5.17.b (Detecting Overlapping Loop Blocks): For type change Δ_T and instance change Δ_I as depicted in Figure 5.17 we obtain the following sets necessary for checking condition ($\bowtie$) of Proposition 4: $EmbracedLoopBlocks(S, \Delta_T) = \{\{B, C\}\}$ and $EmbracedLoopBlocks(S, \Delta_I) = \{\{C, D\}\}$. As it can be easily seen $B \notin \{C, D\} \wedge C \in \{C, D\}$ holds what violates condition ($\bowtie$). Consequently, $(S + \Delta_I) + \Delta_T$ would contain an overlapping loop block.

Generally, in block–structured meta models like BPEL4WS [5] or ADEPT [88], for example, it is forbidden that sync links cross the boundaries of loop blocks. However, uncontrolled propagation of complex process type changes to biased instances may result in such undesired sync links as the following example shows:

Example 5.18.a (Sync Edge Crossing Loop Boundary): Please consider Figure 5.18 where type change Δ_T inserts an embracing loop block around existing control block (B, C) and instance change Δ_I inserts a sync link between activities C and D. Propagating Δ_T to instance–specific schema S_I results in a structural inconcistency, i.e., sync link (C, D) crosses the boundary of (new) loop block (B, C). Obviously, running instance I according to $(S+\Delta_I)+\Delta_T$ may result in an inconsistent execution state: If loop condition $lCond1$ evaluates to `True` the execution states of activities B and C are reset to `NotActivated`. As a consequence, actually, edge marking of sync link (C, D) should have to be reset to `NotSignaled` what would result in an incorrect marking on $(S + \Delta_I) + \Delta_T$ (cf. Definition 4).

Therefore, keeping the claim of avoiding materialization of $(S + \Delta_I) + \Delta_T$ in mind, we now present a method to decide whether $(S + \Delta_I) + \Delta_T$ contains sync links crossing the boundaries

[5] The case where $(begin_{\Delta_T}, end_{\Delta_T}) = (begin_{\Delta_I}, end_{\Delta_I})$ holds is caught by the assumption that Δ_T and Δ_I are disjoint

[6] $\Delta_{\{\}}$ denotes the set based representation of change $\Delta := (op_1, \ldots, op_n)$ which actually is an ordered series of changes $op_i (i = 1, \ldots, n)$, i.e., $\Delta_{\{\}} := \{op_1, \ldots, op_n\}$

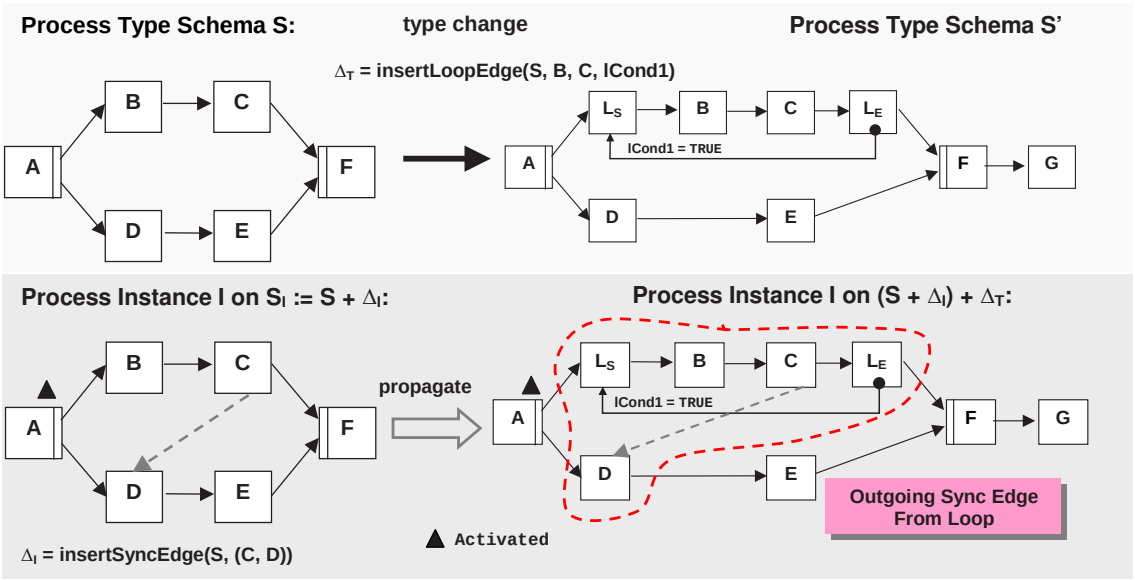


Figure 5.18: Sync Link Crossing Boundary Of A Loop Block

of loop blocks based on the information given by Δ_T and Δ_I.

Proposition 5 (Sync Links Crossing Boundaries Of Loop Blocks) *Let S be a WSM Net and I be a biased instance with starting schema S and execution schema $S_I := S + \Delta_I = (N_I, D_I, NT_I, CtrlE_I, SyncE_I, \ldots)$. Assume that type change Δ_T transforms S into a correct schema S' = (N', D', NT', CtrlE', SyncE', ...).*

Then: $(S + \Delta_I) + \Delta_S$ does not contain sync links crossing the boundaries of loop blocks iff:

$$(\forall (s_I, d_I) \in SyncE_I \setminus SyncE,\ \forall (begin_{\Delta_T}, end_{\Delta_T}) \in EmbracedLoopBlocks(S, \Delta_T):$$
$$\{s_I, d_I\} \cap (begin_{\Delta_T}, end_{\Delta_T}) = \emptyset) \wedge$$
$$(\forall (s_T, d_T) \in SyncE' \setminus SyncE,\ \forall (begin_{\Delta_I}, end_{\Delta_I}) \in EmbracedLoopBlocks(S, \Delta_I):$$
$$\{s_T, d_T\} \cap (begin_{\Delta_I}, end_{\Delta_I}) = \emptyset) \ (\propto)$$

$$\text{whereas } EmbracedLoopBlocks(S, \Delta) := \{\{begin_\Delta, end_\Delta\} \cup (succ^*(S, begin_\Delta) \cap pred^*(S, end_\Delta)) \mid$$
$$\exists insertLoopEdge(S, begin_\Delta, end_\Delta, lC_\Delta) \in \Delta_{\{\}}\}$$

Informally, condition ($\propto$) of Proposition 5 checks whether for all newly inserted sync links and newly inserted embracing loop blocks source or destination activity of the sync link lies within the respective loop block or not:

Example 5.18.b (Detecting Sync Links Crossing Loop Block Boundaries): Again look at Figure 5.18. Condition ($\propto$) of Proposition 5 yields: For $(C, D) \in SyncE_I \setminus SyncE$ holds $C \in \{B, C\}$ and therefore a conflict of the respective type is indicated.

5.4.3.3 Concurrent Application Of Arbitrary Change Transactions

When concurrently applying arbitrary change transactions $\Delta_T := (op_1^T, \ldots, op_n^T)$ and $\Delta_I := (op_1^I, \ldots, op_k^I)$, all possible combinations between their single change operations op_i^T $(i = 1, \ldots, n)$ and op_j^I $(j = 1, \ldots, k)$ may cause structural conflicts. Therefore, actually, we would have to checks all $n * k$ combinations of single operations op_i and op_j in order to determine potential structural conflicts (possibly together with respective graph reductions). However, based on the knowledge summarized in Table 5.1 many change operations like deleting activities or sync edges are totally uncritical regarding their concurrent application. Conversely, we can clearly determine the set of potentially conflicting changes

$$ConfChanges := \{serialInsertActivity(S, \ldots), insertSyncEdge(S, \ldots), insertBetweenNodeSets(S, \ldots),, addDataEdges(S, \ldots), deleteDataEdges(S, \ldots)\ insertLoopEdge\}.$$

Consequently, it is sufficient to restrict conflict test to those subsets of Δ_T and Δ_I which are candidates to cause structural conflicts, i.e., which are contained in $ConfChanges$. Using this information we obtain $ConfCand(\Delta_T) := \{op_i \in \Delta_{T\{\}}|$[7] $op_i \in ConfChanges\}$ for process type change Δ_T and $ConfCand(\Delta_I) := \{op_j \in \Delta_{I\{\}}|\ op_j \in ConfChanges\}$ for process instance change Δ_I. With this we reduce the number of necessary conflict checks to $|ConfCand(\Delta_T)|*|ConfCand(\Delta_I)|$. However, further optimization can be achieved by grouping $ConfCand(\Delta_T)$ and $ConfCand(\Delta_I)$ into further sub–groups according to the type of structural conflict they may cause, e.g., one possible sub–group in this context would be

$$ConfCycle(\Delta_T) := \{op_i \in \Delta_{T\{\}}|op_j \in \{serialInsertActivity(S, \ldots), insertSyncEdge(S, \ldots), insertBetweenNodeSets(S, \ldots), \}\}.$$

Doing so only changes contained in sub–groups of the same type would have to be compared.

To top this discussion off, in the following, we provide an illustrating example for an arbitrary process type change which inserts a new sync edge between newly inserted activities and also deletes several data edges. The uncontrolled propagation of this type change transaction may lead to different structural inconsistencies:

Example 5.19 (Structural Conflicts For Concurrently Applied Change Transactions): In Fig. 5.19 process type change Δ_T transforms schema S into new schema version S' by serially inserting activites X and Y and by connecting them via a sync link (X, Y). Furthermore Δ_T deletes activities F and H together with their respective data links dL_5 and dL_6. Based on original schema S two instances have been started. Instances I_1 and I_3 are biased and therefore run according to their instance-specific schema S_{I_1} and S_{I_3} respectively whereas I_2 is an unbiased instance still running according to S. If now type change Δ_T shall be propagated to these instances we have to check structural as well as state-related compliance for the running instances. Instance change Δ_{I_1} has serially inserted two activites U and T and sync link (T, U) between

[7] $\Delta_{\{\}}$ denotes the set based representation of change $\Delta := (op_1, \ldots, op_n)$ which actually is an ordered series of changes $op_i (i = 1, \ldots, n)$, i.e., $\Delta_{\{\}} := \{op_1, \ldots, op_n\}$

them. At first, the deadlock test derived from Proposition 3 is carried out to detect whether target schema $(S + \Delta_{I_1}) + \Delta_T$ will contain a deadlock-causing cycle or not. After applying the graph reduction rules of Algorithm 3 we obtain that $(S + \Delta_{I_1}) + \Delta_T$ will actually contain a deadlock-causing cycle and therefore I_1 cannot migrate to S' (and remains running according to S). For unbiased Instance I_2 we only have to check state-related compliance as described in Section 4.3.2. Since the previous trace of I_2 can be replayed on S', I_2 is compliant with S' and therefore migrates to S' by applying appropriate marking adaptation rules (cf. Section 4.3.2). For instance–specific change Δ_{I_3} tests on data flow conflicts become necessary since Δ_T and Δ_{I_3} both add and delete data edges (and therefore are possible candidates for data flow conflicts). However, no graph reductions are necessary. The reason is that all activities are present in original process schema S for which a data edge is added or deleted by Δ_T or Δ_{I_3} respectively. Therefore we check condition (♣) of Proposition 1 and obtain that different data elements are concerned by the relevant data flow change operations of Δ_{I_3} and Δ_T. Due to this fact and based on the compliance conditions set out in Section 4.3.2 instance I_3 is compliant with S' regarding structure and state and can be therefore migrated to S'.

5.5 Migrating Process Instances with Disjoint Bias

Consider again Example 5.19 where process instance I_3 has been determined as being compliant with changed process type schema S' regarding structure and state (cf. Criterion 7), i.e., the resulting instance–specific schema $(S+\Delta_{I_3})+\Delta_T$ does not contain structural inconsistencies and the marking of I_3 on S' which results from applying Algorithm 2 is a correct instance marking according to Theorem 9.

However, running (arbitrary) compliant process instance I according to its instance–specific schema $(S + \Delta_I) + \Delta_T$ is still not the desired behaviour. The reason is that in this situation I cannot be considered as process instance running according to changed process type schema S'. This can only be the achieved if we are able to (logically) transfer I from running on instance–specific schema $(S + \Delta_I) + \Delta_T$ to be based on instance–specific schema $S'_I := (S + \Delta_T) + \Delta_I = S' + \Delta_I$. Only if process instance I is executed according to instance–specific schema $S'_I := (S + \Delta_T) + \Delta_I$ it can be considered as really migrated to changed process type schema S'. Reassigning process instance I to changed process type schema S' is very important since

- only then I may benefit from further process optimizations carried out on S'.
- doing so is key to an optimal internal management of process type and process instance data.

How can we realize the (logical) transfer of process instance I from running on $(S+\Delta_I)+\Delta_T$ to process instance I running according to instance–specific schema $(S + \Delta_T) + \Delta_I = S' + \Delta_I$? Obviously, this transfer is possible if the order of applying Δ_T and Δ_I on original process schema S is irrelevant according to resulting schemes $(S + \Delta_I) + \Delta_T$ and $(S + \Delta_T) + \Delta_I$, i.e.,

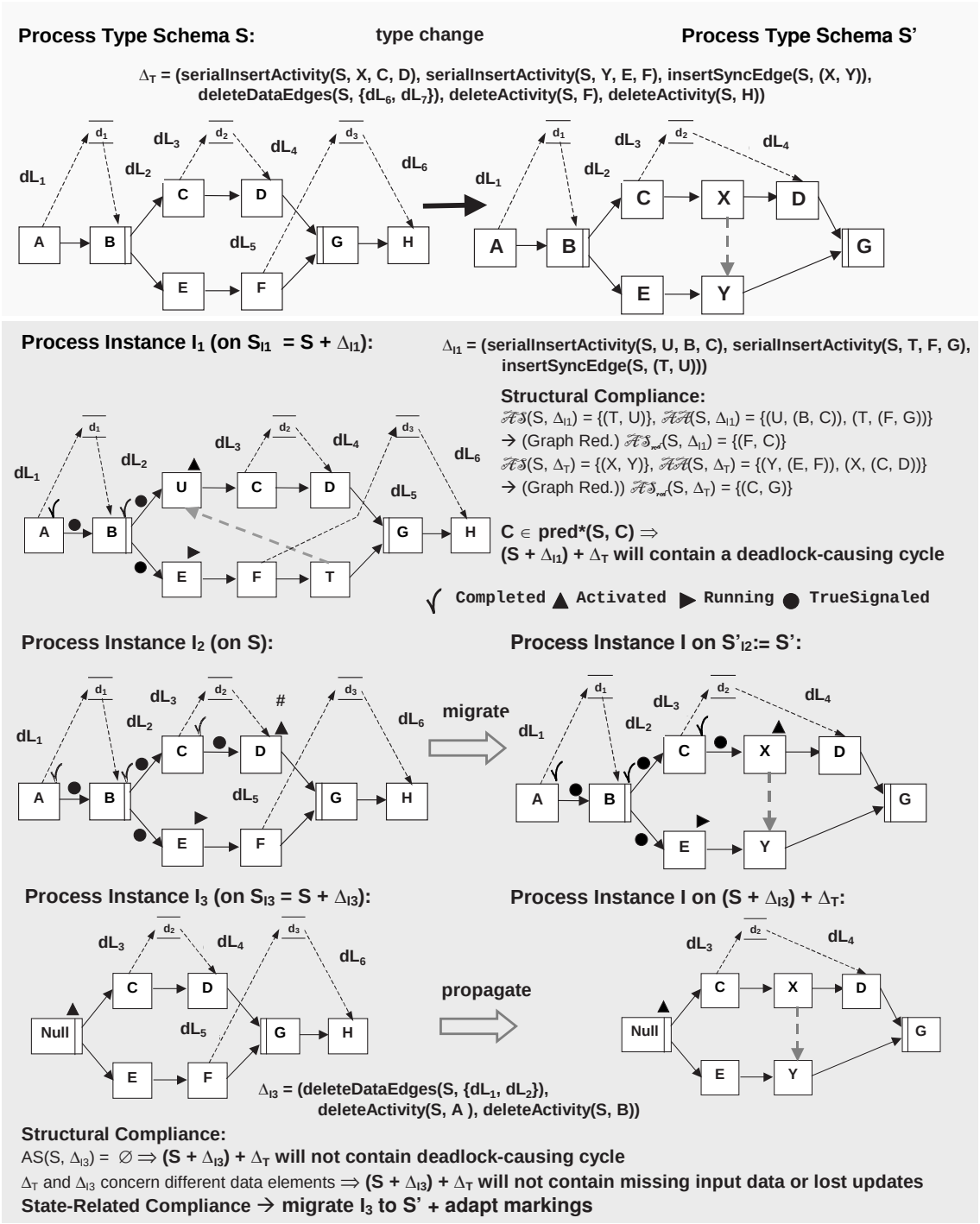


Figure 5.19: Process Type Change and Instance Migration

$(S + \Delta_I) + \Delta_T$ and $(S + \Delta_T) + \Delta_I$ should be the "same" instance–specific schemes. What does the "same" instance–specific schemes exactly mean? We have been already confronted with this problem in Section 5.2 when thinking about an adequate equivalence relation on process schemes. The key to solution is trace equivalence (cf. Definition 8). More precisely, two process schemes are trace equivalent if and only if each possible trace on the one schema can be executed on the other schema and vice versa. Trace equivalence therefore ensures behavioral equivalence of instances. However, behavioural equivalence provides a sufficient basis for transferring the execution of process instance I (running on instance–specific schema $(S + \Delta_I) + \Delta_T$ so far) to instance–specific schema $(S + \Delta_T) + \Delta_I$. More precisely, if $(S + \Delta_I) + \Delta_T \equiv_{trace} (S + \Delta_T) + \Delta_I$ holds process instance I on $(S + \Delta_I) + \Delta_T$ can also be seen as an instance running according to $(S + \Delta_T) + \Delta_I$.

In summary, the desired reassignment of process instance I to changed process type schema S' is possible if $(S + \Delta_I) + \Delta_T \equiv_{trace} (S + \Delta_T) + \Delta_I$ holds. Obviously, this is the case if process type change Δ_T and process instance change Δ_I are disjoint (cf. Definition 10) since one necessary prerequisite for disjoint process type and process instance changes is that they are commutative, i.e., $(S + \Delta_I) + \Delta_T \equiv_{trace} (S + \Delta_T) + \Delta_I$ holds.

Therefore, an important conclusion is that for disjoint process type and process instance changes the respective instances can be always reassigned to the changed process type schema if they are compliant regarding structure and state (cf. Criterion 8). Formally:

Migration Strategy 1 (For Process Instances With Disjoint Bias) *Let S be a (correct) process type schema and I be a process instance which has been started according to S and has been already biased by instance–specific change Δ_I. Let further Δ_T be a process type change which transforms S into another (correct) process type schema S'. Assume that Δ_T and Δ_I are disjoint changes according to Definition 10. Assume further that process instance I is compliant with S' regarding Criterion 8. Then process instance I can be reassigned to changed process type schema S' preserving instance–specific change Δ_I on S', i.e., I runs according to instance–specific schema $S' + \Delta_I$ further on.*

The validity of Migration Strategy 1 directly follows from the assumption that Δ_T and Δ_I are disjoint.

Another interesting result which can be seen from Migration Strategy 1 is that instance–specific change Δ_I can be preserved on S' if it is disjoint with Δ_T. Note that this is not the case if process type and process instance changes overlap. In the next chapter we will extensively discuss how the new bias (based on changed process type schema S') can be calculated according to the particular degree of overlap between process type and process instance changes.

Finally, we illustrate the migration process for instances with disjoint bias by means of the following example:

Example 5.20 (Migrating Process Instance With Disjoint Bias): Consider Figure 5.20 where process instance I has been individually modified by instance–specific change Δ_I. Obviously,

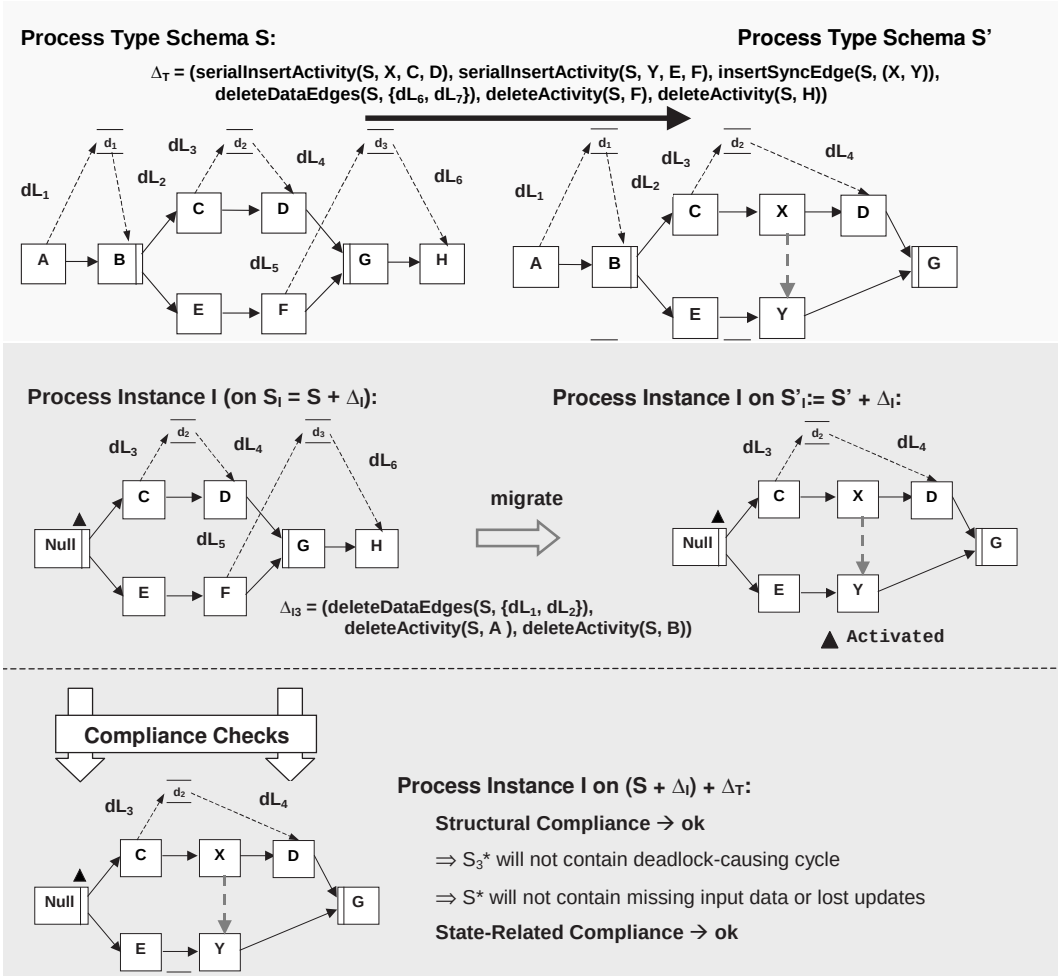


Figure 5.20: Migrating Compliant Instance To S'

Δ_T and Δ_I disjoint. Regarding structural compliance of instance–specific schema $(S+\Delta_I)+\Delta_T$ there will be no inconsistencies like deadlock causing cycles or missing input data for activity executions. I is also compliant regarding its state and can therefore be correctly migrated to S'. As we can see from Figure 5.20 instance–specific schema $(S + \Delta_I) + \Delta_T$ is trace equivalent with instance–specific schema $S'_I := (S + \Delta_T) + \Delta_I = S' + \Delta_I$. Consequently I can be migrated to $S'_I := S' + \Delta_I$ preserving instance–specific change Δ_I on S' (cf. Theorem 1).

5.6 Summary

In this chapter we have provided formal definitions based on which it can be decided whether concurrent changes are disjoint or overlapping. For disjoint process type and instance changes a gerenal correctness criterion was introduced based on which state–related, structural, and semantical compliance can be decided. In order to provide an efficient solution we have developed several conflicts tests which quickly check whether there will be structural conflicts between process type and process instance changes or not. To execute the migration of instances with disjoint bias, commutativity was identified as necessary property of disjoint changes. The reason is that due to commutativity instances can be reassigned to the changed process type schema while preserving their original bias. Finally, the presented results can be transferred to other process meta models as well. The design of the structural conflict tests depends on whether change primitives are offered to users (as indicated by the example of isolated activity nodes for Activity Nets) or change operations with formal pre– and post–conditions are used (e.g., WIDE [26] and TRAMs [67]).

Chapter 6

Migrating Process Instances with Overlapping Bias

The ability to migrate *unbiased* process instances to a changed process type schema is indispensable in practice (cf. Chapter 4). However, this migration support is still not sufficient since one must also adequately deal with concurrent changes on business processes. More precisely, a fully flexible PMS must be able to support the migration of biased instances to a changed process type schema as well. As summarized in Figure 6.1, an approach for handling the migration of process instances with disjoint bias to a changed process schema has been already presented in Chapter 5. Thereby focus was put on providing a general correctness criterion (cf. Correctness Criterion 8) and on (quickly) checking structural and state–related compliance. In this chapter, we present a solution for dealing with the migration of biased process instances for which instance–specific changes and type change overlap (cf. Figure 6.1). This imposes many new challenges like the detection of the particular degrees of overlap and the provision of adequate migration strategies.

The remainder of Chapter 6 is organized as follows: In Section 6.1, we introduce the general challenges that arise when migrating process instances with overlapping bias. Section 6.2 provides a classification of concurrent changes along their particular degree of overlap. In Sections 6.3 and 6.4, we present and discuss different approaches to detect the particular degree of overlap between process type and process instance changes. In Section 6.5 we then stick the strengths of these approaches together to a hybrid approach. Section 6.6 summarizes the different migration strategies along the particular degree of overlap and Section 6.7 provides a method to calculate the new bias after migration. Finally, Section 6.8 summarizes the presented results.

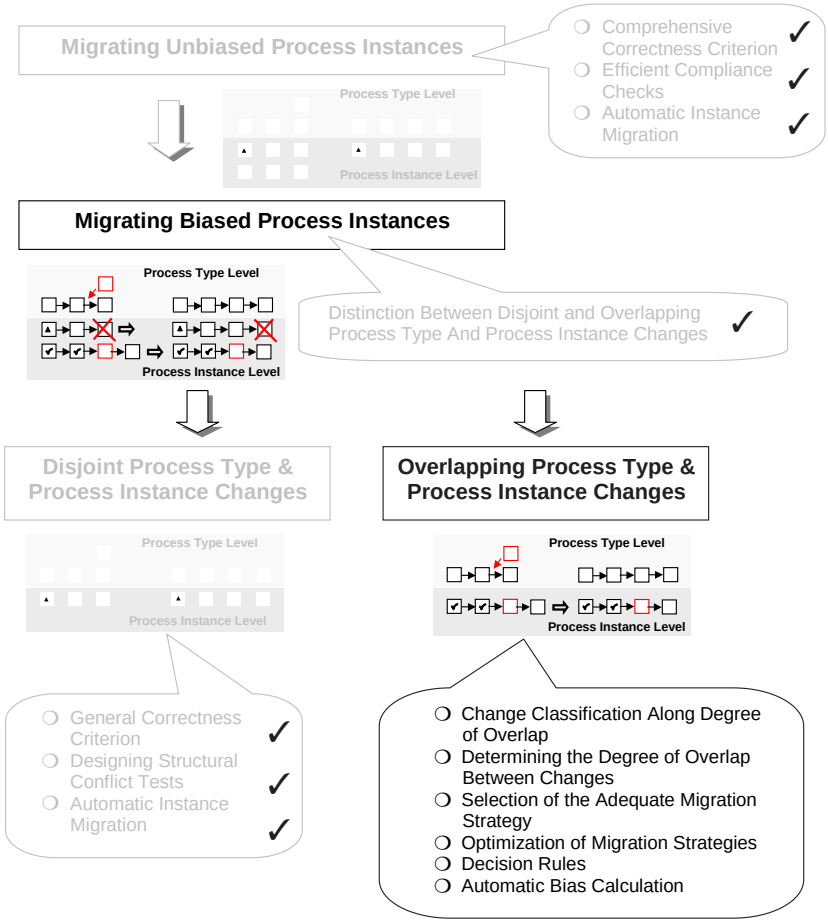


Figure 6.1: Migrating Process Instances with Overlapping Bias

6.1 Advanced Migration Issues – Challenges

As indicated above we want to present an approach to be able to adequately deal with overlapping process type and process instance changes (cf. Definition 10). To shortly repeat the difference between disjoint and overlapping process changes consider the following example:

Example 6.2 (Disjoint And Overlapping Changes:) Look at Figure 6.2 where process type change Δ_T deletes activity D and process instance change Δ_{I_1} deletes activity H. Obviously, these changes work on totally different elements of original process schema S and can therefore be regarded as being disjoint. This corresponds to Definition 10 (cf. Section 5.2) of disjointness: Δ_T and Δ_{I_1} are commutative since $(S + \Delta_{I_1}) + \Delta_T \equiv_{trace} (S + \Delta_T) + \Delta_{I_1}$ holds and the activity sets newly inserted by both changes are disjoint. In contrast, process type change Δ_T and process instance change Δ_{I_2} overlap since both delete same activity D from original schema S.

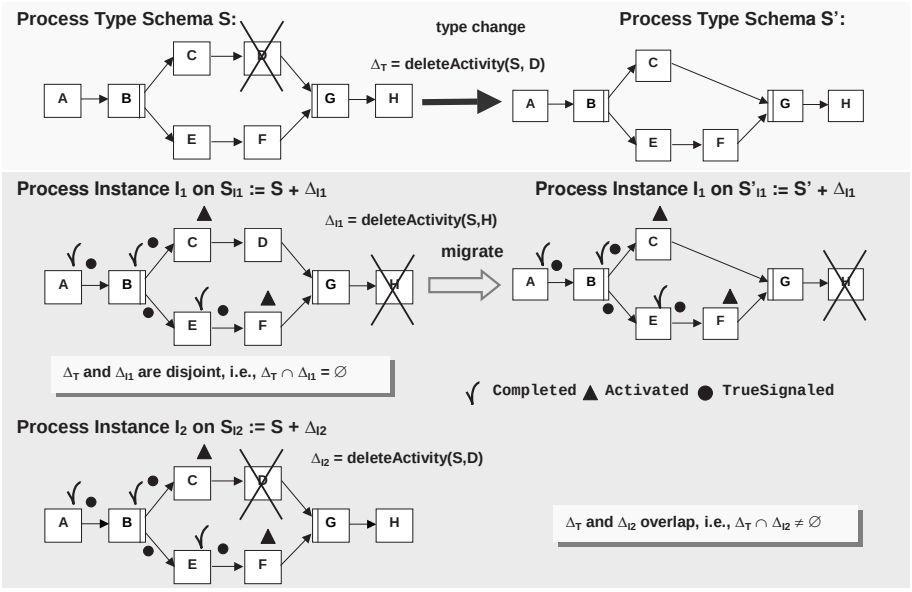


Figure 6.2: Disjoint and Overlapping Process Type and Instance Changes

Again this corresponds to Definition 10: Δ_T and Δ_{I_2} are not commutative since Δ_T cannot be applied to instance–specific schema S_{I_2} and Δ_{I_2} can also not be applied to changed process type schema S'. Consequently, instance–specific schemes $(S + \Delta_{I_2}) + \Delta_T$ and $(S + \Delta_T) + \Delta_{I_2}$ cannot be produced and Δ_T and Δ_{I_2} are rated as overlapping.

When looking at our current definition of disjoint and overlapping process type and process instance changes it strikes that the part concerning disjointness is very precise whereas the part regarding overlapping changes is relatively fuzzy. In other words, the set of process instances with overlapping bias may turn out to be very heterogenous; i.e., it may contain instances for which instance–specific changes significantly vary regarding their degree of overlap with the process type change. Instance–specific changes may reach from changes having the same effects on original process schema S (as the process type change) to changes which only marginally overlap with the process type change. Due to this fact, finding a common migration strategy for all these instances may turn out very difficult or may be even impossible. Consequently, a first important challenge is to find a classification for biased instances along their particular *degree of overlap* between their instances–specific change and the process type change. Doing so is important in order to be able to establish practically applicable migration strategies in the sequel.

Assume that we have found an appropriate classification of process instances for which their instance–specific change overlaps the process type change[1]. The important question then is how to efficiently decide which class a particular process instance belongs to. When discussing this

[1]In the following we use the term "process instances with overlapping bias" short hand for such instances.

issue we will also answer the question raised in Section 5.2 about how to quickly detect process instances with disjoint bias.

Based on an adequate classification dedicated migration strategies can be found. This includes automatic instance adaptations (if desired by users) as well as detailed messages to guide users through the migration process. In other words, depending on the particular degree of overlap (between process type and process instance changes) recommendations and rules can be generated which make the migration process transparent to users and help them to intervene if they want to.

After having selected and applied the right migration strategy (either automatically by the PMS or manually by users) process instances are "linked" (i.e., assigned) to the changed process schema. This holds advantages like the possibility to take part at future type changes (e.g., future process optimizations). As necessary pre–conditions for migrating instances with overlapping bias to a changed process type schema, structural and state–related checks on the respective instances may become necessary. For instances with disjoint bias these checks can be carried out by applying the conflict tests introduced in Section 5.4 and the compliance conditions set out in Section 4.3.2. If they are compliant regarding structure and state, process instances with disjoint bias can migrate to the changed process type schema by maintaining their original bias. An example is depicted in Figure 6.2 where instance–specific change $\Delta_{I_1} = deleteActivity(S, H)$ is maintained after migrating I_1 to changed process type schema S'. In contrast, for compliant process instances with overlapping bias doing so would not be correct; i.e., the effects on original process type schema S which process type and process instance change have in common are already incorporated within schema S'. Therefore, the remaining bias related to S' has to be re–calculated. For example, consider Figure 6.2 where a migration of process instance I_2 to changed process type schema S' is desired. Bias Δ_{I_2} is totally reflected in S' since Δ_T and Δ_{I_2} have the same effects on original schema S. Consequently, Δ_{I_2} should not be maintained on S' after migrating I_2 to S'. Instead, the resulting bias of I_2 on S' must be newly calculated on S' (and becomes the empty set for this example).

In summary, in order to adequately deal with overlapping process type and process instance changes we have

1. to find a classification along the particular *degree of overlap*. Based on this classification appropriate migration strategies can be established.
2. to provide methods to efficiently decide on the particular degree of overlap for given process type and process instance changes.
3. to provide migration strategies along the different degrees of overlap between process type and process instance changes (including rules for guiding the user).
4. to provide methods to automatically adapt process instances when migrating them to the changed process type schema. This includes structural and state–related adaptations (as already known for process instances with disjoint bias) as well as re–calculation of the bias (i.e., the remaining instance–specific change based on S').

6.2 On Classifying Concurrent Changes

In this section, we want to find an appropriate classification for overlapping process type and process instance changes along their particular *degree of overlap*. First of all, we abstract from whether changes are carried out at the type or at the instance level. More precisely, we base our considerations on two arbitrary changes Δ_1 and Δ_2 concurrently applied to same schema S[2].

Let S be a (correct) process schema and let Δ_1 and Δ_2 be two changes which transform S into another (correct) process schema S_1 and S_2 respectively (notation: $S_1 := S + \Delta_1$ and $S_2 := S + \Delta_2$).

The particular degree of overlap between Δ_1 and Δ_2 corresponds to the kind of relation between the concerned changes (like disjointness and overlap of changes in general, cf. Section 5.2). In order to understand the relation between two changes Δ_1 and Δ_2 applied to same original process schema S we must compare their particular effects on S. In other words, the degree of overlap between Δ_1 and Δ_2 corresponds to the degree of overlap between their effects on S.

Which relationships between the effects of two changes Δ_1 and Δ_2 on original schema S are conceivable? As introduced in Section 5.2, Δ_1 and Δ_2 may have totally different effects on S, i.e., Δ_1 and Δ_2 may be *disjoint*. In contrast, Δ_1 and Δ_2 may have exactly the same effects on original schema S. In this case, we denote Δ_1 and Δ_2 as *equivalent* changes. Between these two extremes a "containted–in" relation between Δ_1 and Δ_2 is conceivable [97]. More precisely, Δ_1 may have the same effects on S as Δ_2 has, but may have further effects on S as well. In this case we call Δ_1 and Δ_2 *subsumption equivalent*. Note that this relation is bi–directional. Finally, Δ_1 and Δ_2 may partially have the same effects on S but both Δ_1 and Δ_2 may have additional effects on S as well (not reflected by the other changes). Then we denote Δ_1 and Δ_2 as *partially equivalent*. Informally, the different degrees of overlap between Δ_1 and Δ_2 can be summarized as follows:

Summary 1 (Degrees Of Overlap Between Concurrent Changes) *Let S be a (correct) process schema and let Δ_1 and Δ_2 be two changes which transform S into (correct) process schemes S_1 and S_2 respectively (notation: $S_1 := S + \Delta_1$ and $S_2 := S + \Delta_2$). Then we denote Δ_1 and Δ_2 as*

1. *disjoint if they have totally different effects on S (notation: $\Delta_1 \cap \Delta_2 = \emptyset$)*
2. *equivalent if they have exactly the same effects on S (notation: $\Delta_1 \equiv \Delta_2$)*
3. *subsumption equivalent if Δ_2 has the same effects on S as Δ_1, but Δ_2 has additional effects on S as well (notation: $\Delta_1 \prec \Delta_2$)*

[2]One possible application of these general results is the treatment of process type and process instance changes concurrently applied to original process schema S. This case is reflected throughout all examples presented in this chapter. Apart from this, the general results are also applicable in connection with concurrent instance changes (e.g., introduced by different users at the same time).

4. *partially equivalent if* Δ_1 *and* Δ_2 *have some common effects on* S*, but both have additional different effects on* S *(notation:* $\Delta_1 \between \Delta_2$*)*

We now want to illustrate the different degrees of overlap by applying the general results of Summary 1 to concurrent process type and process instance changes:

Example 6.3 (Concurrent Changes With Different Degrees Of Overlap): Consider Figure 6.3: Type change Δ_T inserts activity X between activities B and D, and it inserts activity Y between activities A and C.

- Instance I_1: Obviously, Δ_T and the instance–specific change Δ_{I_1} are disjoint since they have totally different effects on S[3]. Therefore I_1 can be linked to $S'_{I_1} := S' + \Delta_{I_1}$ preserving instance–specific change Δ_{I_1} on S' (cf. Theorem 1).
- Instance I_2: Here the situation is quite different. The instance–specific change Δ_{I_2} has exactly the same effects on S as Δ_T since both changes insert the same activities X and Y into S at the same target positions. Therefore Δ_T and Δ_{I_2} are detected as equivalent changes (cf. Summary 1). Intuitively, I_2 can be linked to S' without applying Δ_T to S_{I_2} (since the effects of Δ_T are already contained in S_{I_2}). Furthermore no instance–specific change based on S' has to be kept, i.e., $\Delta_{I_2}(S') = \emptyset$ holds.
- Instance I_3: The instance–specific change Δ_{I_3} has inserted a subset of activities $\{Y\}$ when compared to the set of activities $\{X, Y\}$ inserted by Δ_T. Furthermore, Y has been inserted at the same target position between A and C. Altogether, the effects of Δ_{I_3} on S are subsumed by the effects of Δ_T on S. Consequently, Δ_T and Δ_{I_3} are considered as subsumption equivalent, i.e., $\Delta_{I_3} \prec \Delta_T$ holds (cf. Summary 1). I_3 can therefore be re–linked to S' but without need for maintaining any instance–specific change based on S' since Δ_{I_3} is already reflected within S' ($\Delta_{I_3}(S') = \emptyset$).
- Instance I_4: The effects of the instance–specific change Δ_{I_4} subsume the effects of Δ_T since Δ_{I_4} inserts same activities X and Y into S at the same positions as Δ_T does, but additionally inserts activity Z between D and E. Therefore I_4 can be migrated to S'. However, we have to maintain an instance–specific bias $\Delta_{I_4}(S')$, which differs from original instance–specific bias $\Delta_{I_4}(S)$. More precisely we have to remove the effects of Δ_T (already reflected within S') from Δ_{I_4} in order to obtain the new instance–specific bias $\Delta_{I_4}(S') = (serialInsertActivity(S, Z, D, E))$.
- Instance I_5: The instance–specific change Δ_{I_5} has partially the same effects on S as Δ_T has; i.e., both changes insert activity Y between A and C. However, on the one hand Δ_{I_5} has deleted activity E and on the other hand Δ_T additionally inserts activity X between B and D. According to Summary 1, Δ_{I_5} and Δ_T are partially equivalent, i.e., $\Delta_T \cap \Delta_{I_5} \neq \emptyset$ holds. In the given situation it is possible to migrate I_5 to S' and to determine new instance–specific bias $\Delta_{I_5}(S')$ based on S' ($\Delta_{I_5}(S') = (deleteActivity(S, E))$). However, doing so is not always possible for partially equivalent changes as we will show later on.

[3]We assume that X and Y denote semantically different activities.

Summary 1 provides a rather informal description of the different degrees of overlap between concurrent changes Δ_1 and Δ_2. To base our further consideration on a sound framework we formalize the statements given in Summary 1: Regarding disjoint changes we have already given (formal) Definition 10 (cf. Section 5.2). Equivalence between changes Δ_1 and Δ_2 can be formalized as follows:

Definition 11 (Equivalent Changes) *Let S be a (correct) process schema and let Δ_1 and Δ_2 be two changes which transform S into (correct) process schemes S_1 and S_2. Then we denote Δ_1 and Δ_2 as being equivalent (notation: $\Delta_1 \equiv \Delta_2$) iff S_1 and S_2 are trace equivalent (cf. Def. 8). Formally:*

$$\Delta_1 \equiv \Delta_2 \Longleftrightarrow S_1 \equiv_{trace} S_2$$

Thus concurrent changes Δ_1 and Δ_2 have the same effects on the original schema S if the resulting process schemes S_1 and S_2 show the same behavior, i.e., S_1 and S_2 are trace equivalent (cf. Definition 8). This corresponds to the intuitive understanding of equivalent changes: Δ_1 and Δ_2 have exactly the same effects on process schema S if – based on the resulting schemes $S_1 := S + \Delta_1$ and $S_2 := S + \Delta_2$ – the same behavior can be achieved. More precisely, each activity sequence (together with its read/write operations on data elements) producible on S_1 must be also producible on S_2 and vice versa.

We have formally defined change equivalence based on behavorial equivalence of the resulting process schemes S_1 and S_2 so far. We now try to define subsumption and partial equivalence in a similar manner (cf. Statements 3 + 4 in Summary 1). We start with the definition of subsumption equivalence between changes. Obviously, in this case, S_1 and S_2 are not trace equivalent since Δ_2 has "more effects" on S than Δ_1 has (or vice versa). However, we would obtain trace equivalent schemes if the additional effects of Δ_2 on S (when compared with Δ_1) had been applied to S_1 as well. In terms, Δ_1 is subsumption equivalent with Δ_2 ($\Delta_1 \prec \Delta_2$) if process schemes $(S + \Delta_1) + \mathbf{\Delta_2} \setminus \mathbf{\Delta_1}$ and $S + \Delta_2$ are trace equivalent. Similarly, two concurrent changes Δ_1 and Δ_2 are partially equivalent if $(S + \Delta_1) + \mathbf{\Delta_2} \setminus \mathbf{\Delta_1} \equiv_{trace} (S + \Delta_2) + \mathbf{\Delta_1} \setminus \mathbf{\Delta_2}$ holds. In both cases, the difference of changes $\Delta_1 \setminus \Delta_2$ or $\Delta_2 \setminus \Delta_1$ shall represent the difference of effects of Δ_1 and Δ_2 on S. However, the difference between changes cannot be precisely determined if the change logs to be compared contain *noise*[4] or change operations depend on each other. More precisely:

1. **Missing Canonical Representation Of Changes:** Change logs may often contain information about change operations which actually have no or only hidden effects on the underlying process schema. One example is the (temporary) insertion of activities which are deleted in the sequel. The reason is that users who define the changes (i.e., the process designer or end user) do not always act in a goal–oriented way when modifying a process schema. In fact, they may try out the best solution resulting in noisy information within

[4]Noise in change logs is somewhat comparable to the existence of noisy information in process execution logs. This noisy information impedes, for example, the mining of these logs [50, 47].

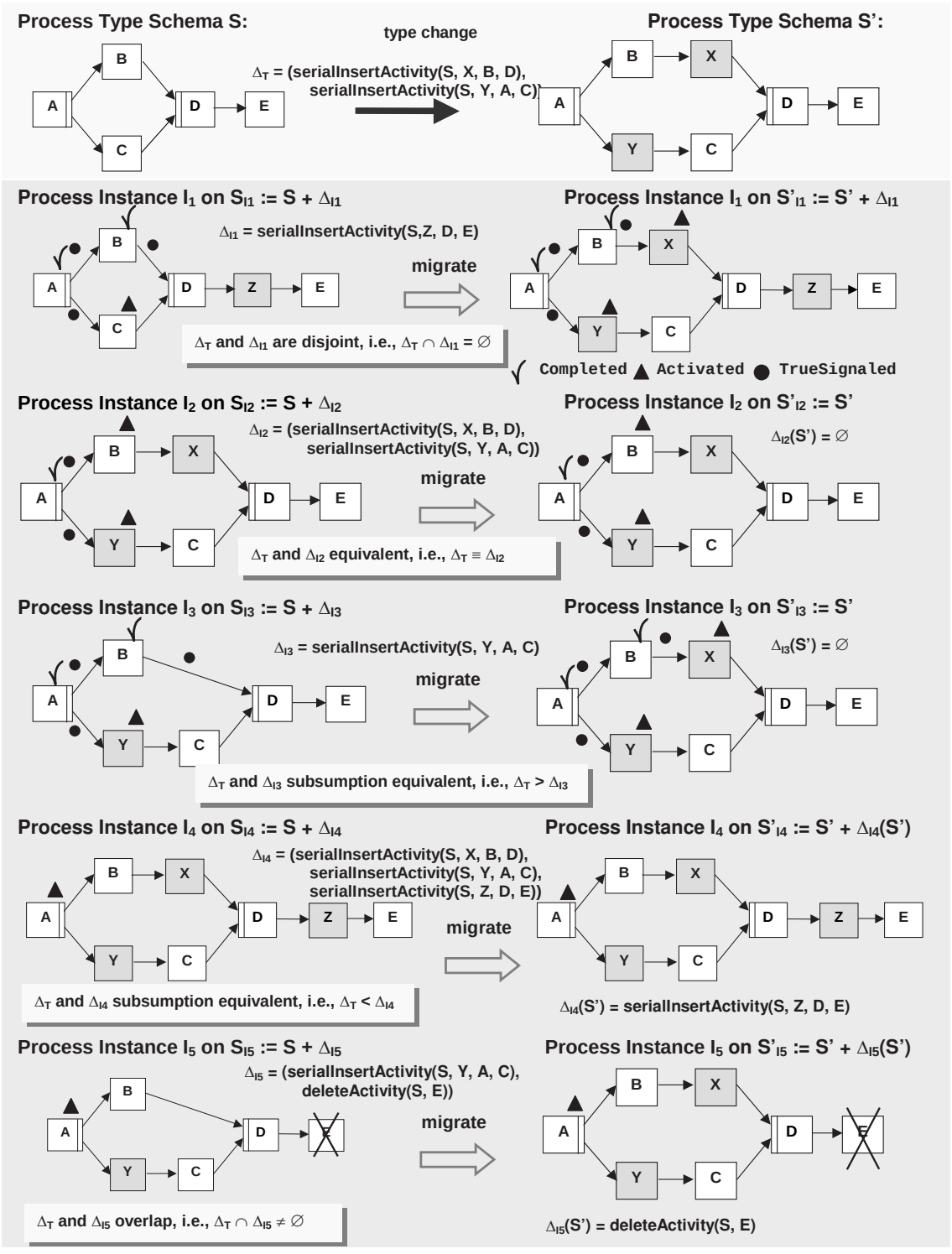


Figure 6.3: Concurrent Changes with Different Degrees of Overlap

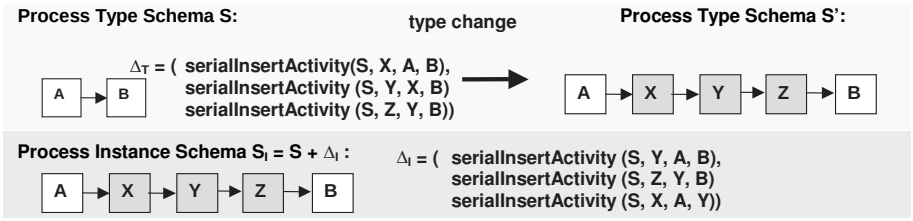


Figure 6.4: Context–Dependent Insert Operations

the change logs. We will extensively discuss the issue of noisy change logs in Section 6.4.2. At the moment, it is sufficient to be aware of the fact that such noise exists.

2. **Context–Dependent Changes:** As discussed in Section 5.4.3 there may be context–dependent changes within a change log, i.e., changes which are based on each other. Applying context–dependent changes in a different order may lead to different change logs such that these logs cannot be properly compared. An example for this phenomenon is depicted in Figure 6.4:

   Example 6.4 (Context–Dependent Insert Operations:) Process type change Δ_T and process instance change Δ_I both insert activities X, Y, and Z into original process schema S at the same target position. Merely, Δ_T uses another order of applying the single insert operations as Δ_I does. However, building the differences between Δ_T and Δ_I results in empty sets. Note that solely based on the change logs, it cannot be determined that Δ_T and Δ_I are actually equivalent changes.

As we will show in Section 6.5.1 we can find methods to *purge* changes from noisy information. However, the problem of context–dependent changes remains, i.e., it is not possible to find a common representation for change logs containing context–dependent changes. Therefore, at the moment, we have to leave our attempt to find formal definitions for subsumption and partially equivalent changes based on change differences and trace equivalence. Fortunately, there is another possibility which finally enables us to formally define subsumption and partial equivalence between concurrent changes (cf. Section 6.6.1).

However, we already have formal definitions for disjoint and equivalent changes. Both are based on trace equivalence between process schemes (cf. Definition 8). Consequently, in order to be able to determine the particular degree of overlap (cf. Section 6.1) we have to ensure trace equivalence. For this purpose, a formal method based on process schema isomorphism is provided in Section 6.3.

6.3 A Formalism Based on Graph Isomorphism

In Sections 5.2 and 6.2, formal definitions for disjoint and equivalent changes have been provided (cf. Definitions 10 and 11). Both definitions are based on trace equivalence between process schemes (cf. Definition 8). Therefore when deciding about different degrees of overlap between changes Δ_1 and Δ_2 the important question is how to ensure trace equivalence between the resulting process schemes $S_1 := S+\Delta_1$ and $S_2 := S+\Delta_2$. Obviously, trying to replay all possible execution histories of process schema S_1 on process schema S_2 and vice versa is far too expensive. The determination of all possible execution histories producible on a process schema may result in exponential complexity. Fortunately, the problem of testing trace equivalence between two process schemes S_1 and S_2 can be solved by testing whether S_1 and S_2 are *isomorphic* [56] or not; i.e., by finding a bijective mapping between the activities and edges of two process schemes. A formal definition of this property is provided in Definition 12. Note that this definition is especially tailored to WSM Nets (cf. Section 3.1.1), i.e., the bijective mapping between the two WSM Nets is already fixed by the labelings of the respective activities and edges.

Definition 12 (Graph Isomorphism (GI)) *Let* $S_i = (N_i, D_i, NT, CtrlE_i, SyncE_i, LoopE_i, DataE_i)$ *($i = 1, 2$) be two (correct) process schemes. Then S_1 and S_2 are isomorphic (formally: $S_1 \simeq S_2$) if condition (♣) holds with*

(♣):

$[[\exists$ *bijective mapping f:* $N_1 \mapsto N_2$ *with*
$(label(n) = label(f(n)) \; \forall \; n \in N_1) \wedge$
$(\forall \; e= (u, v) \in CtrlE_1 : \exists \; e^* = (f(u), f(v)) \in CtrlE_2$ *with* $label(e)= label(e^*)$
$\wedge \; \forall \; e^* = (u^*, v^*) \in CtrlE_2 \; \exists \; e = (f^{-1}(u^*), f^{-1}(v^*)) \in CtrlE_1$
with $label(e^*) = label(e))$ $\wedge$
$(\forall \; e= (u, v) \in SyncE_1 : \exists \; e^* = (f(u), f(v)) \in SyncE_2$ *with* $label(e)= label(e^*)$
$\wedge \; \forall \; e^* = (u^*, v^*) \in SyncE_2 \; \exists \; e = (f^{-1}(u^*), f^{-1}(v^*)) \in SyncE_1$
with $label(e^*) = label(e))\wedge$
$(\forall \; e= (u, v) \in LoopE_1 : \exists \; e^* = (f(u), f(v)) \in LoopE_2$ *with* $label(e)= label(e^*)$
$\wedge \; \forall \; e^* = (u^*, v^*) \in LoopE_2 \; \exists \; e = (f^{-1}(u^*), f^{-1}(v^*)) \in LoopE_1$
with $label(e^*) = label(e))$ $]\wedge$

$[\exists$ *bijective mapping g:* $D_1 \mapsto D_2$ *with*
$(label(d) = label(g(d)) \; \forall \; d \in D_1) \wedge$
$(\forall \; dE = (n, d, mode) \in DataE_1, \; n \in N_1 :$
$\exists \; dE^* = (g(n), g(d), mode) \in DataE_2 : label(dE) = label(dE^*)$
$\wedge \; \forall \; dE^* = (n^*, d^*, mode) \in DataE_2$
$\exists \; dE = (g^{-1}(n^*), g^{-1}(d^*), mode) \in DataE_1 : label(dE^*) = label(dE)]]$

The following theorem formally states that isomorphism between two process schemes S_1 and S_2 also implies trace equivalence between them. Note that depending on the concrete operational semantics of the used process meta model two process schemes can be execution equivalent but not isomorphic. That is the case, for example, if dummy nodes for structuring

matters are used[5] or if the process schemes contain transitive control dependencies. A solution here is to reduce the respective process schemes. If necessary, the node set is reduced by logically discarding all dummy nodes.

Theorem 10 (Trace Equivalence By Process Schema Isomorphism) *Let S_1 and S_2 be two (correct) process schemes. Then S_1 and S_2 are trace equivalent if S_1 and S_2 are isomorphic according to Definition 12. Formally:*

$$S_1 \simeq S_2 \Longrightarrow S_1 \equiv_{trace} S_2$$

A formal proof of Theorem 10 can be found in Appendix C (cf. Proof C.9).

One can claim is that, in general, there is no efficient algorithm for detecting graph isomorphism (GI). In particular, no efficient algorithm has been found for GI (GI lies in class NP) [64]. But there are some special graph classes for which efficient isomorphism algorithms exist, e.g., planar graphs or graphs with bounded valence [24]. However, for two graphs with unique vertex labeling it is trivial to find an isomorphism between them. As it can be seen from Definition 12 this is the case for two process schema graphs since there is always a unique labeling which is preserved during the isomorphism mapping. One way would be to compare the adjacency matrices of both process schemes. Generally, this can be done with complexity $O(n^2)$ if n corresponds to the number of nodes[6]. Regarding process schemes the number of rows in the adjacency matrices depends on the number $|N|$ of activity nodes and the number $|D|$ of data elements. Therefore the complexity of this method results in $O((|N| + |D|)^2)$

To our best knowledge there is no approach with lower complexity than $O(|M|^2)$ (with $M = max(|N|, |D|)$) for this special problem. This could cause a performance problem if we have to check process graph isomorphism for complex process schemes and for a large number of process instances at runtime. Furthermore, as discussed in Section 6.2, verification of trace equivalence using process schema isomorphism is only applicable for detecting disjoint and equivalent process changes. Therefore, in the following sections, we present better approaches. These approaches can be applied for detecting all conceivable degrees of overlap between process changes.

6.4 Structural and Operational Approaches

Let S be a (correct) process schema and Δ_1 and Δ_2 two changes which transform S into (correct) process schemes S_1 and S_2 respectively (notation: $S_1 := S + \Delta_1$ and $S_2 := S + \Delta_2$). As indicated in Sections 6.2 and 6.3, detecting the particular degree of overlap between Δ_1 and Δ_2 based on trace equivalence and process schema isomorphism (cf. Definition 8 and Theorem 10) is not (always) recommendable due to the following reasons:

[5]Such dummy nodes (also called null tasks) are often used in connection with block–structured models, (e.g., BPEL4WS [5, 92] or WSM Nets) in order to preserve the block structuring.

[6]Note that for arbitrary graphs one has to analyze all $n!$ permutations of the adjacency matrix.

1. As discussed in Section 6.2 it is not possible to determine whether Δ_1 and Δ_2 are subsumption or partially equivalent (cf. Summary 1, Section 6.2). For this reason we cannot define and also cannot decide on these two particular degrees of overlap. It is only possible to check whether Δ_1 and Δ_2 are disjoint or equivalent based on trace equivalence between $S_1 + \Delta_2$ and $S_2 + \Delta_1$.
2. However, doing so requires the materialization of the resulting process schemes $S_1 + \Delta_2$ and $S_2 + \Delta_1$ or S_1 and S_2 respectively and the explicit verification of trace equivalence. This approach is not applicable in practice, especially in conjunction with a large number of running process instances.

For these reasons we have to find better suited approaches to define and decide on the particular degrees of overlap between Δ_1 and Δ_2. The information we can use for this purpose comprises process schemes S, S_1, and S_2 as well as changes Δ_1 and Δ_2. Intuitively, taking this information we come to the following three kinds of approaches:

1. **Structural Approaches** which directly compare process schemes S, S_1, and S_2
2. **Operational Approaches** directly contrasting changes Δ_1 and Δ_2 (i.e., looking at the two sets of applied change operations),
3. **Hybrid Approaches** (cf. Section 6.5) combining structural and operational approaches.

In the following we present these variants and systematically rate their particular strenghts and limitations.

6.4.1 Structural Approaches

The essence of all structural approaches is to compare resulting process schemes $S_1 := S + \Delta_1$ and $S_2 := S + \Delta_2$ to gain information about the degree of overlap between Δ_1 and Δ_2. A promising approach to analyze the difference between two process schemes, the so called *Delta Analysis*, has been presented by Guth and Oberweis in [45] and was used by van der Aalst and Basten in [117]. In [117], Delta Analysis is based on four inheritance relations on process schemes. Roughly speaking a process schema S_1 is a subclass of process schema S_2 if it can do everything S_2 can do and more. With this, for example, van der Aalst and Basten determine the *Greatest Common Divisor (GCD)* for process schemes S_1 and S_2 which represents the smallest common superclass of S_1 and S_2. However, this approach cannot be adopted to the problem described in this paper since it is not possible to directly compare the process schemes S_1 and S_2 in order to detect the degree of overlap between the changes Δ_1 and Δ_2. This is illustrated by the following example:

Example 6.5 (Greatest Common Divisor Approach:) Consider process schemes S_1 and S_2 (represented by WF Nets [118] introduced in Section 2.3) as depicted in Figure 6.5a). Applying the

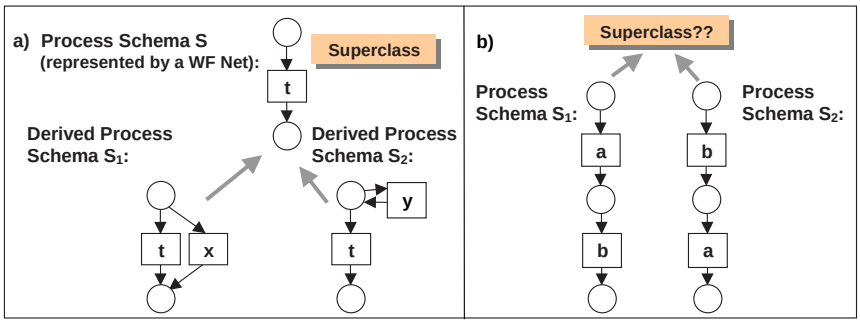


Figure 6.5: Determining the Greatest Common Divisor (Examples)

approach suggested by van der Aalst and Basten [13, 117, 116] we start from process schemes S_1 and S_2 and determine the common superclass S. By contrast, in our approach we already have a common divisor S and derive process schemes S_1 and S_2 by applying Δ_1 and Δ_2 respectively.

However, considering the Delta Analysis approach we can already recognize one common limitation of all structural approaches: they are not able to adequately deal with order–changing operations. One example is depicted in Figure 6.5b) where we cannot find a process schema which represents a common behavior for schemes S_1 and S_2.

As a second possibility, consider the so called *pure structural approach* (cf. Figure 6.10). Here we exploit the set–based representation of WSM Nets (cf. Section 3.1.1) and directly compare activity sets N_1 and N_2, edge sets $CtrlE_1$ and $CtrlE_2$, $SyncE_1$ and $SyncE_2$, $DataE_1$ and $DataE_2$, $LoopE_1$ and $LoopE_2$, and data element sets D_1 and D_2 regarding the two process schemes S_1 and S_2 with $S_i = (N_i, D_i, \ldots), i = 1, 2$. However, doing so is unnecessarily expensive. Actually, we do not have to compare "whole" activity and edge sets of S_1 and S_2 since these sets have been derived starting with same original schema S, i.e., starting with the same activity and edge sets. In other words we already know a common divisor $S = (N, D, \ldots)$ for S_1 and S_2. Therefore we can reduce complexity by exploiting the common ancestry of S_1 and S_2 what results in a third method which we call *aggregated structural approach* (cf. Figure 6.10). More precisely, the aggregated structural approach works by comparing differences between process schema S_1 and original schema S and between process schema S_2 and original schema S. These differences can be easily determined by building the following difference sets:

Definition 13 (Difference Sets fo Control Flow And Data Flow Changes) *Let $S = (N, D, NT, CtrlE, SyncE, LoopE, DataE)$ be a (correct) process schema and let Δ be a change transforming S into another (correct) process schema $S' = (N', D', \ldots)$. Then the difference sets regarding control and data flow can be determined as follows:*

1. *Control Flow Difference Sets:*
   - $N_\Delta^{add} := N' \setminus N$ *and* $N_\Delta^{del} := N \setminus N'$
   - $CtrlE_\Delta^{add} := CtrlE' \setminus CtrlE$ *and* $CtrlE_\Delta^{del} := CtrlE \setminus CtrlE'$

- $SyncE_{\Delta}^{add} := SyncE' \setminus SyncE$ *and* $SyncE_{\Delta}^{del} := SyncE \setminus SyncE'$
- $LoopE_{\Delta}^{add} := LoopE' \setminus LoopE$ *and* $LoopE_{\Delta}^{del} := LoopE \setminus LoopE'$

2. *Data Flow Difference Sets:*

- $D_{\Delta}^{add} := D' \setminus D$ *and* $D_{\Delta}^{del} := D \setminus D'$
- $DataE_{\Delta}^{add} := DataE' \setminus DataE$ *and* $DataE_{\Delta}^{del} := DataE \setminus DataE'$ □

We now apply Definition 13 to detect the particular degree of overlap between changes Δ_1 and Δ_2 on original schema S. For this purpose the following difference sets (cf. Definition 13) have to be compared:

1. *Control Flow Difference Sets to be compared:*

- $N_{\Delta_1}^{add}$ and $N_{\Delta_2}^{add}$
- $N_{\Delta_1}^{del}$ and $N_{\Delta_2}^{del}$
- $CtrlE_{\Delta_1}^{add}$ and $CtrlE_{\Delta_2}^{add}$
- $CtrlE_{\Delta_1}^{del}$ and $CtrlE_{\Delta_2}^{del}$
- $SyncE_{\Delta_1}^{add}$ and $SyncE_{\Delta_2}^{add}$
- $SyncE_{\Delta_1}^{del}$ and $SyncE_{\Delta_2}^{del}$
- $LoopE_{\Delta_1}^{add}$ and $LoopE_{\Delta_2}^{add}$
- $LoopE_{\Delta_1}^{del}$ and $LoopE_{\Delta_2}^{del}$

2. *Data Flow Difference Sets to be compared:*

- $D_{\Delta_1}^{add}$ and $D_{\Delta_2}^{add}$
- $D_{\Delta_1}^{del}$ and $D_{\Delta_2}^{del}$
- $DataE_{\Delta_1}^{add}$ and $DataE_{\Delta_2}^{add}$
- $DataE_{\Delta_1}^{del}$ and $DataE_{\Delta_2}^{del}$

To illustrate the general results of this section achieved so far we apply them to concurrent process type and process instance changes Δ_T and Δ_I on process schema S. We provide the following example:

Example 6.6 (Application Of Structural Approach To Concurrent Activity Insertion): Consider Figure 6.6. Both Δ_T and Δ_{I_1} serially insert activity X at the same position ("between B and C") into S whereas Δ_{I_2} serially inserts another activity Y between A and B. Obviously, Δ_T and Δ_{I_1} overlap since they offend against claim (2) for disjoint changes (cf. Definition 10). Using the aggregated structural approach, we obtain $N_{\Delta_T}^{add} = N_{\Delta_{I_1}}^{add} = \{X\}$. This corresponds to the expected result, i.e., the multiple insertion of same activity X. Regarding instance I_2 on S, Δ_T and Δ_{I_2} are disjoint according to Def. 10. Application of the aggregated structural approach results in $N_{\Delta_T}^{add} \cap N_{\Delta_{I_2}}^{add} = \emptyset$, $N_{\Delta_T}^{del} \cap N_{\Delta_{I_2}}^{del} = \emptyset$, $CtrlE_{\Delta_T}^{add} \cap CtrlE_{\Delta_{I_2}}^{add} = \emptyset$, and $CtrlE_{\Delta_T}^{del} \cap CtrlE_{\Delta_{I_2}}^{del} = \emptyset$. Interpreting this result, we can state that Δ_T and Δ_{I_2} are disjoint.

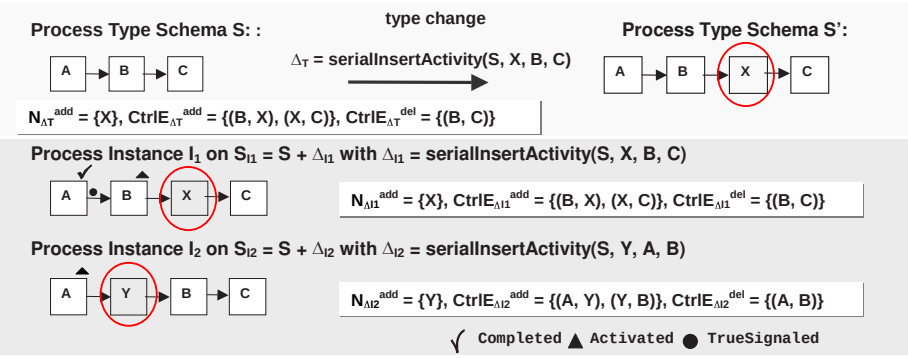


Figure 6.6: Application of Structural Approach to Concurrent Activity Insertion

These first two examples show that the aggregated structural approach works fine for insert (and delete) operations. We are able to precisely determine which activities have been inserted or deleted. In contrast, for move operations the aggregated structural approach (and consequently the pure structural approach) may be too imprecise. This is shown by the following example of changes Δ_T and Δ_I concurrently applied at the process type and process instance level:

Example 6.7 (Application Of The Structural Approach To Concurrent Activity Shifting): Consider Figure 6.7. For all three changes Δ_T, Δ_{I_1}, and Δ_{I_2} applied to schema S, $N^{add}_{\Delta_T} = N^{add}_{\Delta_{I_1}} = N^{add}_{\Delta_{I_2}} = \emptyset$ and $N^{del}_{\Delta_T} = N^{del}_{\Delta_{I_1}} = N^{del}_{\Delta_{I_2}} = \emptyset$ holds (no activity has actually been inserted or deleted). Determining the sets of newly inserted and deleted control edges for Δ_T and Δ_{I_1} yields $CtrlE^{add}_{\Delta_T} = CtrlE^{add}_{\Delta_{I_1}} = \{(A,C),(C,B),(B,D)\}$ and $CtrlE^{del}_{\Delta_T} = CtrlE^{del}_{\Delta_{I_1}} = \{(A,B),(B,C),(C,D)\}$ respectively. From this result it could be concluded that $\Delta_T \equiv \Delta_{I_1}$ holds, but it is not exactly observable which activity has been actually moved. In contrast, comparing respective edge sets for Δ_T and Δ_{I_2}, we obtain non–disjoint difference sets of control edges, i.e., $CtrlE^{add}_{\Delta_T} \cap CtrlE^{add}_{\Delta_{I_2}} \neq \emptyset$ and $CtrlE^{del}_{\Delta_T} \cap CtrlE^{del}_{\Delta_{I_2}} \neq \emptyset$. This indicates that $\Delta_T \cap \Delta_{I_2} \neq \emptyset$ holds. However, no further statement is possible since we cannot derive whether Δ_T and Δ_{I_2} are subsumption or partially equivalent. Therefore results based on structural approaches are too imprecise in conjunction with order–changing operations since we cannot exactly determine which activity has been actually moved. For example, in case of Δ_T and Δ_{I_1}, activity C as well as activity B could have been moved. When comparing Δ_T with Δ_{I_2} we can only conclude that these changes actually overlap but we are not able to make further statements.

One may claim that a solution for the above drawback of structural approaches is to "map" a move operation onto respective delete and insert operations. Type change Δ_T depicted in Figure 6.7, for example, could be translated into $deleteActivity(S, B)$ operation followed by $serialInsertActivity(S, B, C, D)$ operation. However, doing so is not sufficient in conjunction with structural approaches. Activity C is actually neither deleted nor inserted what is reflected by difference sets $N^{add}_{\Delta_T} = \emptyset$ and $N^{del}_{\Delta_T} = \emptyset$.

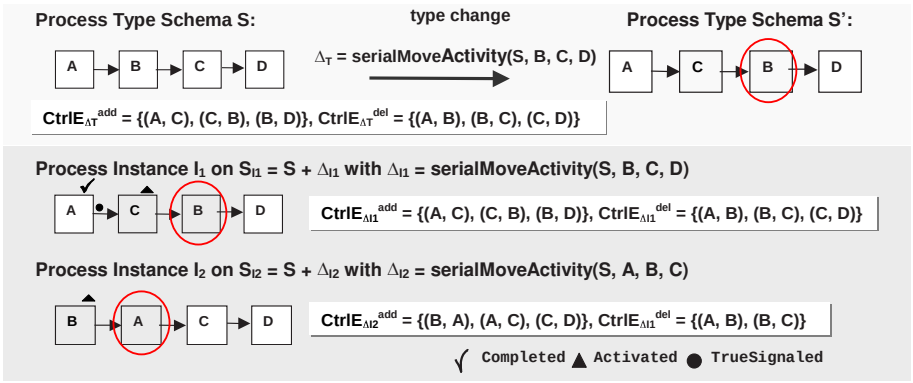


Figure 6.7: Application of the Structural Approach to Concurrent Activity Shifting

The limitations coming with applying structural approaches (cf. Summary 2) are aggravated if changes comprise several move operations. Taking this imprecise information it is not possible to derive adequate migration strategies in the following.

Summary 2 (Advantages and Limitations of Structural Approaches) *Let S be a (correct) process schema and let Δ_1 and Δ_2 be two changes transforming S into (correct) process schemes $S_1 := S + \Delta_1$ and $S_2 := S + \Delta_2$ respectively. Applying the (aggregated) structural approach by comparing the difference sets between S_1 and S_2 (cf. Definition 13) has the following advantages and limitations:*

- **Advantages:** *precise information concerning insert and delete operations (based on $N^{add}_{\Delta_i}$ and $N^{del}_{\Delta_i}$ $(i = 1, 2)$)*
- **Limitations:** *imprecise information concerning order–changing operations*

6.4.2 Operational Approach

A solution to overcome the drawback of structural approaches in conjunction with order–changing operations – not knowing which activities have been actually moved (cf. Summary 2) – may be to directly compare the applied changes Δ_1 and Δ_2. Obviously, Δ_1 and Δ_2 contain precise information about the applied change operations in general and about the actually moved activities in particular. However, this operational approach also shows limitations. Change logs may contain information about change operations which actually have no or only hidden effects on the underlying process schema. As already explained users who define changes (i.e., process designers or end users) do not always act in a goal–oriented way when modifying a process schema. In fact they may try out the best solution resulting in noisy information within the change logs. To get the idea behind in the following let Δ be an arbitrary change log applied to a (correct) process schema S what results in another process schema $S' := S + \Delta$.

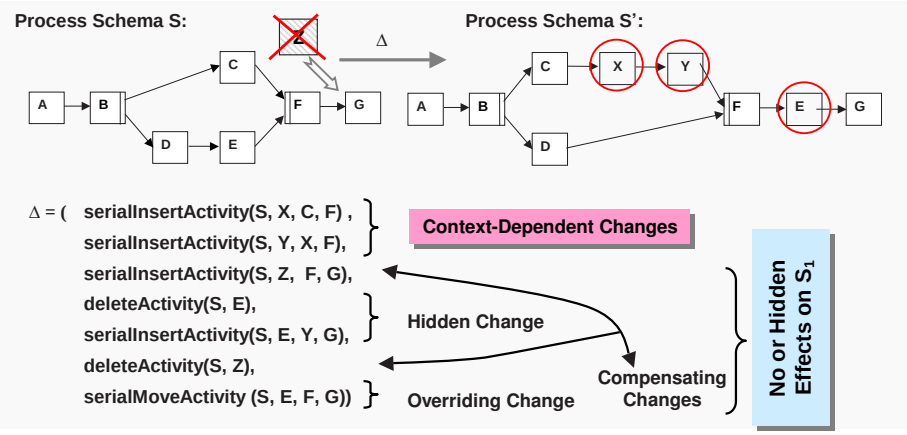


Figure 6.8: Noisy Process Change Log(Example)

1. The first group of changes without any effects on S' are *compensating changes*, i.e., change operations mutually compensating their effects.

   Example 6.8.a (Compensating Changes): A simple example is depicted in Figure 6.8, where activity Z is first inserted (between F and G) and afterwards deleted by the user. Consequently, the respective operations $serialInsertActivity(S, Z, F, G)$ and $deleteActivity(S, Z)$ have no visible effect on S'.

2. The second category of noise in change logs comprises changes which only have hidden effects on S'. Such *hidden changes* always arise from deleting an activity which is then inserted again at another position. This actually has the effect of a move operation:

   Example 6.8.b (Hidden Changes): An example is given in Figure 6.8 where activity E is first deleted an then re–inserted between Y and G. The effect behind is the same as of the move operation $serialMoveActivity(S, E, Y, G)$.

3. There are changes overriding preceding changes (note that a change transaction is an ordered series of single change operations).

   Example 6.8.c (Overriding Changes): Again consider Fig. 6.8 where the effect of the hidden move operation $serialMoveActivity(S, E, Y, G)$ is overwritten by move operation $serialMoveActivity(S, E, F, G)$, i.e., in S' activity E is finally placed between F and G.

However, the presence of compensating, hidden, or overriding changes within a change transaction is a cumbersome problem but we can find methods to *purge* a log from these kinds of changes (cf. Section 6.5). As we will see, doing so is essential in order to find a canonical and minimal view on change logs. This, in turn, is necessary to be able to determine which activities actually have been moved by a change.

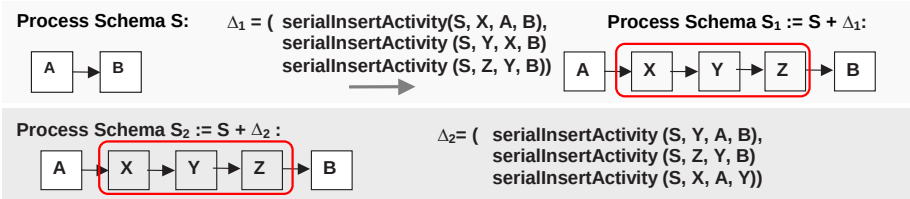


Figure 6.9: Equivalent Process Type and Instance Changes (Example)

A more severe limitation of the operational approach is its disability to adequately deal with *context–dependent changes*, i.e., changes which are mutually based on each other. An example is depicted in Figure 6.8: First, activity X is inserted serially between C and F. Based on this a second activity Y is inserted between $\underline{X}$ and F. Obviously, the second insertion uses the newly added activity of the first insertion as change context.

Why are such context–dependent process changes critical when applying the operational approach?

Example 6.9 (Context–Dependent Changes): Figure 6.9 illustrates the underlying problem. Obviously, Δ_1 and Δ_2 are equivalent since S_1 and S_2 are trace equivalent (cf. Definition 8). Unfortunately, this equivalence relation cannot be determined based on the depicted change logs (capturing Δ_1 and Δ_2) since the single insert operations for X, Y, and Z have been applied in different order within Δ_1 and Δ_2. Therefore the operational approach sketched so far would only detect an overlapping (multiple insertion of same activities) but would not be able to determine the degree of overlap, i.e., the total equivalence between Δ_1 and Δ_2.

Summary 3 (Advantages And Limitations Of Operational Approaches) *Let S be a (correct) process schema and let Δ_1 and Δ_2 be two changes transforming S into (correct) process schemes $S_1 := S + \Delta_1$ and $S_2 := S + \Delta_2$ respectively. Applying the operational approach by directly comparing changes Δ_1 and Δ_2 we observe the following advantages and limitations:*

- **Advantages:** *precise information for order–changing operations*
- **Limitations:** *– noisy change logs containing compensating, hidden, or overriding changes*
  – context–dependent changes

At this point a very important conclusion is that structural approaches have no problems with context–dependent changes.

Example 6.9.b (Context–Dependent Changes): Consider again Figure 6.9. Applying the aggregated structural approach (cf. Section 6.4.1) we obtain
$N^{add}_{\Delta_1} = N^{add}_{\Delta_2}$, $N^{del}_{\Delta_1} = N^{del}_{\Delta_2}$, $CtrlE^{add}_{\Delta_1} = CtrlE^{add}_{\Delta_2}$, and $CtrlE^{del}_{\Delta_1} = CtrlE^{del}_{\Delta_2}$
(all other difference sets are empty). Consequently, $\Delta_1 \equiv \Delta_2$ holds.

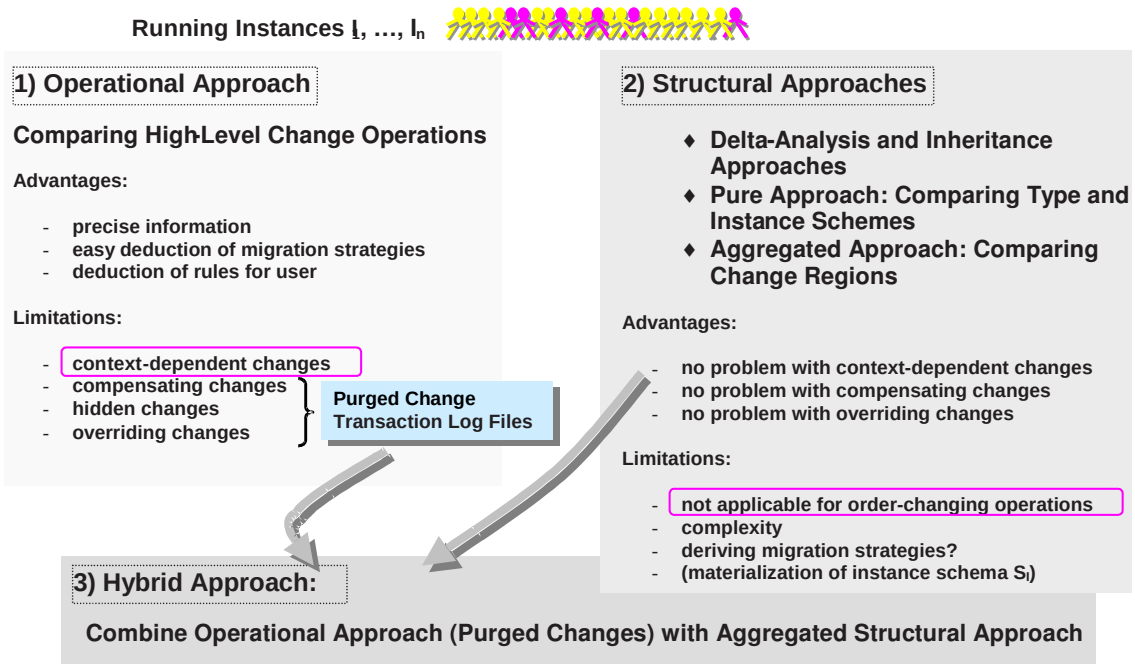


Figure 6.10: Approaches for Detecting Degree of Overlap Between Concurrent Changes

Taking Summaries 2 and 3 we now have the following situation (cf. Figure 6.10): Structural approaches are able to cope with context–dependent changes as well as with compensating, hidden and overriding changes. The reason is that structural approaches are based on the actual effects on a process schema. However, they are unable to adequately deal with order–changing operations. In contrast, when applying the operational approach we are able to precisely determine which activities have been moved but we are not able to handle context–dependent changes. Altogether, in the following section we combine both methods to a *hybrid approach* in order to exploit the particular strengths and to overcome the particular limitations.

6.5 Hybrid Approach

As discussed in the previous section, structural and operational approaches are not sufficient for correctly determining the degree of overlap between process type and process instance changes. The challenge is now to find an approach which overcomes these drawbacks. The hybrid approach presented in the following combines elements of structural and operational approaches (cf. Section 6.4) by exploiting there particular strengths and by overcoming their particular limitations (cf. Summaries 2 and 3). Altogether, this hybrid approach actually allows the formal definition of the still missing relations on process changes – subsumption and partial equivalence. Finally, it provides the basis to choose the right migration strategy along the particular

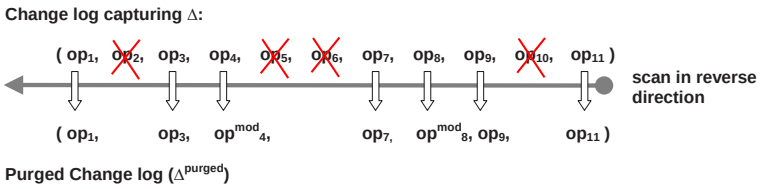


Figure 6.11: Basic Principle of Purging Change Logs

degree of overlap between process changes. Therefore, in Section 6.5.1, we present a method to purge change logs from noise like overriding, hidden, or compensating changes (cf. Section 6.4.2). Based on purged change logs it becomes possible to exactly determine which activities have been actually moved [99]. For other change operations like the insertion of activities the difference sets (cf. Definition 13) can be used.

6.5.1 Purging Change Logs

In this section, we provide an algorithm purging noise from change logs. For this purpose, let S be a (correct) process schema and let Δ be a change which transforms S into another (correct) process schema S'. Informally, the algorithm works as follows: Firstly, the control and data flow difference sets for change Δ on S, e.g., the set N_{Δ}^{add} of newly inserted activities and the set N_{Δ}^{del} of deleted activities (cf. Definition 13, cf. Section 6.4.1) are determined (*Structural Approach*). Taking this information the change log capturing Δ is *purged*. More precisely, this purging is accomplished by scanning the change log of $\Delta = (op_1, \ldots, op_n)$ in reverse direction and by determining whether change operation op_i $(i = 1, \ldots, n)$ has actually any effect on S. If so we incorporate op_i into an new – intially empty – change log Δ^{purged} (cf. Figure 6.11). Finally, in order to reduce the number of necessary change log scans to one we use auxiliary sets to memorize which activities, sync edges, data elements, and data edges have been already treated. The following informal description focuses on the insertion, deletion, and moving of activities in order to get the idea behind the respective algorithm (cf. Algorithm 9 in Appendix D). However, the used methods can be also applied to purge logs capturing information about the insertion and deletion of sync edges, data elements, and so on[7].

- Assume that we find a log entry op_i for an operation inserting activity X between activities *src* and *dest* into S and that X has not been considered so far, i.e., op_i is the last change operation within Δ[8] which manipulates X. If X has been already present in S ($X \notin N_{\Delta}^{add}$) a *hidden change* is found (cf. Sect. 6.4.2). Consequently, a respective log entry for an operation moving X between *src* and *dest* is created and written into Δ^{purged}.

[7]Note that Algorithm 9 splits block operations (like e.g., inserting blocks) into the respective single operations (e.g., inserting the activities contained in the respective block).

[8]Note that we traverse the change logs in reverse order.

- If log entry op_i denotes an operation deleting activity X from S and X has been not considered so far but X is still present in S' ($X \notin N_{\Delta}^{del}$) then we have found a *compensating change*. Therefore op_i (and the respective insert operation) are left outside Δ^{purged}.

- If log entry op_i denotes an operation moving activity X between activities src and $dest$ and op_i is the last operation working on X within Δ we have to distinguish between two cases: If X has been inserted befort op_i (i.e, $X \in N_{\Delta}^{add}$) we write a new log entry in Δ^{purged} denoting an operation inserting X between src and $dest$. If X has been also present in S ($X \notin N_{\Delta}^{add}$) we write op_i unalteredly into Δ^{purged}.

Of course, the purged change logs resulting from Algorithm 9 are not correct in the sense that they can be "replayed" on S in order to obtain S'. Purged change logs are only used to determine set N_{Δ}^{move} of activities which have been moved by Δ. Among other things this will serve as the basis for determining the degree of overlap between concurrently applied changes (respective defintions and strategies are provided in Section 6.6).

Definition 14 (Purging Changes and Consolidated Sets) *Let S = (N, D, ...) be a (correct) process schema. Let further Δ be a change transaction which transforms S into another (correct) process schema S' = ($N', S', \ldots$). Then the purged representation of Δ, Δ^{purged} and the set of actually moved activities N_{Δ}^{move} can be determined as described by Algorithm 9 (cf. Appendix D).*

How Algorithm 9 works is illustrated by the following example.

Example 6.12 (Purging a Change Log): Figure 6.12 illustrates the mode of operation of Algorithm 9 applied to the log capturing change Δ_1 in Figure 6.8. Initially, Algorithm 9 determines the set of newly inserted and deleted activities regarding schema S, i.e., $N_{\Delta_1}^{add} = \{X, Y\}$ and $N_{\Delta_1}^{del} = \emptyset$ hold. Based on this information change log Δ_1 is run through once (in reverse direction) and purged from noisy operations op_6, op_5, op_4, and op_3. Algorithm 9 finishes with purged change transaction Δ_1^{purged} as depicted in Figure 6.12. Based on this purged change log the set of activities actually moved by Δ_1 can be determined as $N_{\Delta_1}^{move} = \{E\}$. Together with the set of newly inserted and deleted activities we obtain consolidated activity sets ($N_{\Delta_1}^{add}, N_{\Delta_1}^{del}, N_{\Delta_1}^{move}$) = ($\{X, Y\}, \emptyset, \{E\}$).

However, to be able to define subsumption and partially equivalent changes we need further information about the target context of insert operations. Based on the consolidated activity sets (cf. Definition 14) we can only check, for example, whether the same set of activities has been inserted but we do not recognize in which target context. However, this is information important in order to distinguish between equivalent changes ("same activities into same target context") and partially equivalent changes ("same activities into different context") for example.

Change Log (in reverse order):	Initialization:	Purged Change Log
	$A = \emptyset$;	
	$N_{\Delta 1}^{add} = \{X, Y\}$;	
	$N_{\Delta 1}^{del} = \emptyset$;	
Δ_1= (		Δ_1^{purged}= (
op_7 = serialMoveActivity(S, E, F, G),	$E \notin A \Rightarrow A = \{E\}$;	
	$E \notin N_{\Delta 1}^{add} \wedge$ new pos. $\Rightarrow$	op_3 = serialMoveActivity (S, E, F, G),
op_6 = deleteActivity(S, Z),	$Z \notin A \Rightarrow A = \{E, Z\}$;	
	$Z \notin N_{\Delta 1}^{del}$;	
op_5 = serialInsertActivity (S, E, Y, G),	$E \in A$;	
op_4 = deleteActivity(S, E),	$E \in A$;	
op_3 = serialInsertActivity (S, Z, F, G),	$Z \in A$;	
op_2 = serialInsertActivity (S, Y, X, F),	$Y \notin A \Rightarrow A = \{E, Z, Y\}$;	
	$Y \in N_{\Delta 1}^{add} \Rightarrow$	op_2 = serialInsertActivity (S, Y, X, F),
op_1 = serialInsertActivity (S, X, C, F))	$X \notin A \Rightarrow A = \{E, Z, Y, X\}$;	
	$X \in N_{\Delta 1}^{add} \Rightarrow$	op_1 = serialInsertActivity (S, X, C, F),

Figure 6.12: Purging a Change Log

6.5.2 Anchor Sets and Order Sets

In this section we want to extend the current approach towards the ability to define all degrees of overlap between process changes. More precisely, we still need two kinds of information. At first it is necessary to know the target context of insert and order–changing operations. Secondly, we also need to know the order in which activities have been inserted into or moved to the target context. As mentioned, this information is required in order to correctly classify concurrent changes according to their degree of overlap.

Example 6.13.a (Insertion into Different Context): Consider Figure 6.13 where process type change Δ_T inserts activity sequence $X \rightarrow Y$ between activities B and C. In addition, an instance–specific change Δ_1 has inserted X and Y serially between B and C. This results in consolidated activity sets ($N^{add}_{\Delta_{I_1}} = \{X, Y\}$, $N^{del}_{\Delta_{I_1}} = \emptyset, N^{move}_{\Delta_{I_1}} = \emptyset$). Obviously, Δ_T and Δ_{I_1} are equivalent. However, the instance–specific change Δ_{I_2} has the same consolidated activity set ($N^{add}_{\Delta_{I_2}} = \{X, Y\}$, $N_{\Delta^{del}_{I_2}} = \emptyset, N^{move}_{\Delta_{I_2}} = \emptyset$) though changes Δ_{I_2} and Δ_T are not equivalent – Δ_{I_2} has added the same activities X and Y to S but has embedded them in a different target context when compared to Δ_T (between activities C and D). Therefore, changes Δ_T and Δ_{I_2} are only partially equivalent.

What we can see from Example 6.13 is that it is not sufficient to base the determination of the degree of overlap solely on the comparison of their consolidated activity sets. We also need to evaluate the information about the target context of insert and move operations.

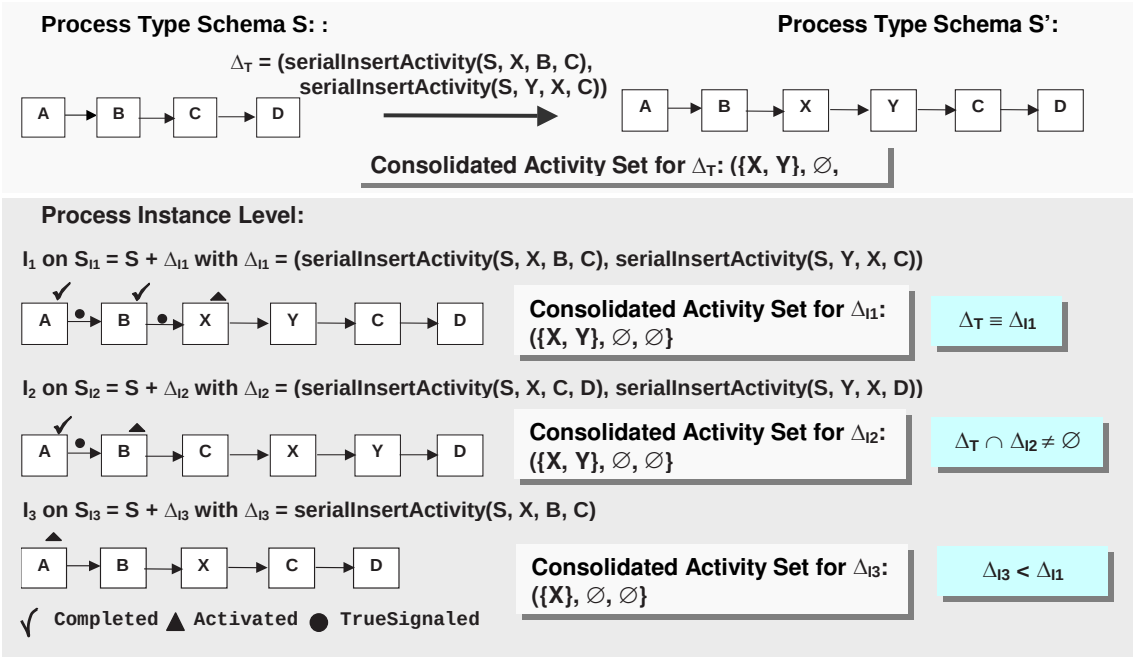


Figure 6.13: Insertion and Moving in Different Context

But on which information shall a comparison between the target context of newly inserted or moved activities be based on? Again problems arise in conjunction with context–dependent changes. The context of newly inserted activities, i.e., their direct successors and predecessors within the changed process schema, may also have been newly inserted or may consist of shifted activities. As an example consider the insertion contexts of activities X and Y (in S') as depicted in Figure 6.13. The insertion context of activity X, for example, is given by activities B and Y whereby activity Y has been newly inserted as well. Change Δ_{I_3} inserts activity X into S between activities B and C (this change would be subsumption equivalent to Δ_{I_1} since it inserts a subset of the activities added by Δ_{I_1} into S at the same target context). However, comparing the target context of both operations inserting X (Δ_1 inserts X between B and Y whereas Δ_2 inserts X between B and C) we cannot conclude that they are subsumption equivalent.

From the above example we can see the challenge we must deal with when determining the target context of newly inserted or shifted activities. We have to determine the target context of changes based on the original schema S in order to provide a common basis for their comparison. This, in turn, can be difficult if we apply context–dependent changes since in this case we have to "transfer" the context to activities which are already present within the original schema.

We have already discussed an approach to transfer a target context from newly inserted activities to activities already present in the original process schema: For the structural conflict test to detect deadlock causing cycles (via newly inserted sync links) we have developed special

graph reduction techniques [101]. They transfer the order relations set out by the newly inserted sync links to virtual sync links between activities within the original schema (cf. Algorithm 3, Section 5.4). Similarly, we can act in order to determine the target context for insert and move operations. The target context of an insert or move operation Δ applied to a process schema S is called the *anchor set* of Δ based on S. Formally:

Algorithm 4 (Anchors for Insert Operations) *Let S = (N, D, CtrlE, ...) be a (correct) process schema and let Δ be a change which transforms S into another (correct) process schema S' = (N', D', CtrlE',...). Let further N_Δ^{add} and N_Δ^{move} be the consolidated activity sets for insert and move operations (cf. Definition 14). Then the anchors of the insert operations applied within Δ – $AnchorIns(S, \Delta)$ – can be determined as follows:*

$AnchorIns(S, \Delta) = \emptyset$;

```
forall (X ∈ N_Δ^add) do
   find {(left, X), (X, right)} ∈ CtrlE';
   while (left ∈ N_Δ^add ∪ N_Δ^move))[9] do
      find (leftleft, left) ∈ CtrlE';
      left = leftleft;
   od
   while (right ∈ N_Δ^add ∪ N_Δ^move)) do
      find (right, rightright) ∈ CtrlE';
      right= rightright;
   od
od
```

$AnchorIns(S, \Delta) = AnchorIns(S, \Delta) \cup$ {(left, X, right)};

Anchor sets for move operations can be determined analogously.

Algorithm 5 (Anchors for Move Operations) *Let S = (N, D, CtrlE, ...) be a (correct) process schema and let Δ be a change which transforms S into another (correct) process schema S' = (N', D', CtrlE',...). Let further N_Δ^{add} and N_Δ^{move} be the consolidated activity sets for insert and move operations (cf. Definition 14). Then the anchors of the move operations applied within Δ – $AnchorMove(S, \Delta)$ – can be determined as follows:*

$AnchorMove(S, \Delta) = \emptyset$;

```
forall (X ∈ N_Δ^move) do
   find {(left, X), (X, right)} ∈ CtrlE';
   while (left ∈ N_Δ^add ∪ N_Δ^move)) do
      find (leftleft, left) ∈ CtrlE';
      left = leftleft;
   od
   while (right ∈ N_Δ^add ∪ N_Δ^move)) do
      find (right, rightright) ∈ CtrlE';
```

[9] It is not necessary to consider N_Δ^{del} here since deleted activities cannot build anchors for other activities.

```
        right= rightright;
    od
od
```
$AnchorMove(S, \Delta) = AnchorMove(S, \Delta) \cup \{$`(left, X, right)`$\};$

Applying Algorithms 4 + 5 we can determine the target context of an insert or move operation which is based on original schema S and not on the changed schema S'. This is important in order to obtain a common basis for comparing changes. Otherwise, for example, the original target context of an insertion may be not present in S' if it has been deleted by change Δ.

We illustrate the function of Algorithms 4 and 5 in the following example. Here, they are applied to a concrete scenario of concurrently applied changes at type and instance level:

Example 6.13.b (Determination Of Anchor Sets): Consider again Figure 6.13. Solely on the basis of the consolidated activity sets we cannot distinguish between "Δ_T and Δ_{I_1}" (being equivalent changes) and "Δ_T and Δ_{I_2}" (being partially equivalent changes). Both Δ_{I_1} and Δ_{I_2} insert the same activities but into a different target context. The specific target context of an insert operation can be determined by applying Algorithm 4. For type change Δ_T and instance–specific change Δ_{I_1} we obtain $AnchorIns(S, \Delta_T) = \{(B, X, C), (B, Y, C)\}$ and $AnchorIns(S, \Delta_{I_1}) = \{(B, X, C), (B, Y, C)\}$ respectively. As we can easily see $AnchorIns(S, \Delta_T) = AnchorIns(S, \Delta_{I_1})$ holds and therefore both changes can be considered as being equivalent. However, for Δ_{I_2} we obtain $AnchorIns(S, \Delta_{I_2}) = \{(C, X, D), (C, Y, D)\}$. From this we can conclude that Δ_T and Δ_{I_2} insert the same activities X and Y but into a different target context based on S. Consequently, they cannot be (subsumption) equivalent and are therefore regarded as being partially equivalent.

Generally, information about the consolidated activity sets and the respective anchor sets will not be sufficient. For example, two changes may insert the same activities into the same target context but in different order as the following example shows:

Example 6.14.a (Insertion into Different Order): Consider Figure 6.14 where Δ_T, Δ_{I_1}, and Δ_{I_2} all insert and move the same activities

$$(N^{add}_{\Delta_T} = N^{add}_{\Delta_{I_1}} = N^{add}_{\Delta_{I_2}} = \{X, Y\},\ N^{move}_{\Delta_T} = N^{move}_{\Delta_{I_1}} = N^{move}_{\Delta_{I_2}} = \{B\}).$$

As anchor sets we obtain

- $AnchorIns(S, \Delta_T) = \{(C, X, D), (C, Y, D)\}, AnchorMove(S, \Delta_T) = \{(C, B, D)\}$,
- $AnchorIns(S, \Delta_{I_1}) = \{(C, X, D), (C, Y, D)\}, AnchorMove(S, \Delta_{I_1}) = \{(C, B, D)\}$,
- $AnchorIns(S, \Delta_{I_2}) = \{(C, X, D), (C, Y, D)\}, AnchorMove(S, \Delta_{I_2}) = \{(C, B, D)\}$.

At first glance it seems that $\Delta_{I_1} \equiv \Delta_T$ and $\Delta_{I_2} \equiv \Delta_T$ holds. However, this is not true for Δ_{I_2} and Δ_T since Δ_{I_2} has inserted and moved the respective activities in a different order than Δ_T.

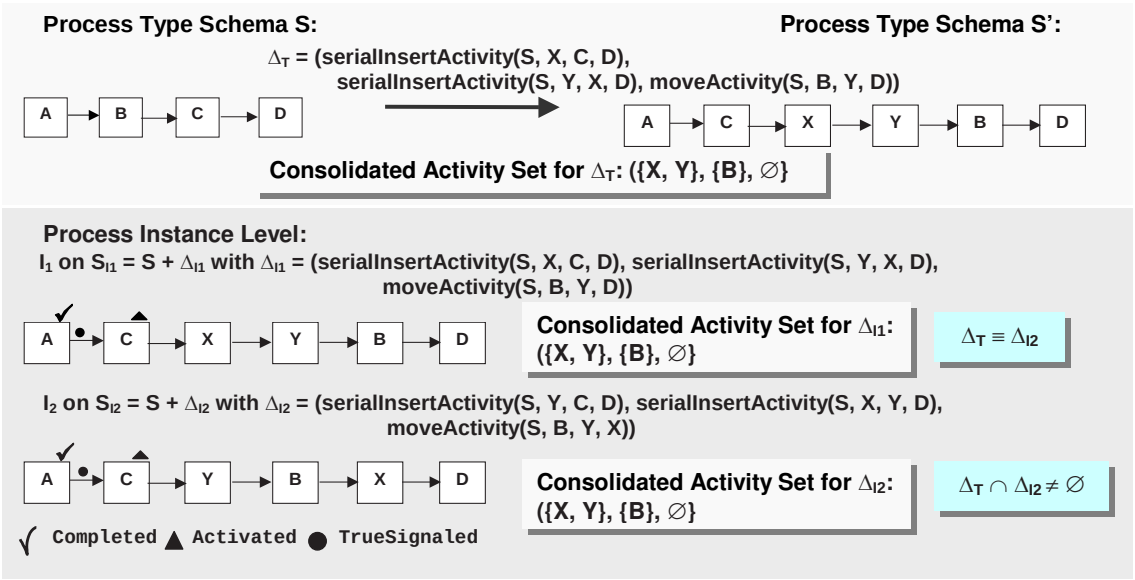


Figure 6.14: Insertion and Moving in Different Order

We additionally have to care about the order of newly inserted or shifted activities when comparing schema changes. The respective information can be gained by scanning the resulting control edge set after applying the change. As opposed to anchor sets we do not base our considerations about orders on the original schema S, but in the present case we have to look at the order given by the new schema S'. Of course, only those activities have to be compared regarding their order on S' which have been inserted into or moved to the same target context. More precisely, it is only necessary to consider the order of those activities which have the same anchors in S. Applying Definition 15 determines the respective grouping of activities with same anchors:

Definition 15 (Anchor Groups for Insert / Move Operations) *Let S = (N, D, ...) be a (correct) process schema and Δ be a change which transforms S into another (correct) process schema S' = (N', D', CtrlE', ...). Let further $AnchorIns(S, \Delta)$ and $AnchorMove(S, \Delta)$ be the anchor sets for insert and move operations captured by Δ (cf. Algorithms 4 + 5). Then the corresponding anchor group sets $AnchorGroupsIns(S, \Delta)$ and $AnchorGroupsMove(S, \Delta)$ contain groups of (newly inserted or moved) activities which have the same left and the same right anchor. Formally:*

$$AnchorGroupsIns(S, \Delta) = \{G = \{X_1, \ldots, X_n\} \mid$$
$$\forall X_i, X_j \in G : \exists\ (left_i,\ X_i,\ right_i),\ (left_j,\ X_j,\ right_j) \in AnchorIns(S, \Delta)\ with$$
$$left_i = left_j \wedge right_i = right_j\ (i, j = 1, \ldots, n)\}$$

$$AnchorGroupsMove(S, \Delta) = \{G = \{X_1, \ldots, X_n\} \mid$$

$\forall X_i, X_j \in G : \exists$ *(left$_i$, X_i, right$_i$), (left$_j$, X_j, right$_j$)* $\in AnchorMove(S, \Delta)$ *with left$_i$ = left$_j$* $\wedge$ *right$_i$ = right$_j$ (i, j = 1, ..., n)*$\}$

If there is only one activity within an anchor the respective anchor group becomes the empty set since there is no order relation to be considered. For the scenario depicted in Figure 6.14, for example, we obtain

- $AnchorGroupsIns(S, \Delta_T) = \{\{X, Y\}\}, AnchorGroupsMove(S, \Delta_T) = \emptyset$,
- $AnchorGroupsIns(S, \Delta_{I_1}) = \{\{X, Y\}\}, AnchorGroupsMove(S, \Delta_{I_1}) = \emptyset$,
- $AnchorGroupsIns(S, \Delta_{I_2}) = \{\{X, Y\}\}, AnchorGroupsMove(S, \Delta_{I_2}) = \emptyset$.

Based on Definition 15 we are able to determine the order between the activities contained within the same anchor group. Formally:

Algorithm 6 (Order Sets for Insert Operations) *Let S = (N, D, CtrlE, ...) be a (correct) process schema and* Δ *be a change which transforms S into another (correct) process schema S' = (N', E', ...). Let further* $AnchorGroupsIns(S, \Delta)$ *be the anchor group sets for insert operations captured by* Δ *(cf. Definitions 15). Then the order of the insert operations* $OrderIns(S, \Delta)$ *within* $AnchorGroupsIns(S, \Delta)$ *can be determined as follows:*

$OrderIns(S, \Delta) = \emptyset$;

```
forall (G = {X_1, ..., X_n} ∈ AnchorGroupsIns(S, Δ)) do {
   forall (X_i, X_j ∈ G) do
      if (X_i ∈ pred*(S', X_j))
         OrderIns(S', Δ) = OrderIns(S', Δ) ∪ {(X_i, X_j)};
      fi
      if (X_i ∈ succ*(S', X_j))
         OrderIns(S', Δ) = OrderIns(S', Δ) ∪ {(X_j, X_i)};
      fi
   od
od
```

The order sets for move operations can be analogously obtained:

Algorithm 7 (Order Sets for Move Operations) *Let S = (N, D, CtrlE, ...) be a (correct) process schema and* Δ *be a change which transforms S into another (correct) process schema S' = (N', E', ...). Let further* $AnchorGroupsMove(S, \Delta)$ *be the anchor group sets for move operations captured by* Δ *(cf. Definitions 15). Then the order of the move operations* $OrderMove(S, \Delta)$ *within* $AnchorGroupsMove(S, \Delta)$ *can be determined as follows:*

$OrderMove(S, \Delta) = \emptyset$;

```
forall (G = {X_1, ..., X_n} ∈ AnchorGroupsMove(S, Δ)) do
   forall (X_i, X_j ∈ G) do
   if (X_i ∈ pred*(S', X_j))
         OrderMove(S', Δ) = OrderMove(S', Δ) ∪ {(X_i, X_j)};
      fi
      if (X_i ∈ succ*(S', X_j))
         OrderMove(S', Δ) = OrderMove(S', Δ) ∪ {(X_j, X_i)};
      fi
   od
od
```

How the determination of anchor groups and order sets works and how it contributes to detect the degree of overlap between changes is shown in the following example. We apply Definition 15 and Algorithms 6 + 7 to the scenario depicted in Figure 6.14.

Example 6.14.b (Determining Anchor Groups and Order Sets): Consider the example from Figure 6.14. So far Δ_T and Δ_{I_1} as well as Δ_T and Δ_{I_2} have been regarded as being potentially equivalent. This assumption has been based on the comparison of the consolidated activity sets and anchor sets. However, when also taking the respective order sets into account this picture changes like follows:

- $AnchorGroupsIns(S, \Delta_T) = \{\{X, Y\}\}$; $AnchorGroupsMove(S, \Delta_T) = \emptyset$. We obtain $OrderIns(S, \Delta_T) = \{(X, Y)\}$; $OrderMove(S, \Delta_T) = \emptyset$
- $AnchorGroupsIns(S, \Delta_{I_1}) = \{\{X, Y\}\}$; $AnchorGroupsMove(S, \Delta_{I_1}) = \emptyset$. We obtain $OrderIns(S, \Delta_{I_1}) = \{(X, Y)\}$; $OrderMove(S, \Delta_{I_1}) = \emptyset$
- $AnchorGroupsIns(S, \Delta_{I_2}) = \{\{X, Y\}\}$; $AnchorGroupsMove(S, \Delta_{I_2}) = \emptyset$. We obtain $OrderIns(S, \Delta_{I_2}) = \{(Y, X)\}$; $OrderMove(S, \Delta_{I_2}) = \emptyset$

From this, we can see that Δ_T and Δ_{I_1} are actually equivalent but Δ_T and Δ_{I_2} are not. Though Δ_T and Δ_{I_2} have inserted the same activities into the same context, this has been done in different order. The latter can be easily seen from the resulting order sets. Therefore, we conclude that Δ_T and Δ_I are only partially equivalent.

In general, it is not sufficient to consider the order relations for insert and move operations in a separated way. We also have to include the order relations between newly inserted and moved activities into our considerations as the following example shows:

Example 6.15.a (Different Aggregated Order): Consider Figure 6.15 where process type change Δ_T inserts two activities X and Y between anchors C and D and then moves activity B to the position between X and Y. For instance I_1 as well as for I_2 the consolidated activity sets, the anchors sets, and even the order sets related to the insertions are the same (i.e., $OrderIns(S, \Delta_T) = OrderIns(S, \Delta_{I_1}) = OrderIns(S, \Delta_{I_2}) = \{(X, Y)\}$). By contrast, the

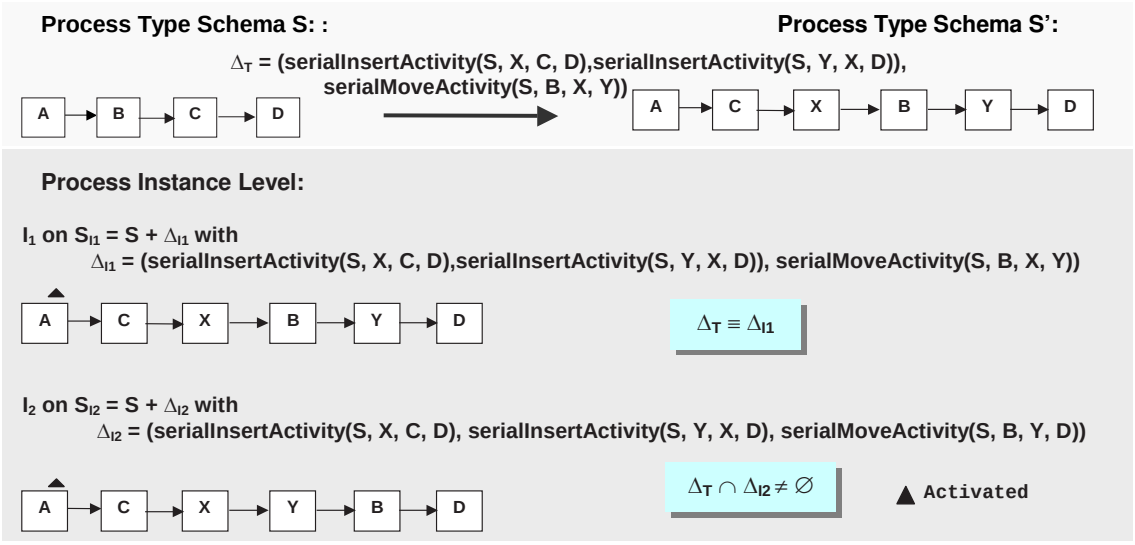


Figure 6.15: Different Aggregated Order

order sets for move operations are empty for Δ_T, Δ_{I_1} and Δ_{I_2}. Nevertheless, there is a difference between the relation of Δ_T and Δ_{I_1} which are equivalent and the relation of Δ_T and Δ_{I_2} which are actually partially equivalent.

To get the *aggregated order* relations for insert and move operations, first of all, we have to determine which activities have been inserted or shifted between the same anchors (related to S). Therefore, analogously to Definition 15, we detect the so called anchor groups for both, insert and move operations.

Definition 16 (Anchor Groups for Insert and Move Operations) *Let S = (N, E, ...) be a (correct) process schema and Δ be a change which transforms S into another (correct) process schema S' = (N', E', ...). Let further* $AnchorIns(S, \Delta)$ *and* $AnchorMove(S, \Delta)$ *be anchor sets for insert and move operations captured by Δ (cf. Algorithms 4 + 5). Then the anchor group set* $AnchorGroupsAgg(S, \Delta)$ *for both anchor sets* $AnchorIns(S, \Delta)$ *and* $AnchorMove(S, \Delta)$ *contains groups of activities which have been newly inserted or moved between the same left and right anchor. Formally:*

$$AnchorGroupsAgg(S, \Delta) = \{G = \{X_1, \ldots, X_n\} \mid \forall X_i, X_j \in G : \exists\ (left_i, X_i, right_i), (left_j, X_j, right_j) \in AnchorIns(S, \Delta) \cup AnchorMove(S, \Delta) \text{ with } left_i = left_j \wedge right_i = right_j\ (i, j = 1, \ldots, n)\}$$

Applying Definition 16 we obtain the following aggregated anchor groups for the scenario depicted in Figure 6.15:

- $AnchorGroupsAgg(S, \Delta_T) = \{\{X, Y, B\}\}$
- $AnchorGroupsAgg(S, \Delta_{I_1}) = \{\{X, Y, B\}\}$
- $AnchorGroupsAgg(S, \Delta_{I_2}) = \{\{X, Y, B\}\}$

Using Definition 16 we can determine the aggregated order for these operations:

Algorithm 8 (Order Sets for Insert and Move Operations) *Let S = (N, E, ...) be a (correct) process schema and Δ be a change which transforms S into another (correct) process schema S' = (N', E', ...). Let further $AnchorGroupsAgg(S, \Delta)$ be the anchor group sets for insert as well as for move operations captured by Δ (cf. Definition 16). Then the order of the insert and move operations $OrderAgg(S, \Delta)$ can be determined as follows:*

$OrderAgg(S, \Delta) = \emptyset$

```
forall (G = {X_1, ..., X_n} ∈ AnchorGroupsAgg(S, Δ)) do
   forall (X_i, X_j ∈ G) do
      if (X_i ∈ pred*(S', X_j))
         OrderAgg(S', Δ) = OrderAgg(S', Δ) ∪ {(X_i, X_j)};
      fi
      if (X_i ∈ succ*(S', X_j))
         OrderAgg(S', Δ) = OrderAgg(S', Δ) ∪ {(X_j, X_i)};
      fi
   od
od
```

Applying the aggregated order sets we can adequately deal with interactions between insert and move operations regarding their order. This is shown by the following example:

Example 6.15.b (Aggregated Order Sets): If we determine the aggregated order sets for changes Δ_T, Δ_{I_1}, and Δ_{I_2} (cf. Figure 6.15) we obtain

- $OrderAgg(S', \Delta_T) = \{(X, B), (B, Y), (X, Y)\}$
- $OrderAgg(S', \Delta_{I_1}) = \{(X, B), (B, Y), (X, Y)\}$
- $OrderAgg(S', \Delta_{I_2}) = \{(X, B), (Y, B), (X, Y)\}$

With these aggregated order sets we finally reached the milestone of being able to correctly determine the degree of overlap between process changes (cf. Figure 6.16). In summary, we are able to define the remaining degrees of overlap – subsumption and partially equivalent changes – as we will do in the following section.

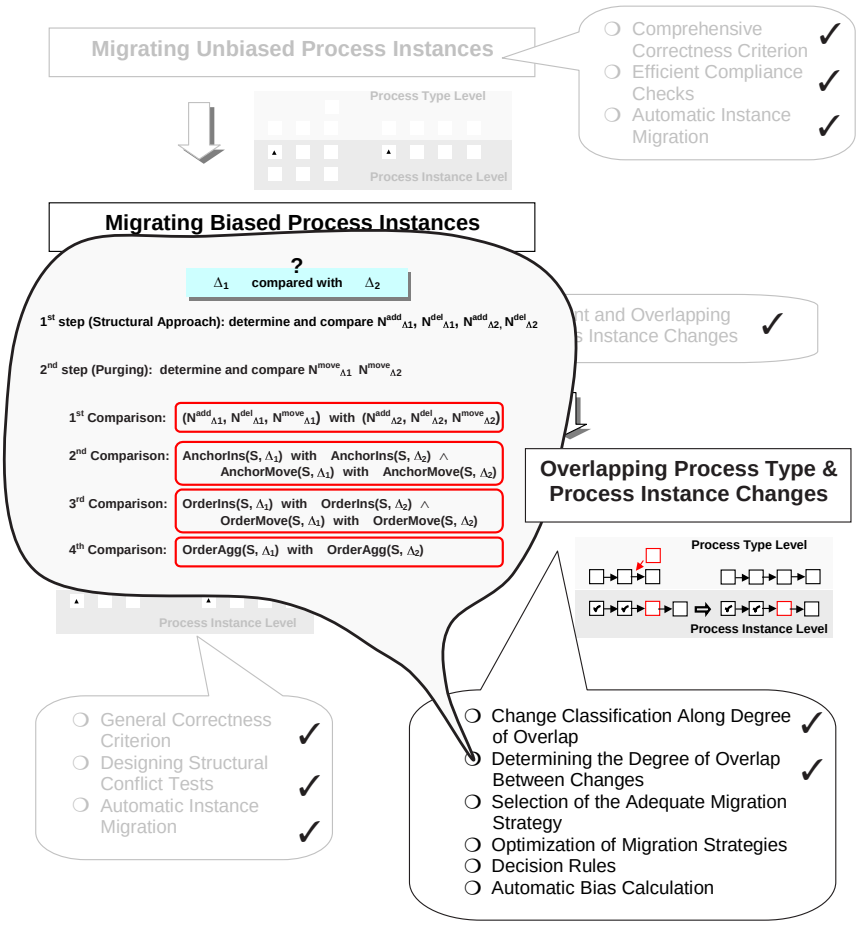


Figure 6.16: Determining Degree of Overlap Between Process Changes

6.6 Migration Strategies and Change Projections

So far it has not been possible to (formally) distinguish between subsumption and partially equivalent process changes though doing so is crucial for applying adequate migration strategies. In the previous section, we have introduced the notions of consolidated activity sets, anchor sets, and order sets in order to precisely determine the degree of overlap between process changes. Based on this we are now able to formally define subsumption equivalent and partially equivalent changes.

6.6.1 Subsumption and Partially Equivalent Changes

In the following let S be a (correct) process schema and let Δ_1 and Δ_2 be two changes which transform S into (correct) process schemes S_1 and S_2 respectively.

According to Summary 1 (cf. Section 6.2), Δ_1 is subsumption equivalent with Δ_2 (notation: $\Delta_1 \prec \Delta_2$) if Δ_2 subsumes the effects of Δ_1 on S, i.e., Δ_2 has the same effects on S as Δ_1 has but causes additional effects too. Δ_1 and Δ_2 are denoted as partially equivalent if they partially have the same effects on S, but both changes have specific effects on S as well.

We now formalize these two relations between changes Δ_1 and Δ_2. We base our considerations on the consolidated activity sets as well as the specific anchor and order sets of these two changes.

Definition 17 (Subsumption Equivalent Changes) *Let S be a (correct) process schema and Δ_1 and Δ_2 be two changes which transform S into (correct) process schemes S_1 and S_2 respectively. For Δ_i ($i = 1, 2$) let further $N^{add}_{\Delta_i}, N^{del}_{\Delta_i}$ and $N^{move}_{\Delta_i}$ be the consolidated activity sets, $AnchorIns(S, \Delta_i)$ and $AnchorMove(S, \Delta_i)$ the anchor sets, and $OrderIns(S, \Delta_i)$ and $OrderMove(S, \Delta_i)$ the order sets. Then:*
We denote Δ_1 as subsumption equivalent with Δ_2 (notation: $\Delta_1 \prec \Delta_2$) if the following conditions hold:

1. **Change Operations on Activity Sets:**
$$(N^{add}_{\Delta_1} \subseteq N^{add}_{\Delta_2} \wedge\ N^{del}_{\Delta_1} \subseteq N^{del}_{\Delta_2} \wedge\ N^{move}_{\Delta_1} \subseteq N^{move}_{\Delta_2}) \wedge$$
$$(AnchorIns(S, \Delta_1) \subseteq AnchorIns(S, \Delta_2) \wedge$$
$$AnchorMove(S, \Delta_1) \subseteq AnchorMove(S, \Delta_2)) \wedge$$
$$(OrderIns(S, \Delta_1) \subseteq OrderIns(S, \Delta_2), \wedge$$
$$OrderMove(S, \Delta_1) \subseteq OrderMove(S, \Delta_2) \wedge$$
$$OrderAgg(S, \Delta_1) \subseteq OrderAgg(S, \Delta_2))$$

2. **Change Operations on Sync And Loop Edges:**
$$(SyncE^{add}_{\Delta_1} \subseteq SyncE^{add}_{\Delta_2} \wedge\ SyncE^{del}_{\Delta_1} \subseteq SyncE^{del}_{\Delta_2}) \wedge$$
$$(LoopE^{add}_{\Delta_1} \subseteq LoopE^{add}_{\Delta_2} \wedge\ LoopE^{del}_{\Delta_1} \subseteq LoopE^{del}_{\Delta_2})$$

3. **Change Operations on Data Flow:**
$$(D^{add}_{\Delta_1} \subseteq D^{add}_{\Delta_2} \wedge\ D^{del}_{\Delta_1} \subseteq D^{del}_{\Delta_2}) \wedge$$
$$(DataE^{add}_{\Delta_1} \subseteq DataE^{add}_{\Delta_2} \wedge\ DataE^{del}_{\Delta_1} \subseteq DataE^{del}_{\Delta_2})$$

4. **Change Operations on Attributes:**
$$ChangedAttr_{\Delta_1} \subseteq ChangedAttr_{\Delta_2}$$

Two changes are subsumption equivalent if a real subset relation ("$\subset$") holds for at least one case in Definition 17. In particular, changes Δ_1 and Δ_2 are equivalent if for every relation

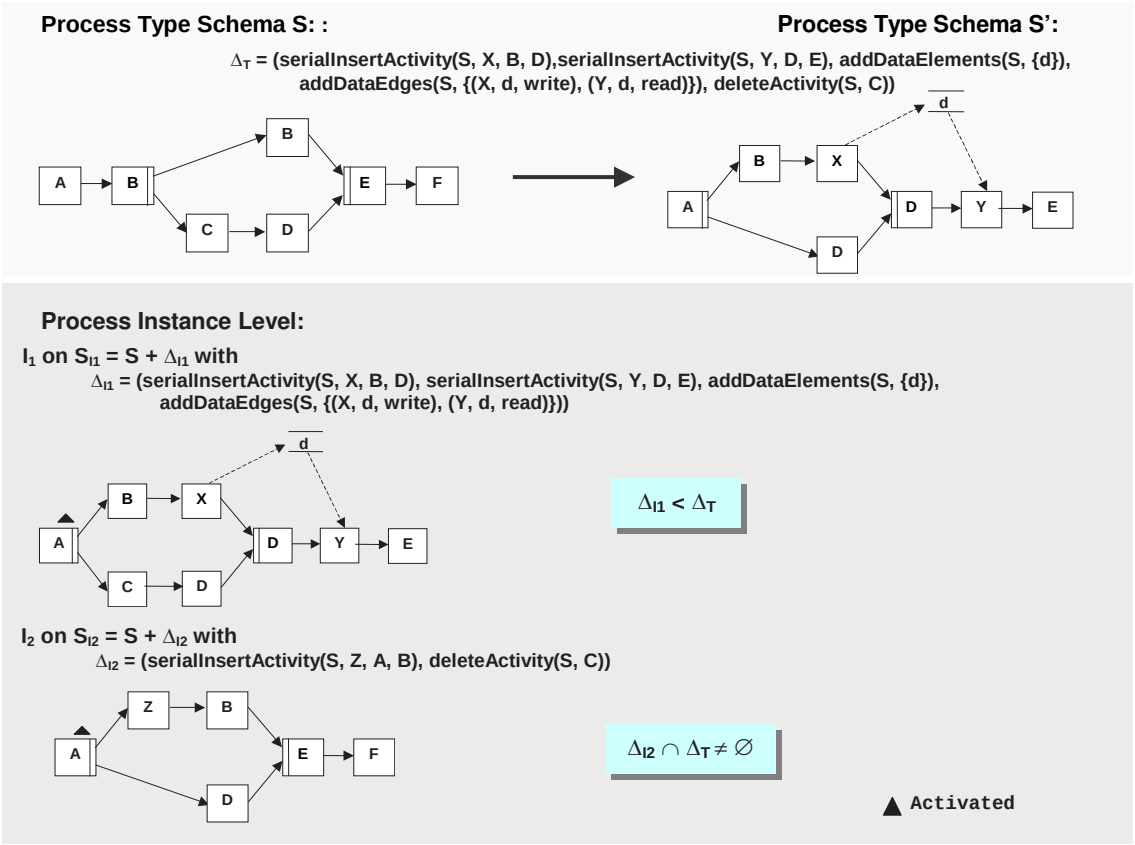


Figure 6.17: Subsumption and Partially Equivalent Changes

"=" holds as we will show in Section 6.6.4; i.e., equivalence can be considered as a special case of subsumption equivalence.

Example 6.17.a (Subsumption Equivalent Changes): Consider Figure 6.17. Type change Δ_T inserts two activites X and Y with a data dependency between them and deletes activity C. Instance change Δ_{I_1} also inserts activites X and Y and the respective data dependency but does not delete activity C. According to Definition 17 $\Delta_{I_1} \prec \Delta_T$ holds. The sets of inserted activites are equal (i.e., $N^{add}_{\Delta_I} = N^{add}_{\Delta_{I_1}}$) as well as the respective anchor and order sets. However, the set $N^{del}_{\Delta_{I_1}}$ of activities deleted by Δ_{I_1} constitutes a subset of $N^{del}_{\Delta_T}$.

Definition 18 (Partially Equivalent Changes) *Let S be a (correct) process schema and Δ_1 and Δ_2 be two changes which transform S into (correct) process schemes S_1 and S_2 respectively. For Δ_i ($i = 1, 2$) let further $N^{add}_{\Delta_i}, N^{del}_{\Delta_i}$, and $N^{move}_{\Delta_i}$ be the consolidated activity sets, $AnchorIns(S, \Delta_i)$ and $AnchorMove(S, \Delta_i)$ the anchor sets, and $OrderIns(S, \Delta_i)$ and $OrderMove(S, \Delta_i)$ the order sets. Then we denote Δ_1 and Δ_2 as being partially equivalent*

(notation: $\Delta_1 \between \Delta_2$) if the following conditions hold:

$$\neg(\Delta_1 \cap \Delta_2 = \emptyset) \wedge \neg(\Delta_1 \equiv \Delta_2) \wedge \neg(\Delta_1 \prec \Delta_2) \wedge \neg(\Delta_2 \prec \Delta_1)$$

$$\equiv$$

$$\neg(S_1 + \Delta_2 \equiv_{trace} S_2 + \Delta_1 \wedge N^{add}_{\Delta_1} \cap N^{add}_{\Delta_1} = \emptyset) \wedge$$
$$\neg(S_1 \equiv_{trace} S_2) \wedge$$
$$\neg(\Delta_1 \prec \Delta_2) \wedge \neg(\Delta_2 \prec \Delta_1) \ (\Psi)$$

Two changes Δ_1 and Δ_2 are partially equivalent if they are not disjoint, equivalent, or subsumption equivalent, i.e., partial equivalence can be defined as the complement of all other change relations.

Example 6.17.b (Partially Equivalent Changes): Consider again Figure 6.17 where instance change Δ_{I_2} has inserted activity Z between A and B. Additionally, Δ_{I_2} has deleted activity C. Obviously, Δ_T and Δ_{I_2} are not commutative since Δ_T cannot be applied to S_{I_2}. Consequently, Δ_T and Δ_{I_2} are not disjoint. S_{I_2} and $S' := S + \Delta_T$ are not trace equivalent and therefore, Δ_T and Δ_{I_2} are not equivalent. According to Definition 17, Δ_T and Δ_{I_2} are also not subsumption equivalent since $N^{add}_{\Delta_T} \not\subset N^{add}_{\Delta_{I_2}} \wedge N^{add}_{\Delta_{I_2}} \not\subset N^{add}_{\Delta_T}$ holds. Altogether, we obtain partial equivalence of Δ_T and Δ_{I_2} (both changes delete the same activity C).

We have provided formal definitions for all kinds of overlapping changes. In the following section we complement these results with adequate migration strategies.

6.6.2 On Selecting Migration Strategies for Overlapping Process Changes

According to the particular degree of overlap between process type and process instance changes different migration strategies are to be applied. In this section we provide adequate strategies for migrating process instances with equivalent and subsumption equivalent bias.

Let S be a (correct) process type schema and let $I = (S, \Delta_I, \ldots)$ be a process instance running on S with instance–specific bias Δ_I. Let further Δ_T be a process type change which transforms S into another (correct) schema S'.

For equivalent, subsumption equivalent and disjoint process type and process instance changes we provide migration strategies which can be automatically applied by the process management system. In particular, one deposits these migration strategies within the PMS. If, for example, an equivalent change takes place the system correctly reports this to the user as, for example, done in our proof–of–concept prototype (cf. Chapter 7). Furthermore, the PMS offers a default migration strategy that consists of re–linking the instance to the new process type schema (and of concomitantly adapting instance state and instance–specific bias to the empty set). Of course, the user is not forced to take over this default migration strategy.

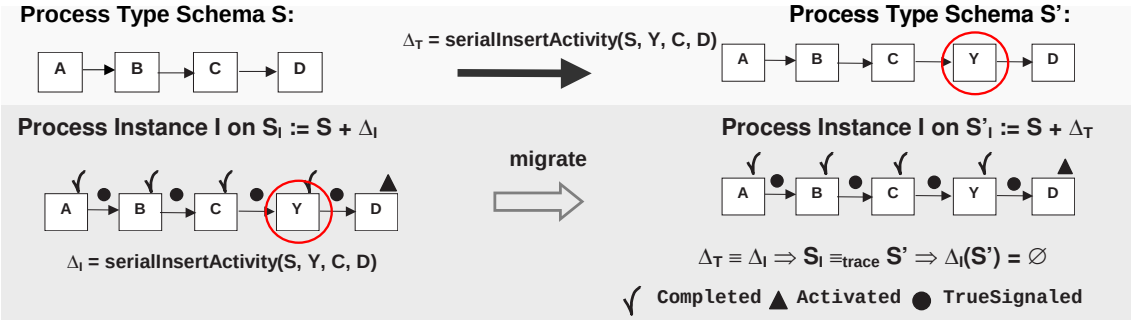


Figure 6.18: Migrating Instances with Equivalent Bias

Migration Strategy 2 (For Process Instances with Equivalent Bias) *Let S be a (correct) process type schema and let Δ_T be a process type change which transforms S into another (correct) type schema S'. Let further $I = (S, \Delta_I, \ldots)$ be a process instance on S with current instance execution schema $S_I := S + \Delta_I$. Finally, let Δ_T and Δ_I be equivalent changes, i.e., $\Delta_T \equiv \Delta_I$ holds. Then I can correctly migrate to S' (without propagating Δ_T to S_I) resulting in bias $\Delta_I = \emptyset$ on S', i.e., $I = (S', \emptyset, \ldots)$.*

Actually we do not propagate Δ_T to S_I since $S_I := S+\Delta_I \equiv_{trace} S+\Delta_T$ already reflects Δ_T. Consequently, no structural conflicts between Δ_T and Δ_I can occur. Interestingly, for instances with equivalent bias, migration is even possible if these instances have actually progressed too far and are therefore no longer compliant with S'. In the given case change Δ_T has been completely and precisely anticipated by the instance–specific change Δ_I. Note that Δ_I was introduced at a point in time the respective instance has been compliant with S'. Altogether, for migrating process instances with equivalent bias, state–related and structural compliance are always fulfilled (cf. Criterion 8).

Example 6.18 (Migrating Instances with Equivalent Bias): Consider Figure 6.18. Process type change Δ_T and process instance change Δ_I are equivalent since both have inserted activity Y into the same context. According to Migration Strategy 2, we can re–link I to S' without any further compliance checks. Note that this is possible though I is actually not state compliant with S'[10]. The resulting bias $\Delta_I(S')$ is nullified on S'.

Regarding subsumption equivalence between concurrent process type and process instance changes (on S) we have to distinguish two cases: (1) Δ_T is subsumption equivalent with Δ_I, i.e, Δ_I contains all effects of Δ_T on S and has additional effects (formally: $\Delta_T \prec \Delta_I$) or (2) Δ_I is subsumption equivalent with Δ_T, i.e., Δ_T includes the effects of Δ_I on S but has

[10]Note that there is a conflict between the runtime performance of the PMS and the goal of migrating as many instances as possible. One possibility to increase the migration rate is to first check state–related compliance by verifying the state–conditions set out in Section 4.3.2. This can be done very quickly. However, doing so, some instances with equivalent bias may be excluded from migration due to the fact that they are actually not compliant regarding their state (what is irrelevant for instances with equivalent bias).

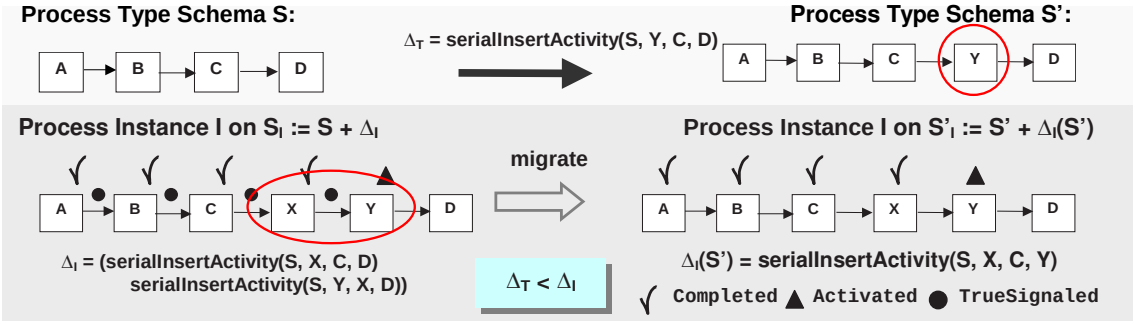


Figure 6.19: Migrating Instances with $\Delta_T \prec \Delta_I$

additional effects (formally: $\Delta_I \prec \Delta_T$). In particular, we need different migration strategies for these two cases.

Migration Strategy 3 (for Process Instances with $\Delta_T \prec \Delta_I$) *Let S be a (correct) process type schema and Δ_T be a process type change which transforms S into another (correct) type schema S'. Let further $I = (S, \Delta_I, \ldots)$ be a process instance on S with current instance execution schema $S_I := S + \Delta_I$ and with $\Delta_T \prec \Delta_I$ (i.e., Δ_T is subsumption equivalent with Δ_I). Then: I can correctly migrate to S' (without propagating Δ_T to S_I) with resulting bias $\Delta_I(S') = \Delta_I \backslash \Delta_T$[11] on S', i.e., $I = (S', \Delta_I(S'), \ldots)$.*

If $\Delta_T \prec \Delta_I$ holds the effects of Δ_T on S are already reflected in S_I. Therefore, Δ_T has not to be propagated to S_I. Consequently we do not have to check state–related or structural compliance (cf. Criterion 8) for this case. However, S_I deviates from S' since Δ_I has additional effects on S when compared to Δ_T. Therefore we have to store a remaining bias $\Delta_I(S')$ on S' after re–linking I from S to S'. Note that it is difficult to determine $\Delta_I(S')$ since it is not always sufficient just to calculate the difference between the set–based representations of Δ_T and Δ_I even if both changes are purged (cf. Section 6.5). More precisely, there may be context–dependent changes within Δ_T or Δ_I (cf. Section 6.2) which counterpart this straightforward approach. Instead we have to develop other methods to calculate $\Delta_I(S')$ (cf. Section 6.7).

Example 6.19 (Δ_T Sumsumption Equivalent With Δ_I): Regarding Figure 6.19 type change Δ_T is subsumption equivalent with instance change Δ_I. Therefore we re–link I to S' without further compliance checks. In contrast to instances with equivalent bias we have to store a resulting bias $\Delta_I(S')$ on S' containing all effects Δ_I "has more" than Δ_T (cf. Migration Strategy 3).

Migration Strategy 4 (for Process Instances with $\Delta_I \prec \Delta_T$) *Let S be a (correct) process type schema and Δ_T be a process type change which transforms S into another (correct) type*

[11] $\Delta_I \setminus \Delta_T$ denotes all effects of Δ_I resulting in S_I on S not present in S'. How to concretely calculate this bias will be described in Section 6.7.

schema S'. Let further $I = (S, \Delta_I, \ldots)$ be a process instance on S with current instance execution schema $S_I := S + \Delta_I$ and with $\Delta_I \prec \Delta_T$ (i.e., Δ_I is subsumption equivalent with Δ_T). Then: I can correctly migrate to S' with resulting bias $\Delta_I = \emptyset$ on S', i.e., $I = (S', \emptyset, \ldots) \Longleftrightarrow$

$\Delta_T \setminus \Delta_I$ can be correctly propagated to S_I regarding its state (cf. Criterion 7).

If $\Delta_I \prec \Delta_T$ holds the new process type schema $S' := S + \Delta_T$ completely reflects the effects of Δ_I on S. Furthermore, the effects which Δ_T has in addition to Δ_I are also contained in S'. Though, effectively, no parts of Δ_T are propagated to S_I (doing so would only result in S' again) logically we have to check state–related compliance on S_I for these parts of Δ_T which are not yet reflected by Δ_I; i.e., we must check the compliance conditions set out in Section 4.3.2 for $\Delta_T \setminus \Delta_I$. Determining $\Delta_T \setminus \Delta_I$ poses similar problems as determining the resulting bias $\Delta_I(S')$ in case $\Delta_T \prec \Delta_I$ holds (see above). Again, at this point, we refer to Section 6.7 where we present a method to build up a virtual change representing the difference between two other changes Δ_T and Δ_I. This is based on the respective purged change logs and on the difference sets.

If $\Delta_I \prec \Delta_T$ holds there can be no structural conflict between Δ_T and Δ_I, or more precisely, between $\Delta_T \setminus \Delta_I$ and Δ_I. If this had been the case Δ_T itself would already contain a structural conflict what is contradictory to our general assumptions.

Finally, the resulting bias $\Delta_I(S')$ on S' becomes the empty change if $\Delta_I \prec \Delta_T$ holds. For this case all effects of Δ_I on S are reflected by Δ_T.

Example 6.20 (Δ_I Subsumption Equivalent With Δ_T): Consider Figure 6.20 where type change Δ_T subsumes instance change Δ_I, i.e., $\Delta_I \prec \Delta_T$. In this case, we have to check state–related compliance for $\Delta_T \setminus \Delta_I$ (cf. Section 6.7). If I is compliant with S' it can be re–linked to S' and the resulting bias $\Delta_I(S')$ becomes empty (cf. Migration Strategy 4).

Process instances with partially equivalent bias cannot be uniformly treated since this class of instances may contain strongly varying instance–specific changes. These instance–specific changes may range from inserting (different) activities into the same target context at type and instance level to type and instance changes having almost the same effects on S. Furthermore, for some cases there is no automatic migration strategy as the following example shows:

Example 6.21 (Migrating Instances With Partially Equivalent Bias): Consider Figure 6.21 where type change Δ_T destroys the target context of instance change Δ_I. This is caused by deleting source activity C of the newly inserted sync edge (B, C). In this case, Δ_T cannot be applied to S_I (and therefore Δ_T and Δ_I are not commutative and consequently not disjoint). A possible measure here is to report the context–destroying conflict between Δ_T and Δ_I to the user and to ask him or her whether it is desired to migrate I to S' by specifying the remaining bias $\Delta_I(S')$ on S' or to let I finish according to S (cf. Section 6.7).

Consequently, it is not possible to provide a (default) migration strategy for partially equivalent process type and instance changes. However, it is possible to subclassify the set of partially equivalent changes (cf. Section 6.6.3), report the results to the user, and present decision rules

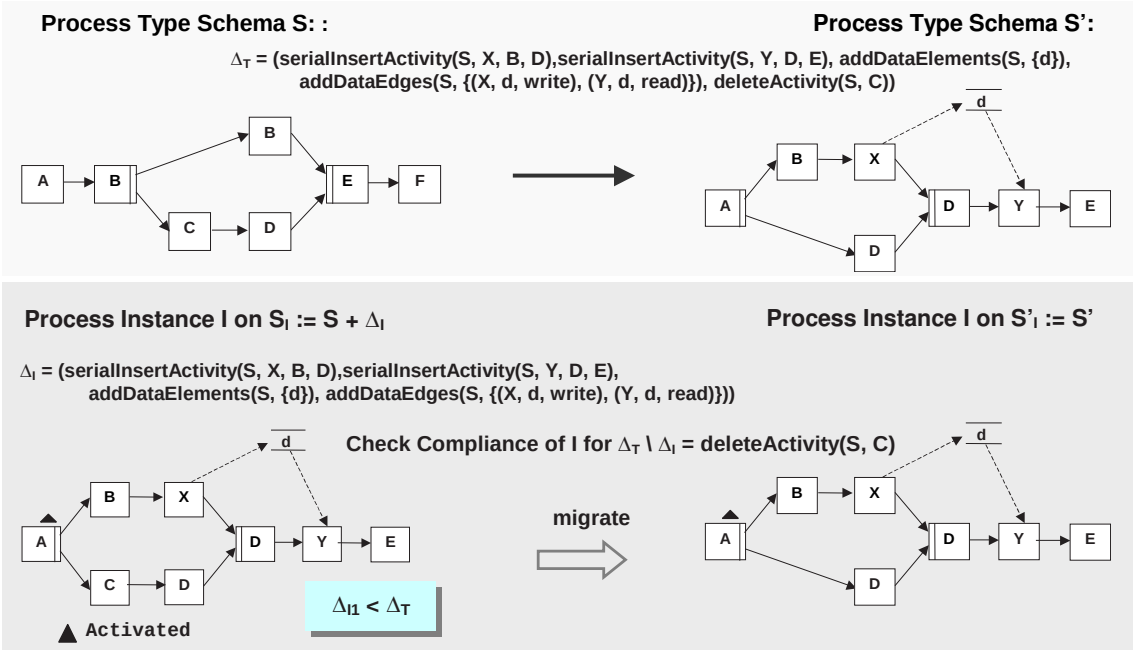


Figure 6.20: Migrating Instances with $\Delta_I \prec \Delta_T$

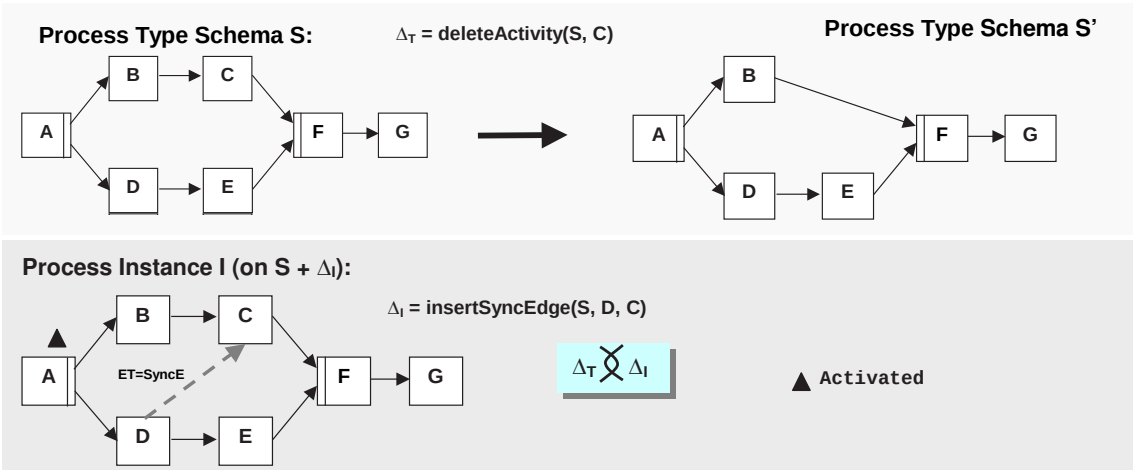


Figure 6.21: Migrating Instances with Partially Equivalent Changes

afterwards (cf. Section 6.7). Note that for partially equivalent changes state–related and structural compliance must be ensured since those parts of Δ_T and Δ_I which are different from each other may form a structural conflict within the resulting schema (cf. Criterion 8). Furthermore, the changes contained in $\Delta_T \setminus \Delta_I$ may cause state–related conflicts.

In summary, it is favorable if process instances have an equivalent, subsumption equivalent or disjoint bias (when compared with the process type change). For these cases we can offer automatic migration strategies. However, applying arbitrary change transactions consisting of different kinds of change operations with upmost probability we will also get process instances with partially equivalent changes (for which we then have to involve the user in the migration decision). This user interactions slow down the migration process. However, for many process instance changes which are detected as being partially equivalent this classification is too coarse. The reason is that they are partially equivalent regarding the changes as a whole. However, we will show in the next section that is sufficient to put focus on the different kinds of change operations, e.g., activity insertions when determining the degree of overlap.

6.6.3 On Optimizing Migration Strategies for Overlapping Process Changes

In the previous section we have provided an adequate migration strategy for each degree of overlap between process type and process instance changes. In this section we show how the application of these migration strategies can be optimized, i.e., how we can increase the number of process instances for which one of the automatic migration strategies can be applied.

The conditions for equivalent as well as for subsumption equivalent changes (cf. Definitions 11 and 17) are very precise. In contrast condition ψ of Definition 18 for partially equivalent changes is relatively coarse. As a consequence, usually there will be only few process instances being classified as having equivalent or subsumption equivalent bias. In contrast the set of instances with partially equivalent bias will contain a greater number of instances (with strongly varying bias) which may cause expensive user interactions (cf. Section 6.6.2)[12].

When looking at Figure 6.22 we see that changes Δ_1 and Δ_2 capture different kinds of change operations like, for example, insert, delete, and move operations. More precisely, Δ_1 and Δ_2 are *equivalent* regarding delete operations (both delete activity E) and Δ_2 is subsumption equivalent with Δ_1 regarding insert operations. Furthermore, Δ_2 is *subsumption equivalent* with Δ_1 regarding data flow operations whereas Δ_1 is *subsumption equivalent* with Δ_2 regarding move operations. Comparing Δ_1 and Δ_2 as a whole both changes would be assumed as being partially equivalent although they are (subsumption) equivalent regarding the different kinds of change operations.

Therefore we group the changes regarding the different kinds of applied change operations. Then solely the groups including the same kind of change operations are compared. Doing so enables a more fine–granular determination of the degree of overlap. Taking this into considera-

[12]Th standard case, however, is that process instances have a disjoint bias.

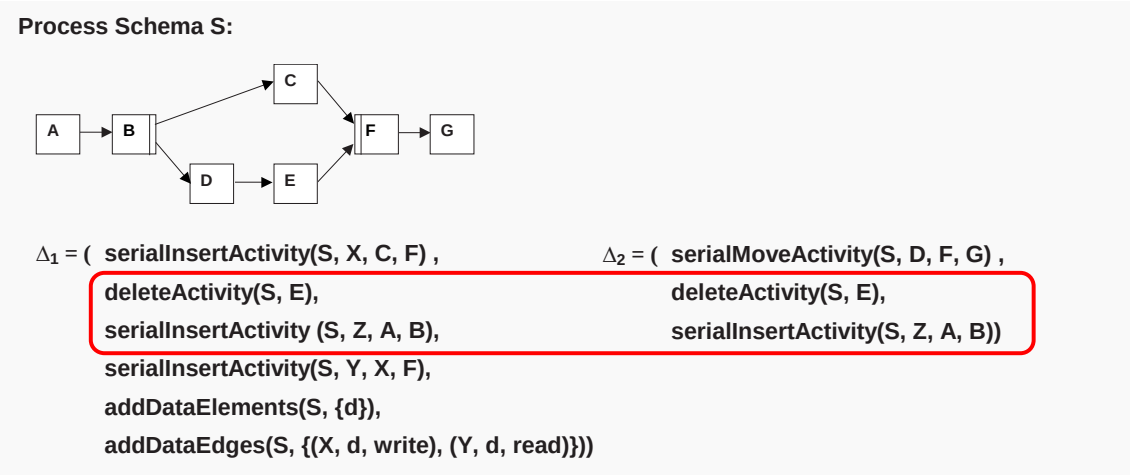


Figure 6.22: Partially Equivalent Changes

tion for the delete operation captured by Δ_1 and Δ_2 (cf. Figure 6.22) we can apply the default migration strategy for equivalent changes (cf. Theorem 2) for example. Furthermore, we can better assist the user and provide more precise problem reports to them. Thus the transparency of the overall migration process can be improved when comparing changes according to their *projections* on the different kinds of operations instead of comparing whole change transactions.

Auxiliary Definition 1 (Assigning Change Operations to Change Types) *Let Δ be a change. Let further*

1. *Change be a set of change operations (cf. Tables 3.2 and 3.3) with*

   $Change:=\{[serial|parallel|branch]InsertActivity(\ldots),\ deleteActivity(\ldots),$
   $[serial|parallel|branch]MoveActivitiy(\ldots),\ insertSyncEdge(\ldots),\ deleteSyncEdge(\ldots),$
   $insertLoopEdge(\ldots),\ deleteBlock(\ldots),\ addDataElements(\ldots),\ deleteDataElements(\ldots),$
   $addDataEdges(\ldots),\ deleteDataEdges(\ldots),\ changeActivityAttributes(\ldots),$
   $changeEdgeAttributes(\ldots)\}$ *and*

2. *OpType be a set of operation types with* $OpType:=\{ins\_Act,\ del\_Act,\ move\_Act,\ ins\_Sync,$ $del\_Sync,\ ins\_Loop,\ del\_Loop,\ data,\ attrChange\}$

3. *optype be a function which assigns to each change operation in Change its specific operation type in OpType. Formally:*

$$optype : Change \mapsto OpType$$

(The concrete assignment of change operations to operation types is given in Auxiliary Definition 2 (cf. Appendix B).)

Auxiliary Definition 1 assigns to each change operation a particular key word indicating its type, e.g., key word "ins_Act" is assigned to change operation $serialInsertActivity(\ldots)$. Based on this we can project changes onto the different kind of change types:

Definition 19 (Change Projections) *Let the assumption be as in Auxiliary Definition 1. Then* $\Delta[optype]$ *denotes the projection of* Δ *onto optype. Formally:*
$\Delta[optype] = (op_i, \ldots op_k)$ $(i \leq k \leq n)$ *with:* $\forall op_j (j = 1, \ldots, k)$*:* $optype(op_j) = optype$

The projections can be arbitrarily combined, i.e.,

$\Delta[optype_1 \oplus optype_2]$*:=* $(op_i, \ldots, op_k)$ $(i \leq k \leq n)$ *with:*
$\forall op_j$ $(j = 1, \ldots, k)$*:* $optype(op_j) \in \{optype_1, optype_2\}$

We illustrate the notion of change projections introduced above by providing the following example.

Example 6.22.a (Change Projections): Consider Figure 6.22. The projections for changes Δ_1 and Δ_2 turn out as follows:

- Projections for Δ_1:
  $\Delta_1[ins\_Act] = (serialInsertActivity(S, X, C, F), serialInsertActivity(S, Y, X, F),$
  $serialInsertActivity(S, Z, A, B))$;
  $\Delta_1[del\_Act] = (deleteActivity(S, E))$;
  $\Delta_1[data] = (addDataElements(S, \{d\}), addDataEdges(S, \{(X, d, write), (Y, d, read)\}))$
- Projections for Δ_2:
  $\Delta_2[ins\_Act] = (serialInsertActivity(S, Z, A, B))$;
  $\Delta_2[del\_Act] = (deleteActivity(S, E))$
  $\Delta_2[move\_Act] = (serialMoveActivity(S, D, F, G))$
- Combined Projection for Δ_2:
  $\Delta_2[ins\_Act \oplus move\_Act] = (serialInsertActivity(S, Z, A, B), serialMoveActivity(S, D, F, G))$

The combined projection on operations like inserting and moving activities, $\Delta_2[ins \oplus move\_Act]$ (cf. Auxiliary Definition 2, cf. Appendix B), refers to a possible kind of interaction between these two kinds of operations. As we have discussed in Section 6.5 it is important to consider the aggregated order between activities inserted or shifted to the same context. This plays also a role when determining degrees of overlap in the following (cf. Section 6.6.4).

When determining the particular degree of overlap between process changes we do not consider the changes as a whole but use the single change projections (cf. Definition 19). More

precisely, given two changes Δ_1 and Δ_2 concurrently applied to a process schema S we compare the particular projections of Δ_1 and Δ_2 but not the whole changes. Therewith, we achieve a fine–granular comparison based on which adequate messages for the user can be generated and appropriate migration strategies can be proposed.

Summary 4 (Degree of Overlap Between Change Projections) *Let S be a (correct) process schema and let Δ_i $(i = 1, 2)$ be two changes transforming S into (correct) process schemes $S_i, i = 1, 2$. Assume that $\Delta_i[op\_type]$ are the projections of Δ_i $(i = 1, 2)$ as defined in Definition 19. Then we determine the degree of overlap between Δ_1 and Δ_2 along the degrees of overlap between the projections of the same kind, i.e., we check whether*

$\Delta_1[op\_type] \equiv \Delta_2[op\_type]$ *or*
$\Delta_1[op\_type] \prec \Delta_2[op\_type]$ *(or vice versa) or*
$\Delta_1[op\_type] \between \Delta_2[op\_type]$ *or*
$\Delta_1[op\_type] \cap \Delta_2[op\_type] = \emptyset$

It is also possible to compare the combined projection on certain kinds of changes, e.g., $\Delta_1[ins\_Act \oplus move\_Act] \equiv \Delta_2[ins\_Act \oplus move\_Act]$ if, for example, we want to analyze the interactions between insert and move operations. The challenging question is how to (efficiently) check the different degrees of overlap between Δ_1 and Δ_2 or, more precisely, between the particular projections. In Section 6.3, we have introduced an approach for which trace equivalence between S_1 and S_2 is ensured by verifying that S_1 and S_2 are isomorphic. This method, in turn, is applicable to check equivalence or disjointness of changes Δ_1 and Δ_2 since the respective definitions are based on trace equivalence. However, the isomorphism approach is not adequate for our concerns for the following reasons:

1. It cannot be used to verify subsumption or partial equivalence since we cannot find adequate definitions based on graph isomorphims for these two change relations.
2. As already discussed in Section 6.3, materializing respective process schemes to be checked whether they are isomorphic or not may become costly, especially for a large number of running biased instances.
3. If we want to restrict comparisons between Δ_1 and Δ_2 to comparisons between their particular projections (in order to get the right level of granularity) the isomorphism approach is practically not applicable. For example, if we would like to verify isomorphism for $S + \Delta_1[ins\_Act]$ and $S + \Delta_2[ins\_Act]$ it would become necessary to determine the process schemes resulting from the application of the insert operations (contained within the respective projections). This "projection" on process schemes resulting for the application of change projections would have to be done for each kind of change projection. However, doing so is by far too expensive, especially when regarding a multitude of running instances.

For these reasons we want to find more advanced and efficient methods for detecting the particular degree of overlap between the projections of two changes Δ_1 and Δ_2.

6.6.4 Determining the Degree of Overlap Based on Projections

We can use our hybrid approach (cf. Section 6.5) for detecting the degree of overlap between change projections. As an exampke take changes Δ_1 and Δ_2 as depicted in Figure 6.22. A comparison between activity sets $N^{add}_{\Delta_1} = \{X, Y, Z\}$ and $N^{add}_{\Delta_2} = \{Z\}$, between anchor sets $AnchorIns(S, \Delta_1) = \{(C, X, F), (C, Y, F), (A, Z, B)\}$ and $AnchorIns(S, \Delta_2) = \{(A, Z, B)\}$, and between order stes $OrderIns(S, \Delta_1) = \{(X, Y)\}$ and $OrderIns(S, \Delta_2) = \emptyset$ (cf. Section 6.5) exactly reflects the comparison between the projections on insert operations of Δ_1 and Δ_2. As result of this comparison we obtain that $\Delta_1[ins\_Act] \succ \Delta_2[ins\_Act]$ holds.

The hybrid approach is always applicable for determing the degree of overlap between change projections, i.e., based on this method subsumption and partial equivalence can be detected.

To formally underpin our above considerations we provide Theorems 11 and 12. They state that equivalence and disjointness of two changes Δ_1 and Δ_2 can be verified by applying the hybrid approach to the particular change projections.

Theorem 11 (Equivalent Changes) *Let S be a (correct) process schema and let $\Delta_i, i = 1, 2$ be two changes which transform S into (correct) process schemes $S_i, i = 1, 2$. Then Δ_1 and Δ_2 are equivalent, i.e, $\Delta_1 \equiv \Delta_2$ if the following conditions (1) – (4) hold:*

(1) **Change Operations on Activity Sets:**

(a) $\Delta_1[ins\_Act] \equiv \Delta_2[ins\_Act] \Longleftrightarrow$
$(N^{add}_{\Delta_1} = N^{add}_{\Delta_2} \wedge$
$AnchorIns(S, \Delta_1) = AnchorIns(S, \Delta_2) \wedge$
$OrderIns(S, \Delta_1) = OrderIns(S, \Delta_2))$

(b) $\Delta_1[del\_Act] \equiv \Delta_2[del\_Act] \Longleftrightarrow N^{del}_{\Delta_1} = N^{del}_{\Delta_2}$

(c) $\Delta_1[move\_Act] \equiv \Delta_2[move\_Act] \Longleftrightarrow$
$(N^{move}_{\Delta_1} = N^{move}_{\Delta_2}) \wedge$
$AnchorMove(S, \Delta_1) = AnchorMove(S, \Delta_2)) \wedge$
$OrderMove(S, \Delta_1) = OrderMove(S, \Delta_2))$

(d) $\Delta_1[ins/move\_Act] \equiv \Delta_2[ins/move\_Act] \Longleftrightarrow$
$(N^{move}_{\Delta_1} = N^{move}_{\Delta_2}) \wedge$
$\Delta_1[ins\_Act] \equiv \Delta_2[ins\_Act] \wedge$
$\Delta_1[move\_Act] \equiv \Delta_2[move\_Act] \wedge$
$OrderAgg(S, \Delta_1) = OrderAgg(S, \Delta_2))$

(2) **Change Operations on Sync and Loop Edges:**

(a) $\Delta_1[ins\_Sync] \equiv \Delta_2[ins\_Sync] \Longleftrightarrow SyncE^{add}_{\Delta_1} = SyncE^{add}_{\Delta_2}$

(b) $\Delta_1[del\_Sync] \equiv \Delta_2[del\_Sync] \Longleftrightarrow SyncE^{del}_{\Delta_1} = SyncE^{del}_{\Delta_2}$

(c) $\Delta_1[ins\_Loop] \equiv \Delta_2[ins\_Loop] \Longleftrightarrow LoopE^{add}_{\Delta_1} = LoopE^{add}_{\Delta_2}$

(d) $\Delta_1[del\_Loop] \equiv \Delta_2[del\_Loop] \Longleftrightarrow LoopE^{del}_{\Delta_1} = LoopE^{del}_{\Delta_2}$

(3) **Change Operations on Data Flow:**
$\Delta_1[data] \equiv \Delta_2[data] \Longleftrightarrow$
$(D^{add}_{\Delta_1} = D^{add}_{\Delta_2} \wedge D^{del}_{\Delta_1} = D^{del}_{\Delta_2}) \wedge$
$(DataE^{add}_{\Delta_1} = DataE^{add}_{\Delta_2} \wedge DataE^{add}_{\Delta_1} = DataE^{add}_{\Delta_2})$

(4) **Change Operations on Attributes:**
$\Delta_1[attrChange] \equiv \Delta_2[attrChange] \Longleftrightarrow ChangedAttr_{\Delta_1} = ChangedAttr_{\Delta_2}$

(Φ)

A formal proof of Theorem 11 can be found in Appendix C (cf. Proof C.10). Interestingly, we can proof Theorem 11 by showing that "$(\Phi) \Longrightarrow S_1 \simeq S_2 \overset{Theorem10}{\Longrightarrow} S_1 \equiv_{trace} S_2$" holds.

We pick up the scenario depicted in Figure 6.22 to illustrate the above theorem.

Example 6.22.b (Change Projections): Consider Figure 6.22. Applying the hybrid approach to the projections of Δ_1 and Δ_2 on delete operations (i.e., $\Delta_1[del\_Act]$ and $\Delta_2[del\_Act]$ yields:

$$N^{del}_{\Delta_1} = N^{del}_{\Delta_2} \Longrightarrow \Delta_1[del\_Act] \equiv \Delta_2[del\_Act]$$

To be able to define disjointness of two changes Δ_1 and Δ_2 in a similar way we first have to introduce notions for operations inserting or moving activities into the same context (cf. Definition 20) and for operations destroying the context of other operations (cf. Definition 21).

Definition 20 (Changes with Conflicting Target Context) *Let S be a (corrrect) process schema and let $\Delta_i, i = 1, 2$ be two changes transforming S into (correct) process schemes S_1 and S_2. Let further $N^{add}_{\Delta_i}$ and $N^{move}_{\Delta_i}$ be the activity sets newly inserted or moved by Δ_i and let $AnchorIns(S, \Delta_i)$ and $AnchorMove(S, \Delta_i)$ be the respective anchor sets ($i = 1, 2$). Then we denote Δ_1 and Δ_2 as operations with conflicting target context (notation: $\Delta_1 \curlyvee \Delta_2$) if Δ_1 contains an insert or move operations which inserts or moves an activity into the same context an insert or move operation of Δ_2 does. Formally:*

$\Delta_1 \curlyvee \Delta_2 \Longleftrightarrow$
$(E^{conc\_context}_{\Delta_1}[ins] \cup E^{conc\_context}_{\Delta_1}[move]) \cap (E^{conc\_context}_{\Delta_1}[ins] \cup E^{conc\_context}_{\Delta_2}[move]) \neq \emptyset$

with

- $E^{conc\_context}_{\Delta_1}[ins] = \{(n_1, n_2) | \exists (n_1, X, n_2) \in AnchorIns(S, \Delta_1) \wedge X \in N^{add}_{\Delta_1} \setminus N^{add}_{\Delta_2}\}$
- $E^{conc\_context}_{\Delta_2}[ins] = \{(n_1, n_2) | \exists (n_1, X, n_2) \in AnchorIns(S, \Delta_2) \wedge X \in N^{add}_{\Delta_2} \setminus N^{add}_{\Delta_1}\}$

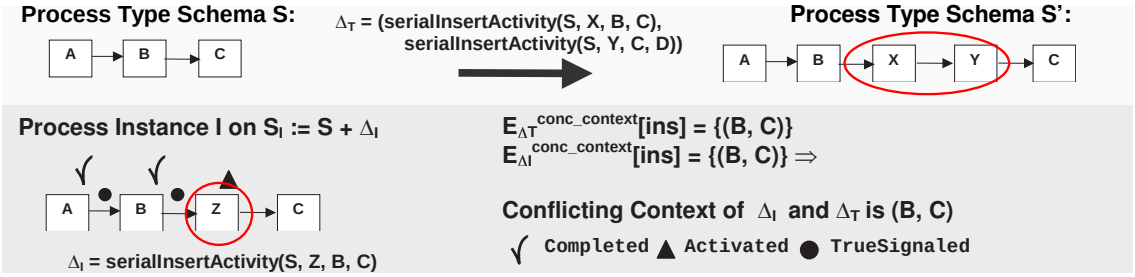


Figure 6.23: Changes with Conflicting Context

- $E_{\Delta_1}^{conc\_context}[move] = \{(n_1, n_2) | \exists (n_1, X, n_2) \in AnchorMove(S, \Delta_1) \wedge X \in N_{\Delta_1}^{move} \setminus N_{\Delta_2}^{move}\}$
- $E_{\Delta_2}^{conc\_context}[move] = \{(n_1, n_2) | \exists (n_1, X, n_2) \in AnchorMove(S, \Delta_2) \wedge X \in N_{\Delta_2}^{move} \setminus N_{\Delta_1}^{move}\}$

Regarding Definition 20, at first, we determine the context of each insert and move operation included in change Δ_1 but not in change Δ_2 (and vice versa). Then we check whether Δ_1 and Δ_2 use the same context or not. As the following example shows changes with conflicting context are also correctly determined for context–dependent changes when applying Definition 20.

Example 6.23 (Changes With Conflicting Context): Look at Figure 6.23 where type change Δ_T inserts activities X and Y between anchors B and C and instance change Δ_I inserts activity Z also between B and C. Determining the conflicting context of both changes we see that (B, C) lies in the intersect of them. Therefore (B, C) is a conflicting context used by changes Δ_T and Δ_I.

As we can see from Definition 20 to check whether $\Delta_1 \curlyvee \Delta_2$ holds we can use the anchor sets for insert and move operations. We extract those anchors which are affected by both changes Δ_1 and Δ_2. Then we pick all those anchors to which Δ_1 and Δ_2 have inserted or shifted different activities.

Definition 21 (Context–Destroying Changes) *Let S be a (corrrect) process schema and let $\Delta_1 = (op_1^1, \ldots, op_n^1)$ and $\Delta_2 = (op_1^2, \ldots, op_m^2)$ be two changes transforming S into (correct) process schemes S_1 and S_2. Then we denote Δ_1 and Δ_2 as context–destroying changes (notation: $\Delta_1 \curlyvee \Delta_2$) if Δ_1 contains a change operation which destroys the context of a change operation contained in Δ_2 or vice versa. This is the case if one of the following conditions holds ($i = 1, \ldots, n$):*

- $\exists op_i^1 = [serial|parallel|branch]MoveActivity(S, X^1, \ldots)$,
  $\exists op_j^2 \in \{serial[Insert|Move]Activity(S, X^2, src^2, dest^2),\ insertSyncEdge(S, src^2, dest^2)\}$
  with $X^1 = src^2 \vee X^1 = dest^2$ *or vice versa ("moving context away")*
- $\exists op_i^1 = deleteActivity(S, X^1)$,
  $\exists op_j^2 \in \{serial[Insert|Move]Activity(S, X^2, src^2, dest^2),\ insertSyncEdge(S, src^2, dest^2)\}$ *with*
  $X^1 = src^2 \vee X^1 = dest^2$ *or vice versa ("deleting context")*

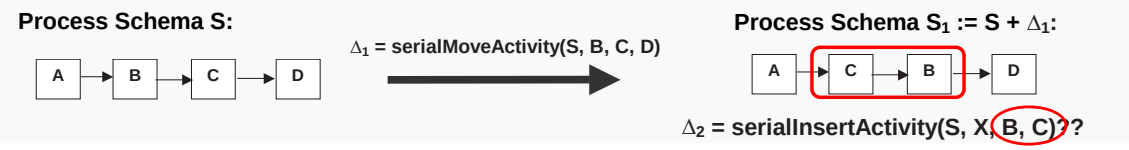


Figure 6.24: Context–Destroying Changes

- $\exists op_i^1 = deleteActivity(S, X^1)$,
  $\exists op_j^2 = changeActivityAttribute(S, X^2, attr^2, nV^2)$
  with $X^1 = X^2$ *or vice versa ("overriding activity attribute change")*
- $\exists op_i^1 = deleteActivity(S, X^1)$,
  $\exists op_j^2 = changeEdgeAttribute(S, (src^2, dest^2), attr^2, nV^2)$
  with $X^1 = src^2 \vee X^1 = dest^2$ *or vice versa ("overriding edge attribute change")*

Definition 21 summarizes the different scenarios in which a change operation destroys the context of another change operation: Within the first two scenarios, a change operation moves or deletes an activity which is used as insertion or move context by another change operation. The next scenario comprises change operations which delete an activity for which an activity attribute is changed by another change operation. Within the last scenario an activity is deleted by a change operation which is source or destination of an edge for which an edge attribute is changed by another change operation. An example for case "moving context away" (cf. Definition 21) is illustrated by Figure 6.24.

Example 6.24 (Context–Destroying Changes): Consider Figure 6.24. Change Δ_1 destroys the context of change Δ_2 since it moves one of the anchors of Δ_2 away. As a consequence, Δ_2 cannot be applied to S_1.

Context–destroying changes constitute a very hard problem. At their presence it is not possible to provide automatic migration strategies (illustrated by Example 6.21). However, we can precisely report the "weak points" in conjunction with context–destroying changes to users. In doing so, we can adequately assist users (cf. Section 6.7).

Preserving Definitions 20 and 21 is important for the disjointness of changes Δ_1 and Δ_2 on S in order to guarantee their commutativity. Otherwise, if, for example, Δ_2 destroys the context of a change operation of Δ_1 process schema $(S + \Delta_1) + \Delta_2$ can be produced but it is not possible to apply Δ_1 to $S_2 := S + \Delta_2$. Thus Δ_1 and Δ_2 are not commutative.

Theorem 12 (Disjoint Changes) *Let S be a (correct) process schema and let $\Delta_i, i = 1, 2$ be two changes which transform S into (correct) process schemes $S_i, i = 1, 2$. Then Δ_1 and Δ_2 are disjoint (i.e, $\Delta_1 \cap \Delta_2 = \emptyset$) if the following conditions hold:*

1. **Change Operations on Activity Sets:**

(a) $\Delta_1[ins\_Act] \cap \Delta_2[ins\_Act] = \emptyset \iff$
$(N^{add}_{\Delta_1} \cap N^{add}_{\Delta_2} = \emptyset \wedge \neg(\Delta_1[ins\_Act] \curlyvee \Delta_2[ins\_Act])$

(b) $\Delta_1[del\_Act] \cap \Delta_2[del\_Act] = \emptyset \iff N^{del}_{\Delta_1} \cap N^{del}_{\Delta_2} = \emptyset$

(c) $\Delta_1[move\_Act] \cap \Delta_2[move\_Act] = \emptyset \iff$
$(N^{move}_{\Delta_1} \cap N^{move}_{\Delta_2} = \emptyset \wedge \neg(\Delta_1[move\_Act] \curlyvee \Delta_2[move\_Act])$

(d) $\Delta_1[ins/move\_Act] \cap \Delta_2[ins/move\_Act] = \emptyset \iff$
$(\Delta_1[ins\_Act] \cap \Delta_2[ins\_Act] = \emptyset \wedge$
$\Delta_1[move\_Act] \cap \Delta_2[move\_Act] = \emptyset \wedge$
$\neg(\Delta_1[ins/move\_Act] \curlyvee \Delta_2[ins/move\_Act])$

2. **Change Operations on Sync and Loop Edges:**

(a) $\Delta_1[ins\_Sync] \cap \Delta_2[ins\_Sync] = \emptyset \iff SyncE^{add}_{\Delta_1} \cap SyncE^{add}_{\Delta_2} = \emptyset$

(b) $\Delta_1[del\_Sync] \cap \Delta_2[del\_Sync] = \emptyset \iff SyncE^{del}_{\Delta_1} \cap SyncE^{del}_{\Delta_2} = \emptyset$

(c) $\Delta_1[ins\_Loop] \cap \Delta_2[ins\_Loop] = \emptyset \iff LoopE^{add}_{\Delta_1} \cap LoopE^{add}_{\Delta_2} = \emptyset$

(d) $\Delta_1[del\_Loop] \cap \Delta_2[del\_Loop] = \emptyset \iff LoopE^{del}_{\Delta_1} \cap LoopE^{del}_{\Delta_2} = \emptyset$

3. **Change Operations on Data Flow:**
$\Delta_1[data] \cap \Delta_2[data] = \emptyset \iff$
$(D^{add}_{\Delta_1} \cap D^{add}_{\Delta_2} = \emptyset \wedge D^{del}_{\Delta_1} \cap D^{del}_{\Delta_2} = \emptyset) \wedge$
$(DataE^{add}_{\Delta_1} \cap DataE^{add}_{\Delta_2} = \emptyset \wedge DataE^{add}_{\Delta_1} \cap DataE^{add}_{\Delta_2} = \emptyset)$

4. **Change Operations on Attributes:**
$\Delta_1[attrChange] \cap \Delta_2[attrChange] = \emptyset \iff ChangedAttr_{\Delta_1} \cap ChangedAttr_{\Delta_2} = \emptyset$

5. **Context–Destroying Change Operations:**
$\neg(\Delta_1 \curlyvee \Delta_2)$

(Θ)

We abstain from a formal proof of Theorem 12 and give a short proof sketch instead: Basically, we have to show that

$$\Theta \Longrightarrow (S_1 + \Delta_2 \equiv_{trace} S_2 + \Delta_1 \wedge N^{add}_{\Delta_1} \cap N^{add}_{\Delta_2} = \emptyset$$

holds (cf. Definition 10). Trivially, the second term of this conjunction is fulfilled by (Θ). The first term about commutativity of Δ_1 and Δ_2 can be similarly shown as trace equivalence within Proof C.10, i.e., by verifying isomorphism between $S_1 + \Delta_2$ and $S_2 + \Delta_1$ (formally: $(\Theta) \Longrightarrow S_1 + \Delta_2 \simeq S_2 + \Delta_1$). At this, we can argue by building the sets of activities, control edges, etc. for $S_1 + \Delta_2$ and $S_2 + \Delta_1$ and by showing that they are equal. For example, let $N_{(1,2)}$ be the activity set of $S_1 + \Delta_2$ (and $N_{(2,1)}$ the activity set of $S_2 + \Delta_1$ respectively). $N_{(1,2)}$ has been build by $[((N + N^{add}_{\Delta_1}) \setminus N^{del}_{\Delta_1}) + N^{add}_{\Delta_2}] \setminus N^{del}_{\Delta_2}$. Since all activity sets are disjoint regarding (Θ) for Δ_1 and Δ_2 we can see that the same term can be build for $N_{(2,1)}$ as the activity set of $S_2 + \Delta_1$. Consequently, $N_{(1,2)} = N_{(2,1)}$ holds.

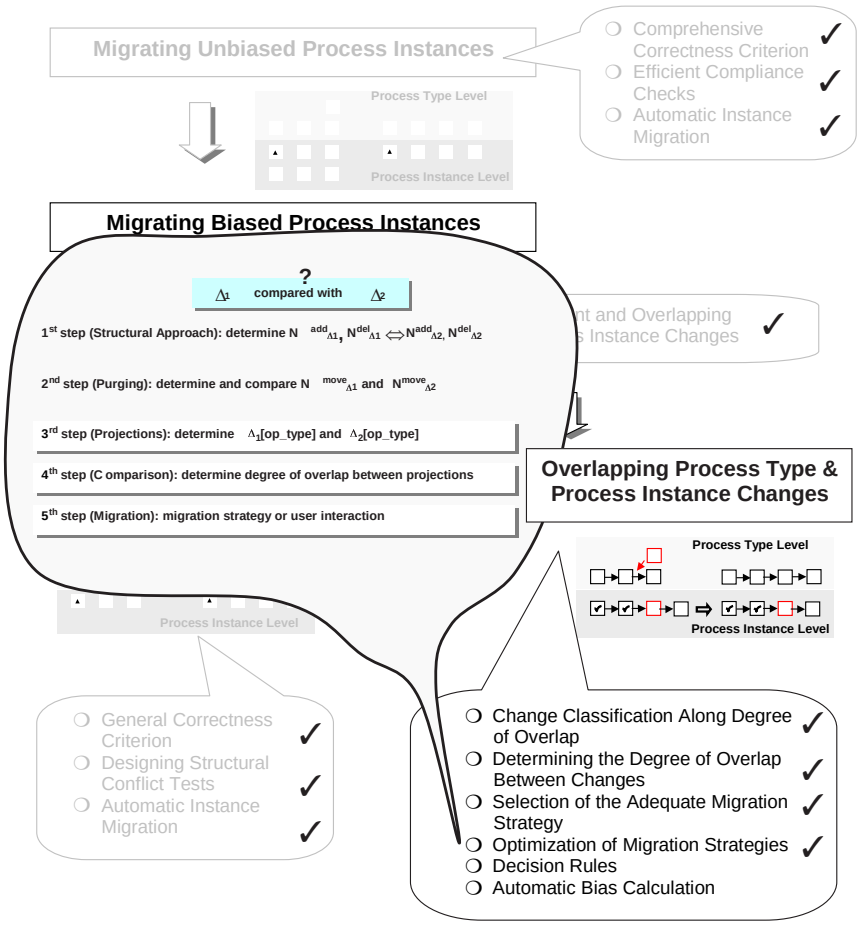


Figure 6.25: Applying Optimized Migration Strategies

Now it becomes clear how to check the degree of overlap between concurrent changes Δ_1 and Δ_2 on process schema S (cf. Figure 6.25). First, we determine the consolidated activity sets (cf. Definition 14), the anchor sets (cf. Algorithms 4 + 5), and the order sets (cf. Algorithms 6 + 7 and Definition 16). Based on this information we determine the degree of overlap between the different projections of Δ_1 and Δ_2 (cf. Definition 19). Finally, we present the migration strategies as summarized in Theorems 2, 3, and 4 to users. If Δ_1 and Δ_2 are partially equivalent (i.e., $\Delta_1 \between \Delta_2$) we provide rules to support the users' decision process. The different possibilities for partial equivalence between Δ_1 and Δ_2 and the resulting decision rules are described in Section 6.7.

To illustrate the theoretical results of this section we provide the following example by applying them to a concrete scenario of concurrent process type and process instance changes.

Example 6.26 (Detecting Degree of Overlap): Consider Figure 6.26 where type change Δ_T

serially inserts two activities X and Y and moves activity B from its current position to position between activities Y and D. To detect the degree of overlap between Δ_T and change Δ_{I_1} on I_1 we determine the difference sets from which we can see that $N^{add}_{\Delta_T} \subset N^{add}_{\Delta_{I_1}}$ and $N^{move}_{\Delta_T} \equiv N^{move}_{\Delta_{I_1}}$ hold (all other difference sets are empty for Δ_T as well as for Δ_{I_1}). Activities X and Y have been inserted and activity B has been moved between the same anchors C and D for Δ_T and Δ_{I_1}. Furthermore, the order sets of Δ_T are subsets of the order sets of Δ_{I_1}. Finally, sets $E^{conc\_context}_{\Delta_T}[ins|move]$ and $E^{conc\_context}_{\Delta_{I_1}}[ins|move]$ are empty what implies that Δ_T and Δ_{I_1} do not insert or move any activity into the same (conflicting) target context. Furthermore, these two changes are not mutually context–destroying. Altogether, we obtain that Δ_T is subsumption equivalent with Δ_{I_1}, i.e., $\Delta_T \prec \Delta_{I_1}$. Using Theorem 3, we migrate I_1 to S'. Doing so we have to calculate new bias $\Delta_{I_1}(S')$ on S'. How this calculation can be carried out is one of the challenges to be discussed in Section 6.7.

Considering instance I_2, we find out that Δ_T actually subsumes Δ_{I_2} regarding insert operations (i.e., $\Delta_{I_2}[ins\_Act] \prec \Delta_T[ins\_Act]$), but Δ_{I_2} subsumes Δ_T regarding move operations (i.e., $\Delta_T[move] \prec \Delta_{I_2}[move]$). Consequently, Δ_T and Δ_{I_2} are related to each other under a special form of partial equivalence, i.e., $\Delta_T \between \Delta_{I_2}$. Figure 6.26 depicts two possible instance–specific schemes for I_2 after its migration to S' whereby the second alternative is automatically determined by applying Algorithm 10 (cf. Section 6.7) and reported to users. Finally, instance Δ_{I_3} is partially equivalent with Δ_T. However, Δ_{I_3} "lies very close" to Δ_T since we insert and move exactly the same activities into the same target context but (only) in different order. This, in turn, is exactly reported to the user to assist him or her in an adequate way.

6.7 Decision Rules and Calculating Bias

In this section, we present decision rules for partially equivalent changes – remember that for this kind of changes in most cases no automatic migration is possible (cf. Section 6.7.1). In Section 6.7.2, we present algorithms which automatically calculate the new bias $\Delta_I(S')$ when migrating instance I to S'.

6.7.1 Decision Rules

Let S be a (correct) process schema and let $I = (S, \Delta_I, \ldots)$ be a biased process instance running on S. Let further Δ_T be a process type change which transforms S into another (correct) process schema S'. Assume that Δ_T and Δ_I are partially equivalent, i.e., $\Delta_T \between \Delta_I$ holds.

The (default) migration strategies for (subsumption) equivalent and disjoint process type and process instance changes have been already presented in Section 6.6.2. As mentioned, for partially equivalent changes there is no common migration strategy. However, partially equivalent changes vary from case to case as Example 6.26 has shown. Based on the presented concepts of change projection (cf. Definition 19) we can distinguish between different kinds of

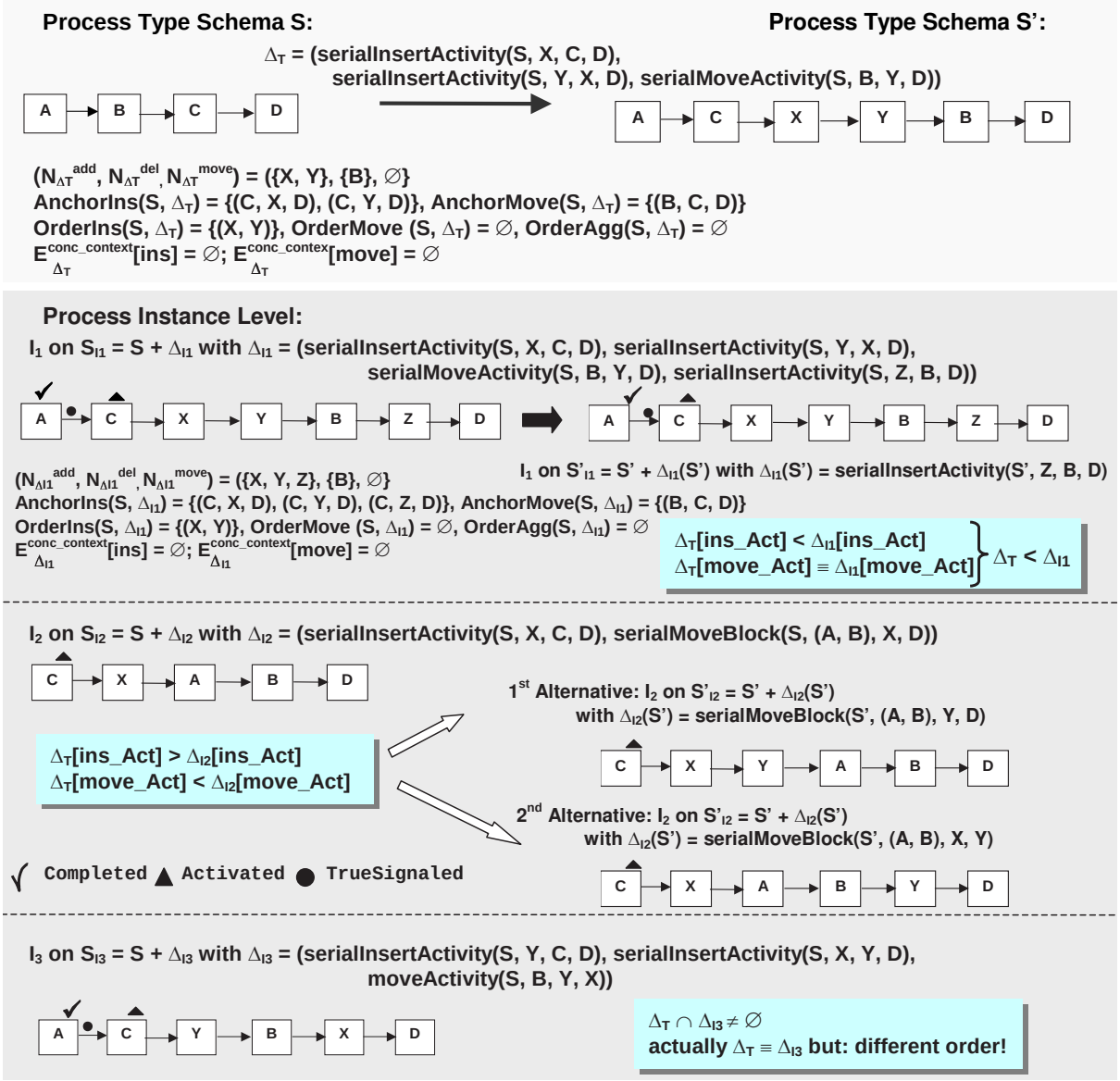


Figure 6.26: Determining Degree of Overlap Between Concurrent Changes

partially equivalent changes. Based on the respective kind of change we can define rules to be reported to users in order to provide optimal assistance. Based on these decision rules the user has two possibilities:

1. Instance I is excluded from the migration to the changed process type schema S' and therefore remains running according to the original schema S.
2. Instance I shall be migrated to the changed process type schema S'. Then I is re–linked to S' and the resulting instance–specific bias $\Delta_I(S')$ has to be determined. For this, there are two possibilities: In some special cases, the system is able to suggest a bias $\Delta_I(S')$ (cf. Algorithm 10) which can be chosen by the user. However, in most cases, the user has to individually specify $\Delta_I(S')$.

The first strategy of excluding instances from migrating to the changed process type schema can be chosen at any time by the user. For the second variant we present a sophisticated classification of instances with partially equivalent bias based on which decision rules for users can be deposited within the system (cf. Tables 6.1 and 6.2).

1) Different Order (cf. Table 6.1): This group contains all partially equivalent changes Δ_T and Δ_I which insert or move the same (or a subset of) activities into or to the same target context but in different order. This can be exactly reported to users such that they are able to decide on two possibilities: Instance change Δ_I is either estimated as being (subsumption) equivalent with type change Δ_T or the different order defined by Δ_I compared to Δ_T is explicitly desired. If the latter is the case users can decide to migrate I to the changed type schema S' by specifying instance–specific bias $\Delta_I(S')$ based on S'. Alternatively, they may let I further run according to the original type schema S.

2) Different Anchors (cf. Table 6.1): The "distance" between changes Δ_T and Δ_I belonging to this group is greater than for changes within the first group. Users might be indifferent regarding the order of newly to insert or to move activities but they should be sure regarding the context (anchors) where these activities are added or moved to. Therefore user decisions are based on the report where same activities (or a subset of them) have been inserted or moved between different anchors by Δ_T and Δ_I. Then users can decide either to finish respective instances according to original schema S or to migrate them to S' by specifying instance–specific change $\Delta_I(S')$.

3) Conflicting Target Context (cf. Table 6.1): A conflicting target context (cf. Definition 20) characterizes only a minor overlap between respective type and instance changes Δ_T and Δ_I. However, this case must be considered when calculating instance–specific change $\Delta_I(S')$ since $\Delta_I(S)$ uses one or more target positions for inserting or moving activities which are already occupied by activities inserted or moved by Δ_T. Therefore, we first detect the conflicting target context for Δ_T and Δ_I, report them to users, and adapt $\Delta_I(S')$ respectively (cf. Algorithm 10).

4) Multiple Operations (cf. Table 6.2): This group contains all concurrently applied changes Δ_T and Δ_I which access same activities. A special case herewith is the multiple insertion of

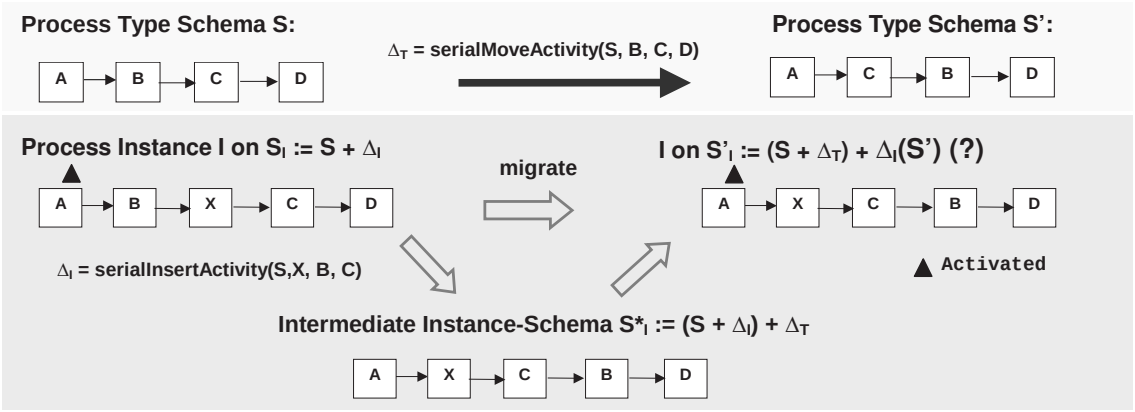


Figure 6.27: Context–Destroying Type Change

activities since this may lead to an instance–specific schema S'_I containing activities twice. This, however, may not correspond to users' intentions wherefore the multiple insertion is reported to them. For concurrent changes Δ_T and Δ_I which delete same activities we can re–link the affected instance I to changed type schema S' and make a suggestion regarding instance–specific change $\Delta_I(S')$ (cf. Algorithm 10, Section 6.7).

5) Context–Destroying Operations (cf. Table 6.2): Context–destroying operations Δ_T and Δ_I (cf. Definition 21) are similar to operations which have a concurrent target context (cf. Definition 20). They are not overlapping in the sense that same activities, sync edges or data elements are manipulated. However there may be problems with specifying $\Delta_I(S')$ on S'. These problems may be serious such that it is not possible to find automatic adaptation methods for $\Delta_I(S')$. If Δ_T destroys the context of Δ_I the intermediate schema $S^*_I := (S + \Delta_I) + \Delta_T$ can be determined. However for the instance–specific schema resulting from the re–linkage of I to S', i.e., $S'_I := (S + \Delta_T) + \Delta_I(S')$ (cf. Figure 6.27) this is not the case. Obviously, Δ_I cannot be maintained on S' since Δ_T has destroyed its context. Consequently, we can only report the respective destroyment of context to users and let them specify $\Delta_I(S')$.

Example 6.27 (Context–Destroying Type Change): Consider Figure 6.27 where type change Δ_T moves activity B from its current position to the position between activities C and D. Instance–specific change Δ_I, in contrast, inserts activity X between activities B and C. Obviously, Δ_T destroys the context of Δ_I ("moving context away", cf. Definition 21). The propagation of Δ_T to I (resulting in intermediate instance–specific schema $S^*_I := (S + \Delta_I) + \Delta_T$) is no problem since Δ_I does not destroy the context of Δ_T. Consequently, in general, the migration of I to S' is possible. The difficult question is how to transfer I from (logically) running according to S^*_I to run according to $S'_I := (S + \Delta_T) + \mathbf{\Delta_I}(\mathbf{S'})$, i.e., how to determine the new bias $\Delta_I(S')$. The simple example depicted in Figure 6.27 shows that this cannot be done automatically.

6) Data Flow Changes (cf. Table 6.2): It is a very sensitive job to adequately deal with the case

that $\Delta_T[data]$ and $\Delta_I[data]$ are partially equivalent. Note that it is difficult to automatically decide about the equality of data elements. Consequently, the user should be asked about the migration of Δ_I to S' and the resulting instance–specific change $\Delta_I(S')$ based on S'.

If it is possible to offer a decision rule which can be automatically executed and is accepted by users we still have to care about possible state–related and structural conflicts. The reason is that the parts of Δ_T which are not contained in Δ_I may cause state–related inconsistencies for I on S'. Furthermore, there may be also structural inconcistencies with S'_I resulting from the concurrent application of $\Delta_T \setminus \Delta_I$ and $\Delta_I \setminus \Delta_T$ on S. Consequently, it is no sufficient to determine $\Delta_I(S') = \Delta_I \setminus \Delta_T$ but we also have to calculate $\Delta_T \setminus \Delta_I$. For partially equivalent process type and process instance changes the calculation of $\Delta_T \setminus \Delta_I$ as well as the calculation of $\Delta_I \setminus \Delta_T$ is not possible (cf. Example 6.27).

Summarizing the results presented in Tables 6.1 and 6.2 and the respective explanations, we can conclude that in case of partially equivalent process type and process instance changes, users should be consulted in most cases. It is not possible to automatically determine $\Delta_I(S')$ on S' after re–linking I to S'. However, for concurrent process type and process instance changes using a conflicting target context and for projections on deleting activities and sync edges we can automatically generate a suggestion for $\Delta_I(S')$, i.e., for these special cases of partially equivalent changes Δ_T and Δ_I the new bias of I on S can be calculated by applying the same algorithm as for determining $\Delta_I(S')$ in case $\Delta_T \prec \Delta_I$ holds. In this case, we can re–link I to S' but we have to store remaining bias $\Delta_I(S')$ for I on S'. A last interesting application of a respective calculation method for change differences is the determination of $\Delta_T(S_I) := \Delta_T \setminus \Delta_I$ if $\Delta_I \prec \Delta_T$ holds. According to Migration Strategy 4 we have to check state–related compliance for I regarding $\Delta_T(S_I)$. Section 6.7.2 presents the respective algorithms and illustrates them by means of examples.

6.7.2 Calculating Bias $\Delta_I(S')$ and $\Delta_T(S_I)$ for Compliance Checks

When considering Tables 6.1 and 6.2 and the respective explanations one important conclusion is that we have to provide methods to calculate instance–specific bias $\Delta_I(S')$ if need be. Basically, we can provide an automatic calculation of $\Delta_I(S')$ for the following cases:

1. Calculating $\Delta_I(S')[projection]$ if
   - $\Delta_I[projection] \prec \Delta_T[projection]$
     (with $projection \in$
     $\{ins\_Act, del\_Act, move\_Act, ins\_Sync, del\_Sync, ins\_Loop, del\_Loop, data, attrChange\}$) or
   - $\Delta_I[projection] \between \Delta_T[projection]$
     (with $projection \in \{del\_Act, del\_Sync\}$)
2. Calculating $\Delta_I(S')$ if $\Delta_T \curlyvee \Delta_I$

Algorithm 10 (cf. Appendix D) checks whether one of the above cases is given. Then it starts with calculating the differences between the consolidated sets of Δ_I and Δ_T (cf. Definition 13).

Based on this, we can, for example, state which activities have been inserted by Δ_I but not by Δ_T. Starting from this, the context of these additionally inserted activities within S'_I are determined and a respective entry within the change logs capturing $\Delta_I(S')$ is generated.

Example 6.28 (Calculating Instance–Specific Change (1)): Consider Figure 6.28 where $\Delta_T \prec \Delta_I$ holds since Δ_I has additionally inserted activities *remind*, *repeat*, and *stop* when compared to Δ_T. According to Migration Strategy 3, I can be re–linked to S' but we have to calculate bias $\Delta_I(S')$ on S' by applying Algorithm 10. Doing so, we first determine $Add := N^{add}_{\Delta_I} \setminus N^{add}_{\Delta_T}$ $= \{remind, repeat, stop\}$. Then it is decided whether the activities contained in set *Add* are inserted in a separated way or context–dependently. Obviously, *remind*, *repeat*, and *stop* are inserted "in a row" and are therefore included in set $Row = \{< remind, repeat, stop >\}$ of ordered context–dependently inserted activities. Afterwards anchor *(collect data, pack goods)* for newly inserted activities *remind, repeat*, and *stop* is determined and the first entry of $\Delta_I(S')$ results in $insertBetweenNodeSets(S', remind, collectdata, packgoods)$. Note that by applying this change operation we obtain the desired parallel branching within S'. Then the remaining entries of $\Delta_I(S')$ are determined as $serialInsert(S', repeat, remind, packgoods)$ and $serialInsert(S', stop, repeat, packgoods)$.

The following Example 6.29 shows that Algorithm 10 also works fine if activities are inserted and moved in a context–dependent way by Δ_I (but not by Δ_T):

Example 6.29 (Calculating Instance–Specific Change (2)): Look at Figure 6.29 where again $\Delta_T \prec \Delta_I$ holds whereby Δ_I has additionally inserted activities X and Y between anchors C and D and has moved activity B to the position between X and Y. Applying Algorithm 10 yields $\Delta_I(S')$ as depicted in Figure 6.29. Obviously, the respective change log capturing $\Delta_I(S')$ can be applied to S' resulting in S'_I.

If Δ_I is subsumption equivalent with Δ_T (i.e., $\Delta_I \prec \Delta_I$) it is possible to determine $\Delta_T(S_I) := \Delta_T \setminus \Delta_I$. Doing so is important in order to maintain correctness for S'_I regarding state–related conflicts between $\Delta_T \setminus \Delta_I$ and I. The respective algorithm (cf. Algorithm 11, Appendix D) works analogously to Algorithm 10 applied in case $\Delta_T \prec \Delta_I$ holds. The resulting change log capturing $\Delta_T(S_I)$ together with S_I contains all necessary information for evaluating our precise state conditions set out in Section 4.3.2.

6.8 Summary

In this chapter, we have developed methods which allow us to adequately classify instances changes with respect to their particular degree of overlap with subsequent process type changes. Our main goal is to support users by offering as many automatisms as possible. Especially, for application–neutral changes we want to simplify daily work by offering the formal foundation for an intelligent change management within PMS. Therefore, depending on the particular degree of overlap between process type and process instance changes (semi–) automatic migration strategies can be offered.

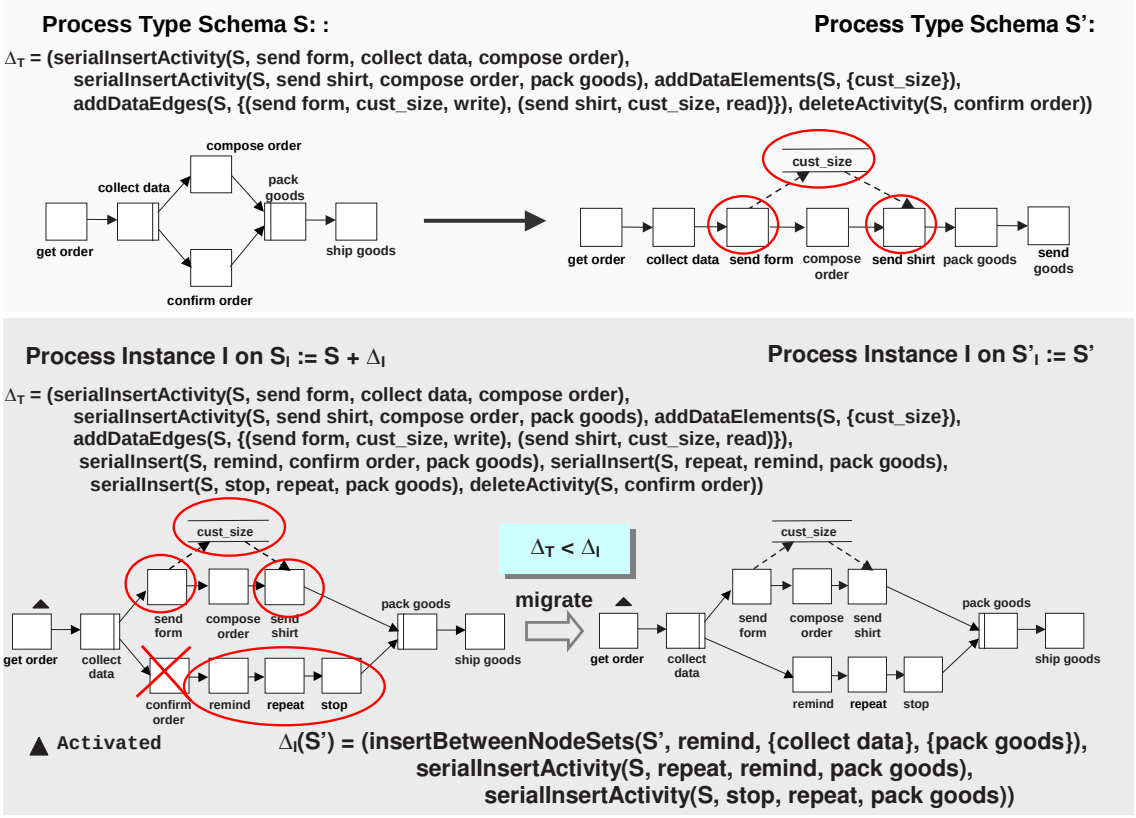


Figure 6.28: Calculating Instance–Specific Change (1)

The classification of instances along their particular degree of overlap was based on the formal notions of trace equivalence and process schema isomorphism. Since it was shown that this is still not sufficient to capture all possible scenarios we have discussed structural and operational approaches and have shown their specific limitations and advantages. Starting from this, the key to solution was to combine both methods to a hybrid approach. The core of the hybrid approach is the purging of change logs from noisy information on the one hand and determining consolidated activity sets, anchor sets, and order sets on the other hand. Based on this, finally, sufficient definitions for subsumption and partially equivalent changes could be found.

For partially equivalent changes we have also shown that the existing classification is still to coarse. Therefore we introduced projections on the particular change type. Using these change projections increases the number of process instances with overlapping bias for which an automatic migration strategy can be applied. This, in turn, contributes to the usability of our solution.

We provided adequate migration strategies to be proposed as default strategies within the system. So users can either choose the suggested strategy or can specify other migration strate-

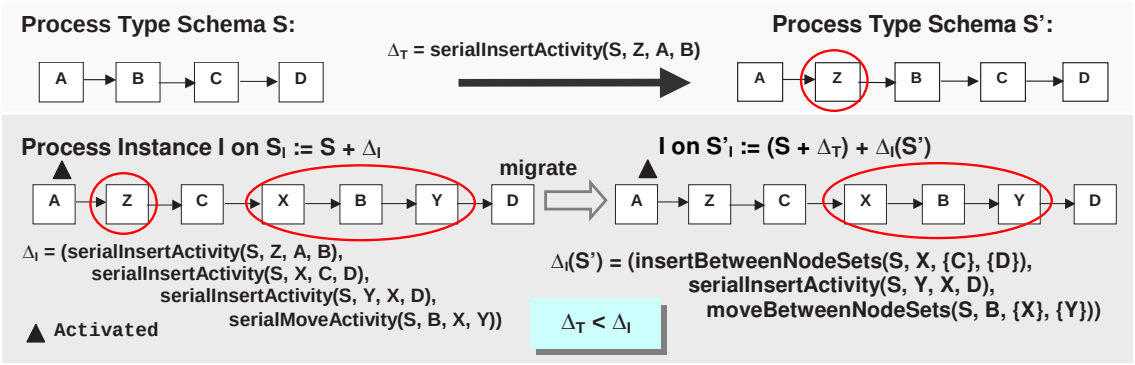


Figure 6.29: Calculating Instance–Specific Change (2)

gies. For those instances for which no default strategy is possible we developed rules which serve as decision help for users. Finally, we developed algorithms to automatically calculate resulting bias on the changed process type schema where automatic strategies are possible.

Table 6.1: Decision Rules for Partially Equivalent Process Type and Instance Changes (1)

Δ_T and Δ_I partially equivalent, i.e., $\Delta_T \between \Delta_I$	
1) Different Order:	
i) $N^{add}_{\Delta_T} = N^{add}_{\Delta_T} \wedge AnchorIns(S, \Delta_T) = AnchorIns(S, \Delta_I) \wedge$ $OrderIns(S', \Delta_T) \neq OrderIns(S', \Delta_I)$ $\Longrightarrow \Delta_T[ins\_Act] \overset{\leftrightarrow}{\equiv} \Delta_I[ins\_Act]$	equivalent insertion regarding activities and anchors but different order ➪ ask user about order
ii) $N^{move}_{\Delta_T} = N^{move}_{\Delta_T} \wedge AnchorMove(S, \Delta_T) = AnchorMove(S, \Delta_I) \wedge$ $OrderMove(S', \Delta_T) \neq OrderMove(S', \Delta_I)$ $\Longrightarrow \Delta_T[move\_Act] \overset{\leftrightarrow}{\equiv} \Delta_I[move\_Act]$	equivalent move regarding activities and anchors but different order ➪ ask user about order
iii) $\Delta_T[ins\_Act] \equiv \Delta_I[ins\_Act] \wedge \Delta_T[move\_Act] \equiv \Delta_I[move\_Act] \wedge$ $OrderAgg(S, \Delta_T) \neq OrderAgg(S, \Delta_I)$ $\Longrightarrow \Delta_T[ins/move\_Act] \overset{\leftrightarrow}{\equiv} \Delta_I[ins/move\_Act]$	equivalent insert and move operations but different order in combination ➪ ask user about order
iv) $N^{add}_{\Delta_T} \subset N^{add}_{\Delta_T} \wedge AnchorIns(S, \Delta_T) \subset AnchorIns(S, \Delta_I) \wedge$ $OrderIns(S', \Delta_T) \not\subset OrderIns(S', \Delta_I)$ $\Longrightarrow \Delta_T[ins\_Act] \overset{\leftrightarrow}{\equiv} \Delta_I[ins\_Act]$ and vice versa for $\Longrightarrow \Delta_I[ins\_Act] \overset{\leftrightarrow}{\prec} \Delta_T[ins\_Act]$	subsumption equivalent regarding activities and anchors but different order ➪ ask user about order
v) $N^{move}_{\Delta_T} \subset N^{move}_{\Delta_T} \wedge AnchorMove(S, \Delta_T) \subset AnchorMove(S, \Delta_I) \wedge$ $OrderMove(S', \Delta_T) \not\subset OrderMove(S', \Delta_I)$ $\Longrightarrow \Delta_T[move\_Act] \overset{\leftrightarrow}{\prec} \Delta_I[move\_Act]$ and vice versa for $\Longrightarrow \Delta_I[move\_Act] \overset{\leftrightarrow}{\prec} \Delta_T[move\_Act]$	subsumption equivalent regarding activities and anchors but different order ➪ ask user about order
vi) $(\Delta_T[ins\_Act] \equiv \Delta_I[ins\_Act] \wedge \Delta_T[move\_Act] \prec \Delta_I[move\_Act]) \vee$ $(\Delta_T[move\_Act] \equiv \Delta_I[move\_Act] \wedge \Delta_T[move\_Act] \prec \Delta_I[move\_Act]) \vee$ $(\Delta_T[ins\_Act] \prec \Delta_I[ins\_Act] \wedge \Delta_T[move\_Act] \prec \Delta_I[move\_Act])$ $OrderAgg(S, \Delta_T) \not\subseteq OrderAgg(S, \Delta_I)$ $\Longrightarrow \Delta_T[ins/move\_Act] \overset{\leftrightarrow}{\prec} \Delta_I[ins/move\_Act]$ and vice versa for $\Longrightarrow \Delta_I[move\_Act] \overset{\leftrightarrow}{\prec} \Delta_T[move\_Act]$	subsumption equivalent insert and move operations but different order for combination ➪ ask user about order
2) Different Anchors:	
i) $N^{add}_{\Delta_T} \subseteq N^{add}_{\Delta_T} \wedge AnchorIns(S, \Delta_T) \not\subseteq AnchorIns(S, \Delta_I)$ $\Longrightarrow \Delta_T[ins\_Act] \between \Delta_I[ins\_Act]$ and vice versa	same activities but different anchors ➪ ask user about anchors
ii) $N^{move}_{\Delta_T} \subseteq N^{move}_{\Delta_T} \wedge AnchorMove(S, \Delta_T) \not\subseteq AnchorMove(S, \Delta_I) \wedge$ $\Longrightarrow \Delta_T[move\_Act] \between \Delta_I[move\_Act]$ and vice versa	same activities but different anchors ➪ ask user about anchors

Table 6.2: Decision Rules for Partially Equivalent Process Type and Instance Changes (2)

Δ_T *and* Δ_I *partially equivalent, i.e.,* $\Delta_T \between \Delta_I$	
3) Conflicting Target Context (cf. Definition 20):	
$\Delta_T \curlyvee \Delta_I \Longleftrightarrow$ $\exists (src, dest) \in$ $(E_{\Delta_1}^{conc\_context}[ins] \cup E_{\Delta_1}^{conc\_context}[move]) \cap$ $(E_{\Delta_1}^{conc\_context}[ins] \cup E_{\Delta_2}^{conc\_context}[move])$	calculate $\Delta_I(S')$ (cf. Alg. 10) ➪ suggest $\Delta_I(S')$ to users of $\Delta_I(S')$ on S'
4) Multiple Operations:	
i) $N_{\Delta_T}^{add} \cap N_{\Delta_I}^{add} \neq \emptyset \Longrightarrow$ $\Delta_T[ins\_Act] \between \Delta_I[ins\_Act]$	detect which activities are inserted twice ➪ report activities (+ anchors) to user
ii) $N_{\Delta_T}^{del} \cap N_{\Delta_I}^{del} \neq \emptyset \Longrightarrow$ $\Delta_T[del\_Act] \between \Delta_I[del\_Act]$	calculate $\Delta_I(S')$ (cf. Alg. 10) ➪ suggest $\Delta_I(S')$ to users
iii) $N_{\Delta_T}^{move} \cap N_{\Delta_I}^{move} \neq \emptyset \Longrightarrow$ $\Delta_T[move\_Act] \between \Delta_I[move\_Act]$	detect which activities are moved by Δ_T and Δ_I ➪ report activities (+ anchors) to user
For the deletion of sync edges act like for the deletion of activities. For all other projections, e.g., for inserting data elements and changing attributes, act like for the multiple insertion of activities.	
5) Context–Destroying Operations (cf. Definition 21):	
$\boldsymbol{\Delta_T} \curlyvee \Delta_I$	determine destroyment of Δ_T reg. Δ_I ➪ report to user
6) Data Flow Changes:	
$\Delta_T[data] \between \Delta_I[data]$	➪ ask user in any case

Chapter 7

Proof–Of–Concept Prototype

In order to prove practical feasibility of our (theoretical) framework for (dynamic) process changes we have developed a powerful software prototype with integrated process modeling and monitoring components [71, 100]. One major goal in design and implementation was to ensure correctness of process instance migration to a changed process type schema for each kind of process instance and at any point in time. In detail, the prototype offers the possibility to generate and change process type schemes. Based on the process type schemes, process instances can be created and started whereby their state is represented by model–inherent markings and stored within execution histories. Process instances can also be ad hoc modified. Then the instance–specific schema reflects the modifications which are logged within a change history. If a process type schema is changed a new version is derived. At first, the prototype checks state–related compliance (cf. Chapter 4) for all running process instances and migrates the compliant unbiased process instances to new schema version. Then all biased process instances are classified along their particular degree of overlap (cf. Chapter 6). For process instances with disjoint bias the structural conflict tests (cf. Chapter 5) are applied. For process instances with overlapping bias their specific degree of overlap is reported and compliant process instances with equivalent and subsumption equivalent bias are migrated to the changed process type schema.

As we will discuss in Section 7.2, the correctness criteria developed in this work and related checks have been realized within the prototype. Furthermore, we paid a lot of attention on implementation and efficiency issues as well since classification of process instances, compliance checks, and instance migrations must be carried out at runtime. As shown in Chapters 4 – 6 many optimizations regarding efficiency have been already conducted on the logical level. Examples include the provision of precise compliance conditions to efficiently check state–related compliance (cf. Section 4.3.2), the quickly applicable conflict tests (cf. Section 5.4), and the optimized instance classification by means of the presented hybrid approach (cf. Section 6.5). These logical optimizations have been realized within the prototype as well. Additionally, further optimization on the physical level have been carried out, e.g., concerning the intelligent management of process type and process instance objects [71]. Overall, the realization of this

proof–of–concept prototype impressively demonstrates our framework for process change management and shows in which direction future adaptive PMS must go in order to finally deliver in practice.

The remainder of this chapter is organized as follows: In Section 7.1, we present the architecture of our proof–of–concept prototype. Section 7.2 shows how this demonstrator works along different examples. Finally, Section 7.3 closes with a summary of the presented results.

7.1 Architectural Considerations

Our prototype is a stand–alone solution which allows to demonstrate all important aspects of process change management. The implementation is completely based on Java. The demonstrator offers the possibility to store process *templates* and process instances as XML files . The latter includes information about current instance states (described by node and edge markings, data element values, and execution histories) and instance–specific bias. Template changes (at process type level) are reflected by maintaining different schema versions. Finally, changes at the process type and the process instance level are additionally stored within respective (purged) change histories (cf. Section 6.5).

Figure 7.1 depicts the underlying system architecture which consists of different modules (called *managers* in the following). The *Demo–Client* provides the possibility to establish and change process templates and process instances and to monitor process instance execution. In order to carry out process instance activities the *Worklist Manager* offers respective work items and the *Execution Manager* controls the progress of instance executions and cares about related marking adaptations. If a template change takes place a new schema version for this template is generated by the *Template Manager*. The *Change Manager* is then responsible for propagating the template changes to related process instances and to enable their migrations to the new schema version (if possible). In addition, process instances can be individually modified by the *Execution Manager*. Finally, the *Logging Module* manages and stores execution and change logs.

We now describe the different managers in detail:

- *Template Manager:* A new template can be defined by means of the *Demo–Client* which offers a set of change operations for this purpose. This, in turn, leads to the creation of an internal template object. The *Template Manager* loads and stores process templates from/into XML files. Furthermore, template changes (e.g., insertion of activities) and the required template versioning are accomplished by this component. If a displayed template has been changed a view update is triggered within the *Demo–Client*. Finally, the *Template Manager* informs the *Change* as well as the *Logging Manager* if a template has been changed.
- *Execution Manager:* With this component new process instances can be created. This is

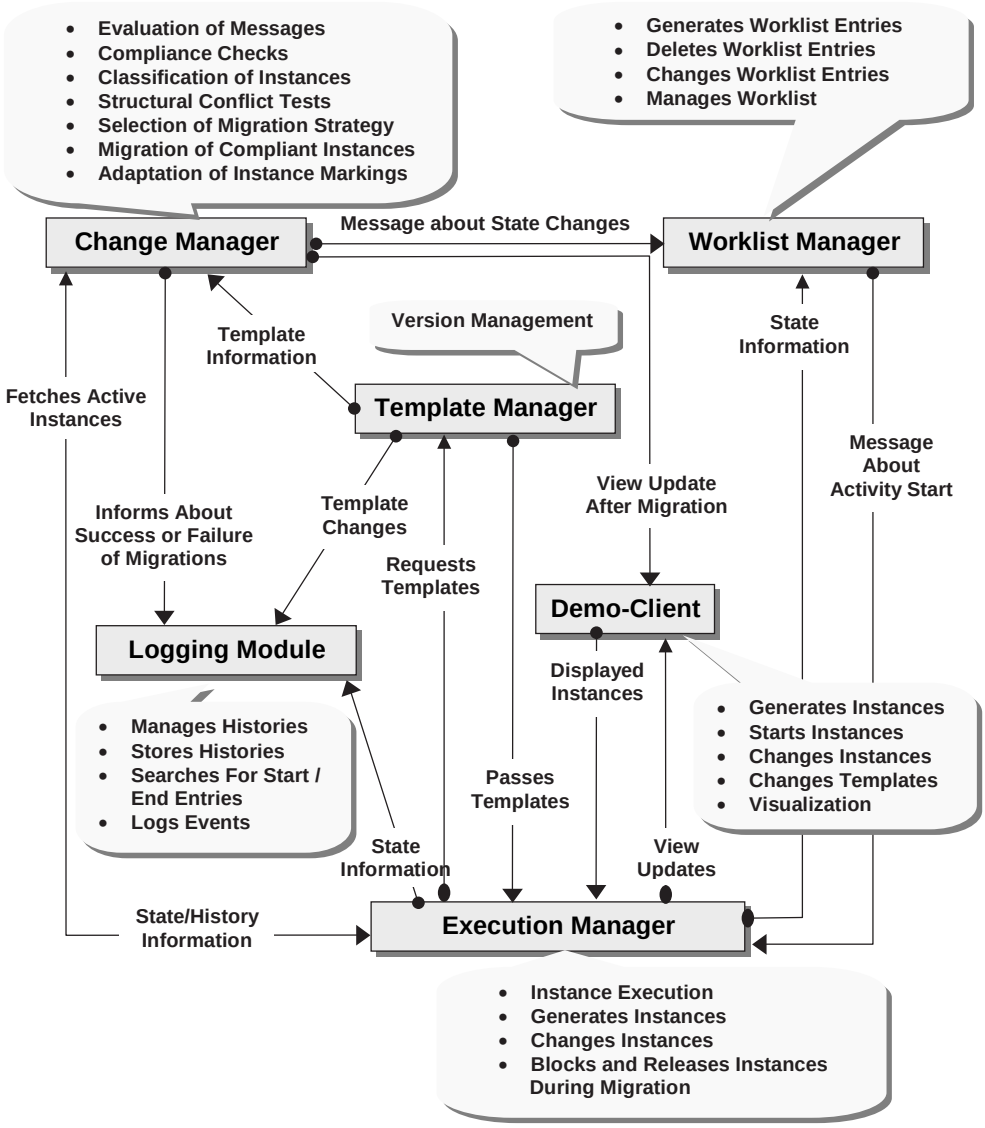


Figure 7.1: System Architecture of Proof–Of–Concept Prototype

done by fetching the respective template, by creating a process instance object, and by initializing this object (e.g., by setting the inital status of process start nodes to `Activated`). Instance states are updated according to messages of the *Worklist Manager*. The *Execution Manager*, in turn, informs the *Worklist Manager* and the *Demo–Client* when an instance state is changed. Furthermore, information about such state changes is sent to the *Logging Manager*. Finally, the *Execution Manager* is responsible for locking instances during compliance checks and migrations, and for releasing these locks afterwards.

- *Logging Module:* This module manages execution and change histories for each instance. For templates change histories are maintained as well. Furthermore, all relevant events like `Start` and `End` events for activity executions or instance–specific changes are logged.

- *Change Manager:* This manager receives messages about template changes from the *Template Manager* and evaluates them respectively. Furthermore, it carries out state–related compliance checks based on the state and the history information passed by the *Execution Manager*. For compliant process instances the *Change Manager* classifies these instances along their particular degree of overlap. Again, for this purpose, information about the change logs of biased instances from the *Execution Manager* and (structural) template information from the *Template Manager* are needed. For process instances with disjoint bias, structural compliance is verified by applying the quick conflict tests introduced in Section 5.4. Based on the classification of the process instances the *Change Manager* selects the right (default) migration strategy. This means that compliant, unbiased process instances and compliant process instances with disjoint bias are re–linked to the new template version by applying respective state adaptations (cf. Section 4.4). For process instances with equivalent and subsumption equivalent bias the classification is reported to users and the default migration strategies are carried out. For process instances with partially equivalent bias the respective sub–classification (cf. Section 6.7) is reported to users such that they can specify desired migration strategies.

- *Worklist Manager:* Depending on messages from the *Execution Manager* this manager generates or updates worklist entries. Generally, it is responsible for worklist management. If a respective message from the *Change Manager* is received particular worklist entries may have to be reset (cf. Algorithm 2, Section 4.4). If the activity related to a work item is selected for execution a respective message is sent to the *Execution Manager*.

- *Demo–Client:* This monitoring and change client allows to create and change templates. It is also possible to create, start and change process instances. Furthermore, templates, instances, worklists, and histories are visualized. Finally, if a migration takes place a *migration report* (cf. Figure 7.4) is depicted. This report shows the classification of running process instances along their particular degree of overlap. Furthermore, it depicts which process instances are compliant with the new template and which amount of time was consumed for checking compliance and for migrating compliant process instances. For non–compliant process instances an explanation is given why they are considered as being non–compliant.

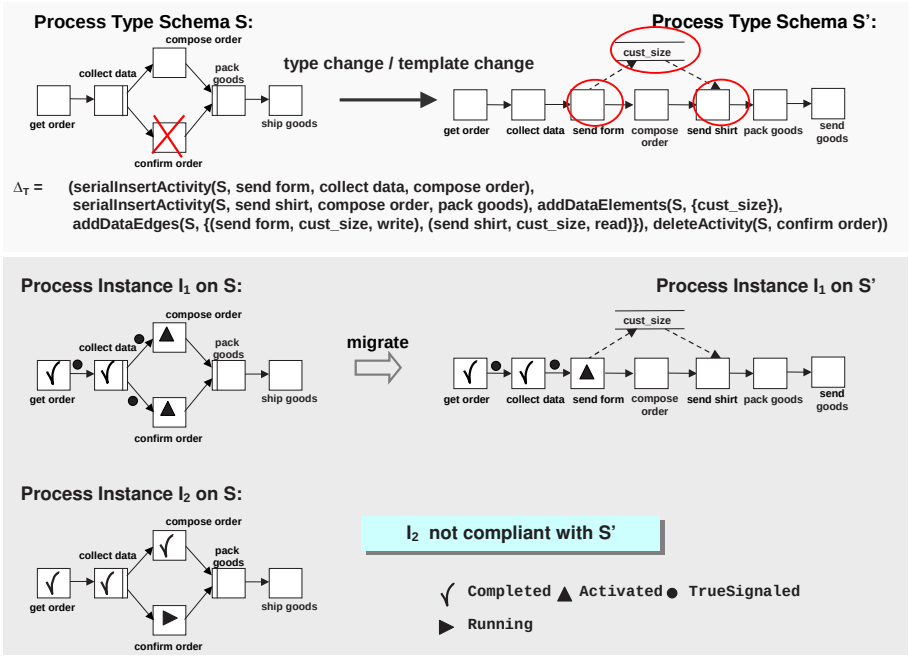


Figure 7.2: Migrating Unbiased Process Instances (Abstract Level)

After sketching the prototype architecture we illustrate its functioning by different examples in the following section.

7.2 Demonstrating a Complete Example

In this section, we systematically show how our prototype works by presenting examples for migrating unbiased and biased instances to a changed process type schema. Doing so, we introduce the basic functions of the demonstrator and refer to the theoretical concepts they are based on. At first, the respective example is depicted on an abstract level. Then we show screenshots regarding its realization within the prototype.

Example 7.2: Migrating Unbiased Process Instances (Abstract Level): The first example refers to the migration of unbiased process instances. As introduced in Chapter 4, compliance of unbiased instances depends on their particular state. Figure 7.2 shows two process instances in different execution states. Process type change Δ_T inserts activities *send form* and *send shirt* with a data dependency between them and deletes activity *confirm order*. Obviously, I_1 is compliant with S' and therefore can be migrated to S'. In contrast, I_2 is not compliant with S' (insertion before completed activity and deletion of running activity) and therefore remains running according to S.

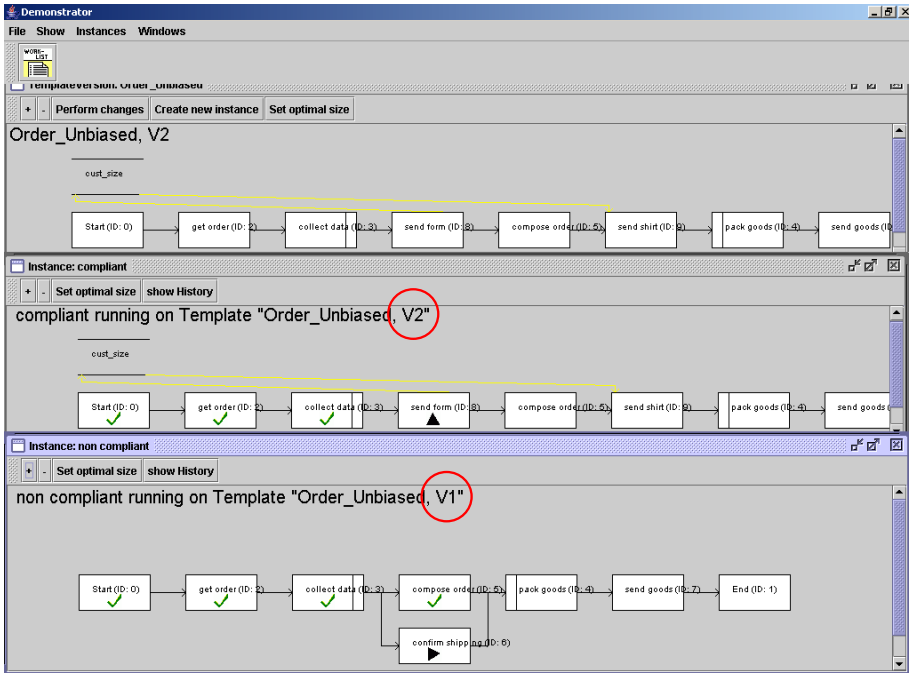


Figure 7.3: Migrating Unbiased Process Instances (Screenshot Prototype)

Example 7.3: Migrating Unbiased Process Instances (Screenshot Prototype): Taking Figure 7.2 and establishing the same example within the prototype we obtain the scenario as depicted in Figure 7.3. The prototype checks compliance of the two instances and correctly decides to migrate instance I_1 to the new version $V2$ of the respective process template. It also carries out the necessary marking adaptations. In contrast, I_2 (what corresponds to process instance *non compliant* within the prototype) is considered as being not compliant and therefore remains running according to the old version of the process template.

As we can see from the *Migration Report* depicted in Figure 7.4 the applicability of the demonstrator is not restricted to only one or two running instances. The scenario depicted in Figures 7.2 and 7.3 captures 2502 running process instances[1]. At this, 1501 instances were compliant with the changed template and could be migrated to it whereas 1001 instances progressed too far. The Migration Report also shows that compliance checks for 2502 instances took 16 ms of time.

Figures 7.2 – 7.4 show that our prototype works fine for the migration of unbiased process instances. The next examples refer to the migration of process instances with disjoint bias to a changed process type schema. At this, we have to care about state–related and structural

[1]For demonstration purposes the prototype offers the possibility to clone process instances. Therewith an arbitrary number of running process instances in a certain state can be achieved.

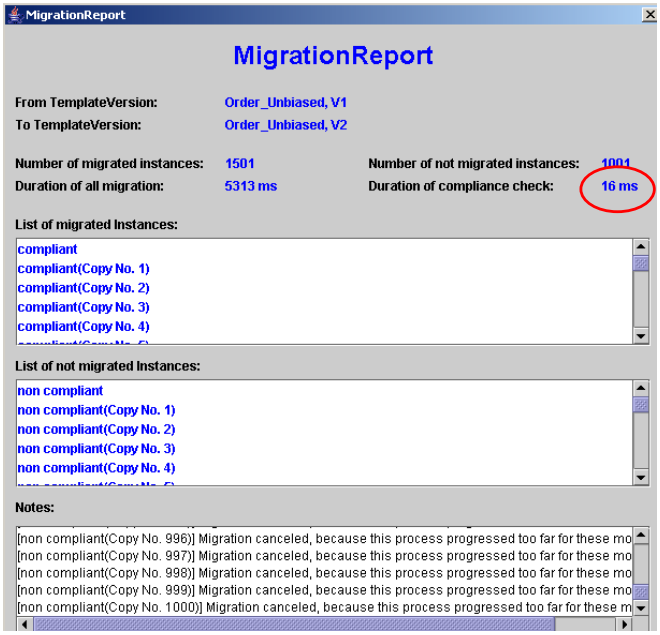


Figure 7.4: Migrating Unbiased Process Instances (Screenshot Migration Report)

compliance of the concerned instances. Similar to our state–related compliance conditions easily to check conflict tests have been established to ensure structural compliance (cf. Section 5.4).

Example 7.5: Disjoint Changes: Deadlock–Causing Cycle (Abstract Level): Consider Figure 7.5 where two biased process instances I_1 and I_2 (running on template S) are depicted. I_1 has been biased by inserting activity *remind* and a sync link from *remind* to activity *compose order*. Bias Δ_{I_2} consists of the serial insertion of activity *send form* between activities *pack goods* and *send goods*. Applying the structural conflict tests introduced in Section 5.4, I_1 is rated as being structurally non compliant with S' since instance–specific schema $S^*_{I_1} := (S + \Delta_{I_1}) + \Delta_T$ (which would result when propagating Δ_T to S_{I_1}) contains a deadlock–causing cycle. In contrast, I_2 is structurally compliant with S' since no test indicates a structural conflict between Δ_T and I_2. Furthermore, I_2 is compliant regarding its state (cf. Section 4.3.2). Consequently, it is possible to migrate I_2 to S' by applying Migration Strategy 1 (cf. Section 5.5). This means that I_2 simply can be re–linked to S' by maintaining original bias Δ_{I_2} on S'.

Now let us have a look how the scenario depicted in Figure 7.5 turns out within our proof–of–concept prototype.

Example 7.6: Disjoint Changes: Deadlock–Causing Cycle (Screenshot Prototype): Looking at Figure 7.6 we can see that, generally, biased process instances are signed with the add–on *"adhoc modified"*. The respective bias is graphically depicted within the respective instance–specific schemes and stored within individual change histories. After propagating the template

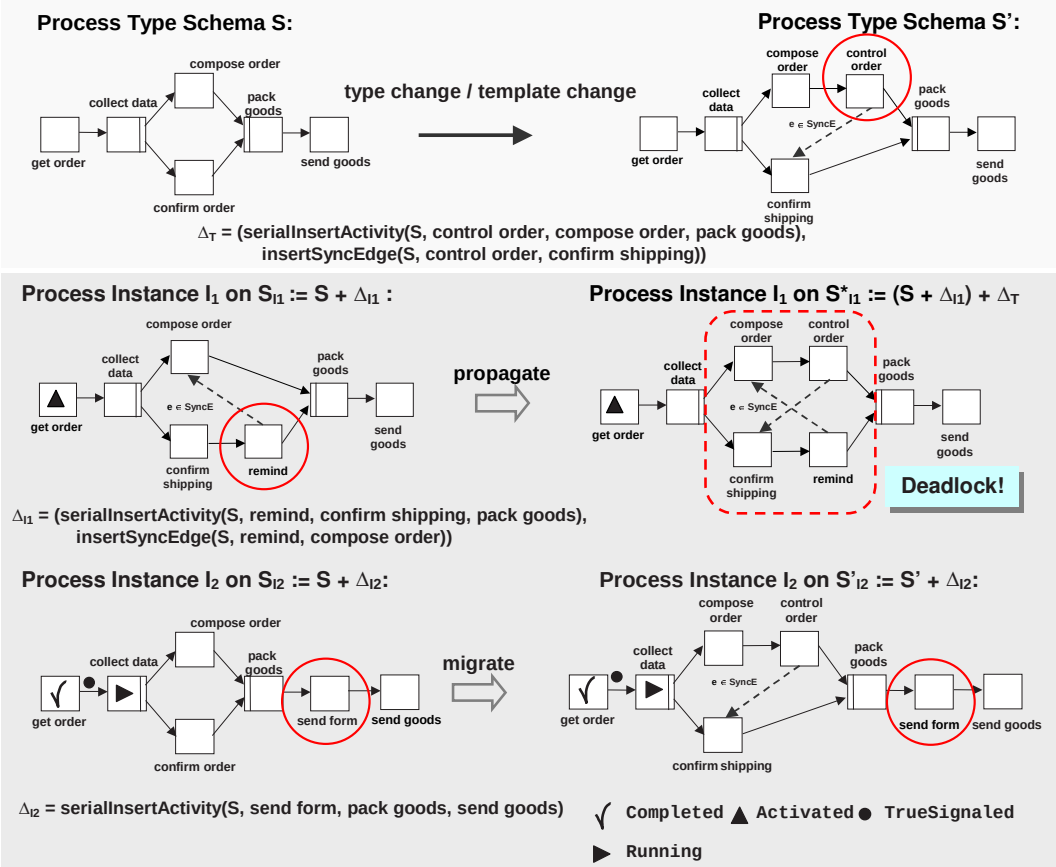


Figure 7.5: Disjoint Changes: Deadlock–Causing Cycle (Abstract Level)

changes to the running instances the *Migration Report* yields that instance I_2 (what corresponds to process instance *compliant* within the prototype) was compliant and therefore migrated to the second version $V2$ of template *Order_Disjoint1*. In contrast, the *Migration Report* shows that instance I_1 (what correponds to process instance *deadlock* within the prototype) could not be migrated due to the fact that there would be a deadlock–causing cycle within the resulting instance–specific schema.

A second scenario for migrating process instances with disjoint bias shows that our prototype is also able to correctly deal with data flow conflicts.

Example 7.7: Disjoint Changes: Missing Input Data (Abstract Level): The respective scenario is depicted in Figure 7.7. Here process instance I is biased due to the deletion of activities *compose order* and *pack goods* as well as to the concomitant deletion of related read and write data links on data element *goods*. Type change Δ_T serially inserts activity *control goods* with read data link connected to data element *goods*. If Δ_T is propagated to S_I in an uncontrolled

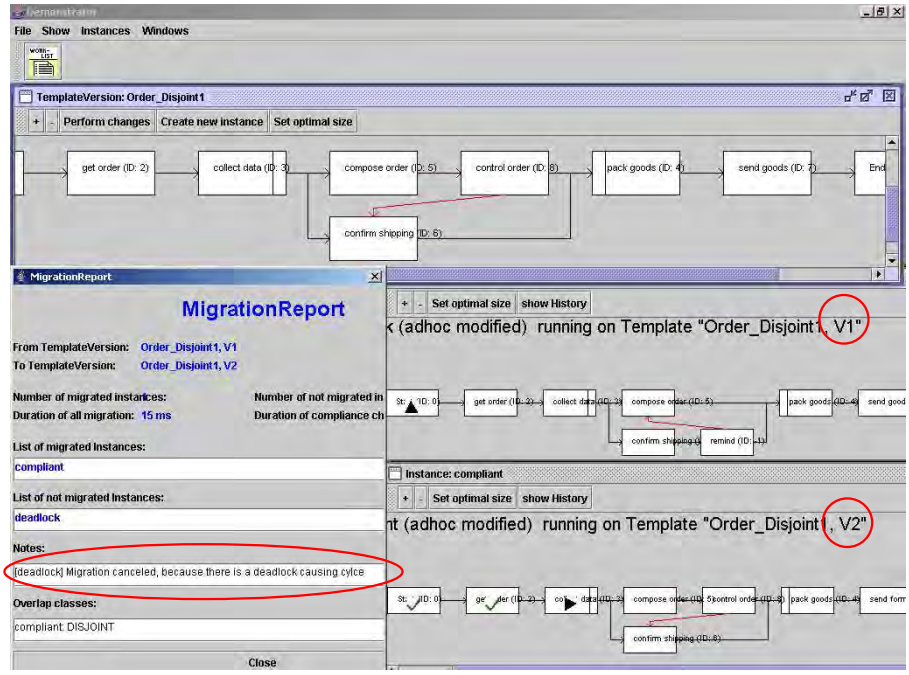


Figure 7.6: Disjoint Changes: Deadlock–Causing Cycle (Screenshot Prototype)

manner the resulting instance–specific schema $S_I^* := (S + \Delta_I) + \Delta_T$ will not be correct. The correctness constraints set out by WSM Nets (cf. Definition 2, Section 3.1.1) are not satisfied since the input parameter of activity *control goods* is no longer supplied.

How the scenario depicted in Figure 7.7 is realized within the prototype is shown by the following example.

Example 7.8: Disjoint Changes: Missing Input Data (Screenshot Prototype): Consider Figure 7.8 where we can see biased process instance I and intended template change Δ_T from Figure 7.7. As the *Migration Report* shows the instance is excluded from migration to the changed template since there would be data flow conflicts within the resulting instance-specific schema.

Finally, we want to present a scenario for the migration of process instances with overlapping bias to a changed process type schema. As an interesting example we present the migration of process instances with equivalent bias (cf. Chapter 6).

Example 7.9: Equivalent Changes (Abstract Level): Look at Figure 7.9 where process type change Δ_T and instance change Δ_I are equivalent since the instance–specific schema S_I is trace equivalent with the new type schema S' (cf. Definition 11). According to Migration Strategy 2 (cf. Section 6.6) for instances with equivalent bias we can re–link I to S' at any time. Doing so the resulting instance–specific bias $\Delta_I(S')$ on S' becomes empty.

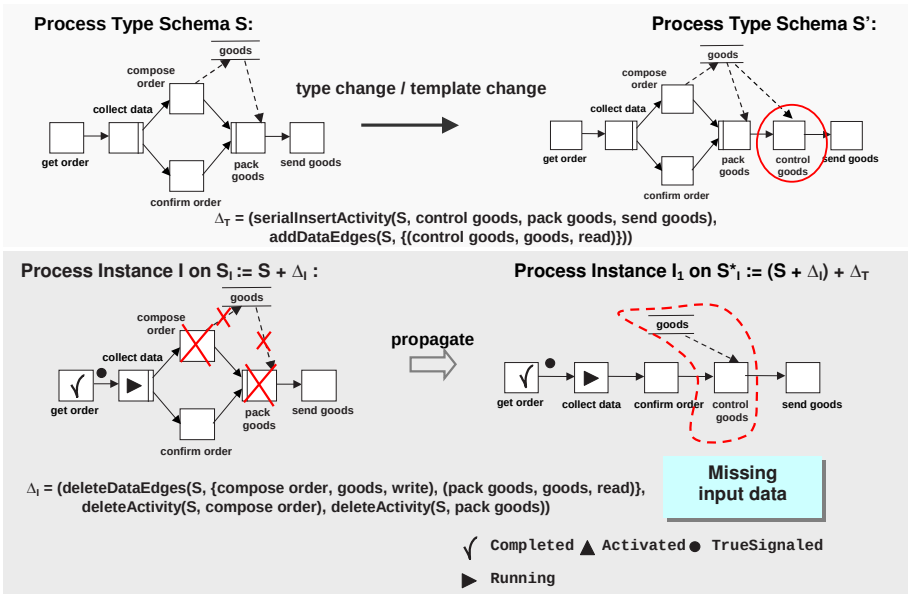


Figure 7.7: Disjoint Changes: Missing Input Data (Abstract Level)

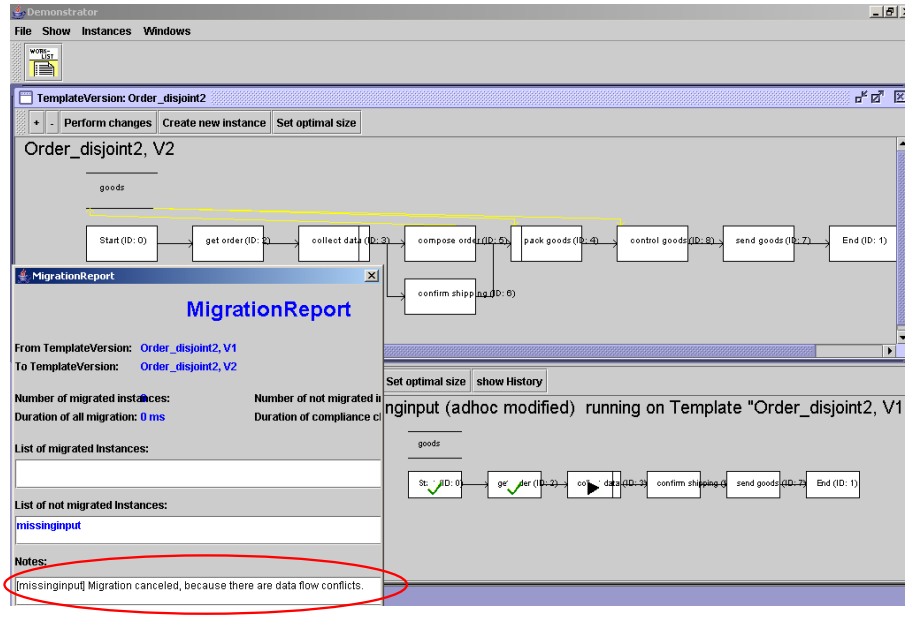


Figure 7.8: Disjoint Changes: Missing Input Data (Screenshot Prototype)

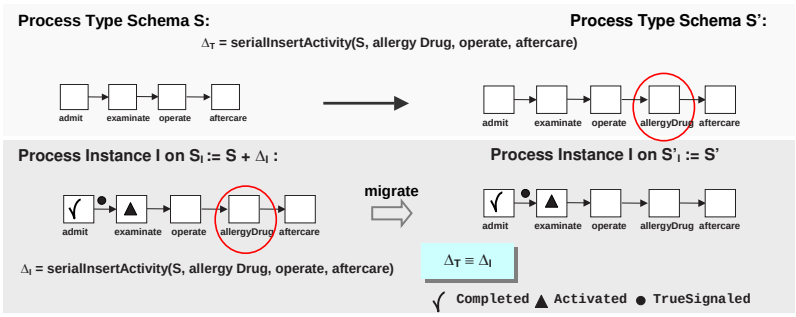


Figure 7.9: Equivalent Changes (Abstract Level)

The transfer of the respective scenario to our prototype is depicted in Figure 7.10.

Example 7.10: Equivalent Changes (Screenshot Prototype): Consider Figure 7.10. Obviously process template *Clinic_Equivalent1*, *V2* corresponds to process type schema S' and instance *equiv1* corresponds to process instance I after migration to S'. From the *Migration Report* we can see that instance *equiv1* was correctly estimated as being equivalent. Based on this classification the default migration strategy was chosen by re–linking *equiv1* to new template version $V2$ and by nullifying bias $\Delta_I(S')$ on S'. This is depicted by retracting the add–on *"adhoc modified"* of *equiv1* (what means that *equiv1* is running on template *Clinical_Equivalent*, *V2* as an unbiased process instance).

Recapitulating this section, we can state that our prototype supports the migration of unbiased instances as well as the migration of instances with disjoint and overlapping bias, in an adequate and user–friendly way. Therefore it is possible to demonstrate the formal concepts of this work and to show in which direction future adaptive PMS must go.

7.3 Summary and Outlook

In this chapter, the developed proof–of–concept prototype for process schema evolution was presented. At first, the basic architecture of this system was introduced. Then important functions of the prototype were demonstrated by means of different change scenarios. These scenarios have illustrated that the prototype supports process type changes as well as process instance changes and adequately deals with the interplay between them. More precisely, when a process type schema is changed for the unbiased process instances state–related compliance is checked and the compliant instances are automatically migrated to the new schema version. Biased process instances are classified along their particular degree of overlap. For process instances with disjoint bias also structural test are carried out. For process instances with equivalent or subsumption equivalent bias the adequate migration strategy is selected and applied. Altogether, this chapter comprises some fundamental concepts for implementating adaptive PMS.

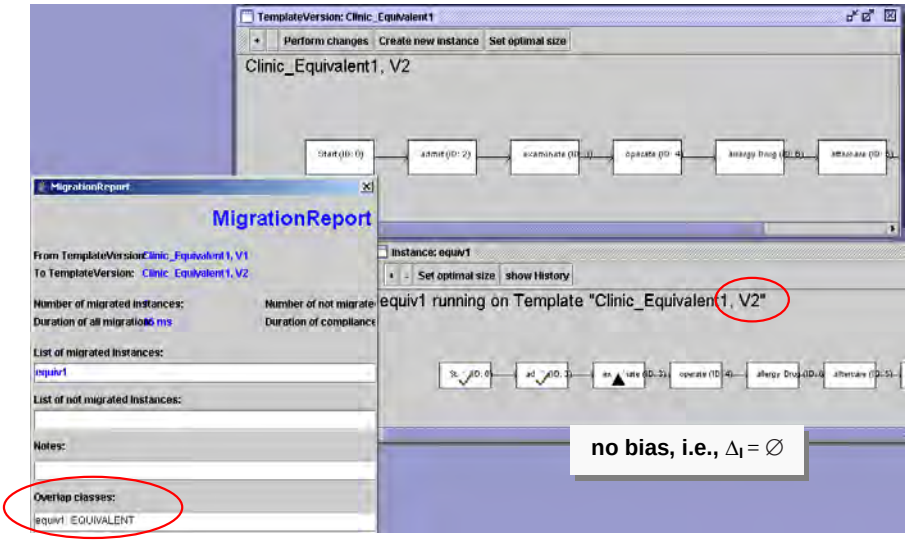


Figure 7.10: Equivalent Changes (Screenshot Prototype)

At the moment our ADEPT PMS prototype [90], which is currently used by several partners [10, 11, 81] as a platform for realizing advanced PAIS (e.g., supporting e–business and clinical workflows) is re–implemented within the AristaFlow project. This project is a cooperation between the Universities of Ulm and Mannheim and several companies (for details see [86]). Within this new prototype the lessons learned from the implementation of the proof–of–concept prototype will be incorporated.

Chapter 8

Summary

A turbulent market and continously arising new IT trends (e.g., e–government, e–auctions, web services) necessitate the integrated and flexible support of business processes. Therefore, throughout the software world process–awareness is a hot topic nowadays (e.g., a lot of research on composing (web) services in a process–oriented manner is being performed). Process management systems offer promising perspectives due to the separation of process logic from the application code; i.e., the process logic is explicitly modeled and deposited within the system and not "hard–wired" within the application programs. This approach offers many advantages. Since process executions are controlled by the process management system both application development and application maintenance are simplified. Furthermore, process changes tend to become more easy since business process changes can be accomplished by adapting the respective process model, i.e., one must not dig into application program code.

Though the separation of process logic and application code provides the basis to be able to change process instances at runtime this is not sufficiently realized in today's systems. Commercial process management systems either forbid changes of process instances (e.g., MQSeries Workflow) or strongly restrict them (e.g., Ultimus). Other systems allow process instance changes but they may lead to inconsistencies and errors in the sequel (e.g., Staffware). This missing flexibility is the main reason for process management systems not being widely spread in practice.

The main goal of this work was to develop a formal framework which enables process management systems to offer a correct, comprehensive, and user–friendly change management. This requires the support of propagating process type changes to running process instances and to migrate them to the changed process type schema afterwards. At this, the challenge was to handle a large number of (long–running) process instances all being in different execution states but also to deal with process instances which have been already individually modified. In particular, the question was not to allow either process type changes or process instance changes but to support their interplay, i.e., the migration of biased process instances to the changed process type schema. This requires a deep understanding of the possible relations between pro-

cess changes and the resulting migration strategies. We provided a prototypical implementation of the developed concepts which supports the migration of unbiased as well as biased process instances to the changed process type schema in a correct and efficient way. The most important results of this work can be summarized as follows:

Migrating Unbiased Process Instances

Basic to process change management is a sound approach for checking compliance of unbiased process instances with the changed process type schema and for migrating compliant instances to this schema. Thereby, special attention must be paid on correctness and consistency as well as on efficiency and usability of the presented concepts. In this work, these challenges were satisfied by establishing a comprehensive correctness criterion, by providing efficient compliance checks, and by automatically adapting states after migrating process instances to the new process type schema.

In detail, we have analyzed different correctness criteria from the literature suggested for migrating process instances to a changed process schema. From this, we have identified compliance as an adequate criterion for meta models storing process instance states as explicit markings. A process instance is said to be compliant with the changed process type schema if its previous execution (represented by its execution history) can be reproduced on the changed process type schema. We recognized that the quality of this compliance criterion depends on which view on execution histories is used. We formally proved that the `Start/End` view on an execution history ensures a correct migration of process instances. Furthermore, we found out that this view is still too restrictive in conjunction with the use of iterative process constructs (i.e., loops). Therefore we developed a special view on execution histories. This so called loop–tolerant view is obtained by (logically) discarding all entries of already finished loop iterations. Using this view we obtain a basis for correct process instance migration but without unnecessary exclusion of process instances from being migrated.

After introducing a formal correctness criterion the challenging question was how to efficiently ensure this criterion in the context of WSM Nets. For this, we provided precise state conditions which make use of the model–inherent markings of WSM Nets as well as of the semantics of the applied change operation. We presented a method to check compliance with complexity $O(n)$ (whereby n corresponds to the number of activities comprised by the respective process instance schema). In average, however, process instance data has to be accessed 2 times.

To correctly and efficiently decide which process instances are compliant is only one side of the coin. The other is to provide efficient methods to automatically adapt instance markings on the changed process type schema. In any case, users must not be burdened with this task. Therefore we developed an algorithm to automatically migrate compliant process instances to the changed process type schema, i.e., to re–link these instances to the new type schema and to automatically adapt their markings correspondingly. The algorithm developed for this purpose is based on the marking and adaptations rules of WSM Nets as well as on the semantics of the applied change operation. With a complexity of $O(n)$ (whereby n corresponds to the number of activities within the respective process instance schema) this algorithm can be effectively applied

even in large–scale environments.

In summary, the presented methods and algorithms for migrating of (unbiased) process instances to a changed process type schema are not bound to the formalism of WSM Nets, i.e., they can be easily transferred to other process meta models with model–inherent markings like, for example, Activity Nets as well. We have shown this in [91].

In order to offer a really flexible process management system it is necessary to support changes at the process instance level as well. Thereby it is not sufficient to separate the treatment of changes at the process type and at the process instance level from each other. It must be also possible to adequately deal with the interplay of process type and process instance changes, i.e., to be able to migrate biased process instances to the changed process type schema as (if desired by users and possible). Therefore this work dealt with the concurrent application of process type and process instance changes as well.

Migrating Process Instances With Disjoint Bias

To put the discussion about the interplay between process type and process instance changes on a solid basis, first of all, we put focus on the relationships between such changes. The core of this discussion was the distinction between changes which have totally different effects on the underlying process schema (e.g., changes which affect different regions of this schema) and changes which have (partially) overlapping effects. This distinction between disjoint and overlapping changes is important since process instances with disjoint bias (i.e., the instance–specific bias and the process type change are disjoint) must be treated in another way than process instances with overlapping bias when migrating them to a changed process type schema. Whether a process instance has a disjoint or overlapping bias does not only affect necessary compliance checks and marking adaptations, but does also affect the (new) bias of this instances when being migrated to the changed process type schema. In order to obtain a formal basis we defined the notations of disjoint and overlapping changes.

For migrating process instances with disjoint bias similar challenges exist as for the migration of unbiased process instances. Of course, it is important to ensure correctness and consistency at any time. The migration of process instances with disjoint bias imposes state–related as well as structural correctness. Consequently, we presented a correctness criterion preserving both properties when migrating process instances. Another challenge was to find methods to efficiently ensure structural correctness when migrating compliant process instances with disjoint bias. In order to achieve this we presented easily and quickly to check structural conflict tests. These conflict tests indicate structural inconsistencies within the process instance schema when propagating the process type change to it (like, for example, deadlock–causing cycles or missing input data). Using them, in particular, it is possible to ensure structural correctness without need for materializing the schema reflecting process instance and the process type change (and consequently without need for explicit compliance checks on this schema). Instead, the conflict tests are solely based on already present information (about applied process instance/process type changes and about the original process schema). Together with the developed precise state conditions the structural conflict tests ensure compliance of process instances with disjoint bias.

Finally, an important task was to find an adequate migration strategy for process instances with disjoint bias. Thereby the goal was to avoid expensive and annoying user interactions. We have shown that compliant process instances with disjoint bias can be automatically re–linked to the changed process type schema while preserving their entire bias. Thus, possibly expensive bias re–calculations on the changed process type schema do not become necessary in this case. Furthermore, the migration can be completely carried out by the system.

In summary, the presented results for migrating process instances with disjoint bias are fundamental (for any adaptive PMS). Furthermore, they can be easily transferred to other process meta models as well.

The last step towards a flexible process change management is to master the migration of process instances with overlapping bias as well.

Migrating Process Instances With Overlapping Bias

For any adaptive PMS it is indispensable to provide solutions for migrating process instances with overlapping bias to a modified process type schema. To be able to do so we firstly presented a classification for overlapping changes along their particular degree of overlap. This classification ranges from equivalent changes (having completely the same effects on the respective process schema) to partially equivalent changes with only minor overlap. This classification is very important in order to find adequate migration strategies. Note that the choice of the right migration strategy depends on the particular degree of overlap between process type and process instance change.

On the way to find formal definitions for the different kinds of overlapping changes, we discussed several approaches. They ranged from structural ones directly comparing process schemes to operational ones directly contrasting changes. Starting from these approaches we combined their particular strenghts to a hybrid approach. This hybrid approach determines all inserted and deleted elements (e.g., activities or edges) by applying a structural approach. In addition, operational methods are used to determine shifted activities. Thereby, a key element of this approach is to purge change logs from noisy information. With the information gained by applying the hybrid approach we were able to define the different kinds of overlapping changes along their particular kind of overlap.

For all kinds of overlapping changes adequate migration strategies were provided. These strategies contain information about necessary compliance checks and process instance adaptations on the changed process type schema. The latter include marking adaptations of respective instances as well as the re–calculation of their bias regarding the changed process type schema. In order to make the classification along the degree of overlap more precise we introduced change projections on the different change types, e.g., projections on activity insertions or deletions. Doing so we obtained a fine–granular classification for overlapping changes what leads to a more precise and therefore more optimized application of the migration strategies.

For partially equivalent changes for which no automatic migration strategy can be found we indicated how to adequately assist users by corresponding messages. These messages exactly

report the particular scenario to users and provide decision guidance for them.

Altogether a complete solution for the adequate treatment of process type and process instance changes was provided in this work. Furthermore, we implemented a proof–of–concept prototype in order to demonstrate the presented concepts. In summary, this thesis provides a complete framework for application–neutral change management in process management systems.

Outlook

Business processes are the "jewels" of every company. The process life cycle [123, 124] summarizes the different life phases of a business process: the process design phase, the configuration phase (during which the processes are implemented), the enactment phase where processes are executed, and the diagnosis phase where instance logs are analyzed in order to further optimize the process instances (cf. Figure 8.1).

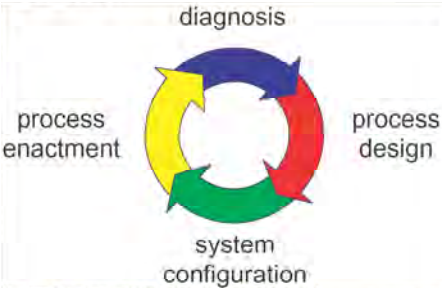


Figure 8.1: The Business Process Life Cycle [123]

Process change management will offer important support for the different phases of the life cycle. During the diagnosis phase process mining techniques may be applied to generate suggestions for process type optimizations from analyzing process instance changes. These process optimizations can be then applied to process type schemes and, in turn, be propagated to already running process instances. In particular, the process instances which have indicated the respective process optimization can be adequately treated using the approach presented in this thesis.

An important aspect is the implementation of flexible PMS. As indicated in Chapter 7, we have already elaborated propositions for the efficient storage and management of process type and process instance schemes [70, 71]. Another question concerns locking of process instances for compliance checks and migration to the changed process type schema but without blocking their execution too long.

This thesis focused on the application–neutral part of change management in process management system. This application–neutral is fundamental to cover practical relevant scenarios which can be (in most cases) automatically handled. Nevertheless, there are some scenarios where also *application–dependent* information is desired. As one example consider semantical conflicts between process type and process instance changes (like medical incompatibility of drugs, cf. Figure 5.7, Section 5.3). To be able to deal with such semantical conflicts it becomes necessary to deposit application information within the system, e.g., "*drugA* is incompatible with *drugB*". If any activity related to one of these drugs is then inserted at process type or process instance level we have to check whether an activity related to the other drug is inserted on the other level or not. If so, we have, at minimum, to report this situation to the user. In this context, we must also deal with the question how to store and evaluate the necessary

semantical information within the system (e.g., within ontologies or case–based reasoning systems [132]). Altogether, incorporating more semantics into PAIS constitutes a promising future research topic.

Another important issue refers to the transferability of the presented results to process management systems with distributed process control; i.e., the partitioning of process schemes and the allocation of these partitions to different process server [15].

This work addresses the handling of process type and process instance changes. However, there are other components within a process management system which may be subject of changes as well like the organizational model or the application components (cf. Figure 8.2). Of course, it is quite important to adequately cope with changes of these components as well and to find complete solutions. Again, it is crucial to deal with the interplay and interactions between changes of all components. For example, organizational changes (e.g., split of an organizational unit, fusion of user roles) may require adaptations of actor assignments at process type level [93, 137]. To be able to manage changes at all levels of a PMS is very challenging and will be a hot topic in future research.

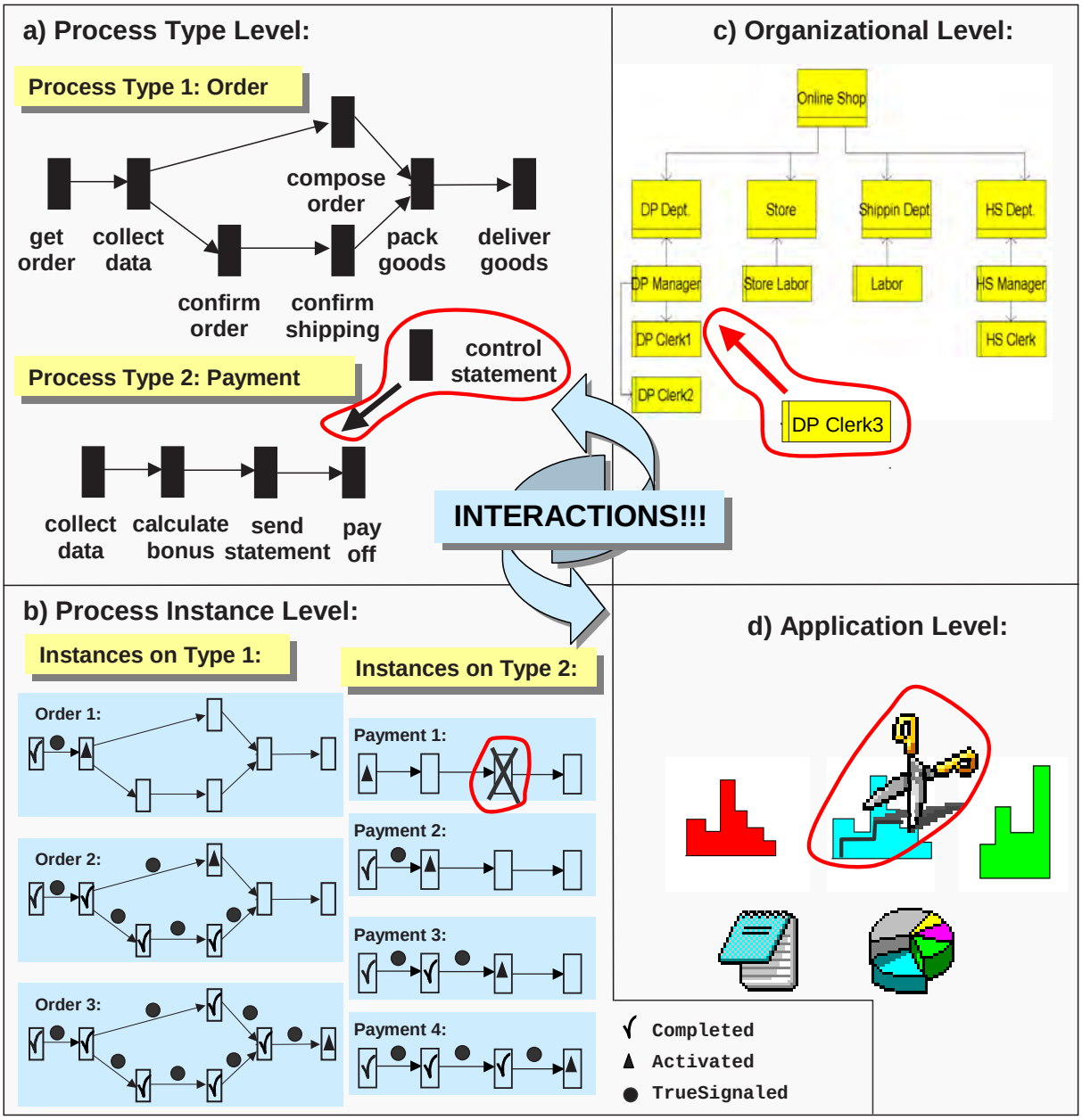


Figure 8.2: Changing Process Management Components – The Big Picture

Bibliography

[1] A. Agostini and G. De Michelis. Improving flexibility of workflow management systems. In *Proc. Int'l Conf. on Business Process Management (BPM'00)*, pages 218–234, 2000.

[2] A. Agostini and G. de Michelis. A light workflow management system using simple process models. In *Int'l Journal of Collaborative Comp.* [61]. 335-363.

[3] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services – Concepts, Architectures and Applications.* Springer, 2004.

[4] J. Andany, M. Leonard, and C. Palisser. Management of schema evolution in databases. In *Proc. Int'l Conf. on Very Large Databases (VLDB'91)*, pages 161–170, Barcelona, September 1991.

[5] T. Andrews, F. Curbera, H. Dholakia, and Y. Goland et al. *BPELWS - Business Process Execution Language for Web Services – Version 1.1.*, 2003. BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, Siebel Systems.

[6] E. Badouel and J. Oliver. Reconfigurable nets: a class of high level petri nets supporting dynamic changes with workflow systems. Technical Report PI1163, Inria, 1998.

[7] B.R, Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, 1992.

[8] K. Baina, S. Tata, and K. Benali. A model for process service interaction. In v.d. Aalst et al. [122], pages 261–275.

[9] S. Bandinelli, A. Fugetta, and C. Ghezzi. Software process model evolution in the SPADE environment. *IEEE Transactions on Software Engineering*, 19(12):1128–1144, December 1993.

[10] S. Bassil, M. Benyoucef, R. Keller, and P. Kropf. Addressing dynamism in e-negotiations by workflow management systems. In *Proc. Workshop on Negotiations in e-Markets – Beyond Price Discovery (DEXA'02)*, September 2002.

[11] S. Bassil, R. Keller, and P. Kropf. A workflow–oriented system architecture for the management of container transportation. In Desel et al. [35], pages 116–131.

[12] T. Basten. Branching bisimilarity is an equivalence indeed!. *Information Processing Letters*, 58(3):141–147, 1996.

[13] T. Basten and W.M.P. v.d. Aalst. Inheritance of behavior. *Journal of Logic and Algebraic Programming*, 47(2):47–145, 2001.

[14] C. Batini, M. Lenzerini, and S.B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[15] T. Bauer. *Efficient Realization of Enterprise–Wide Workflow Management Systems.* PhD thesis, University of Ulm, 2001. (in German).

[16] T. Bauer and P. Dadam. Efficient distributed workflow management based on variable server assignments. In *Proc. Int'l Conf on Advanced Information Systems Engineering (CAiSE'00)*, pages 94–109, Stockholm, June 2000.

[17] C. Beckstein and J. Klausner. A planning framework for workflow management. In *Proc. Workshop Intelligent Workflow and Process Management*, Stockholm, 1999.

[18] P. Berry and K.L Myers. Adaptive process management: An al perspective. In *Proc. Workshop Towards Adaptive Workflow Systems (CSCW'98)*, Seattle, 1998.

[19] T. Beuter. *Workflow Management for Product Development Processes.* PhD thesis, University of Ulm, 2002. (in German).

[20] T. Beuter, P. Dadam, and P. Schneider. The WEP model: Adequate workflow-management for engineering processes. In *Proc. Europ. Conf. on Concurrent Engineering*, Erlangen, April 1998.

[21] P. Bichler, G. Preuner, and M. Schrefl. Workflow transparency. In *Proc. Int'l Conf on Advanced Information Systems Engineering (CAiSE'97)*, pages 423–436, Barcelona, June 1997.

[22] D. Bogia and S. Kaplan. Flexibility and control for dynamic workflows in the wOrlds environment. In *Proc. Int'l Conference on Organisational Computing Systems (COOCS'95)*, pages 148–161, 1995.

[23] Y. Breitbart, A. Deacon, H.–J. Schek, A. Sheth, and G. Weikum. Merging application–centric and data–centric approaches to support transaction–oriented multi–system workflows. *ACM SIGMOD Record*, 22(3):23–30, 1993.

[24] H. Bunke and X. Jiang. Graph matching and similarity. In H. N. Teodorescu, D. Mlynek, A. Kandel, and H. J. Zimmermann, editors, *Proc. Intelligent Systems and Interfaces.* Kluwer Academic Publishers, 2000.

[25] F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM TODS*, 24(3):405–451, 1999.

[26] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow evolution. *Data and Knowledge Engineering*, 24(3):211–238, 1998.

[27] D. Chiu, Q. Li, and K. Karlapalem. Web interface-driven cooperative exception handling in ADOME. *Informations Systems*, 26(2):93–120, 2001.

[28] S. Conrad. *Federated Database Systems – Concepts of Data Integration.* Springer, 1997. (in German).

[29] F. Curbera, R. Khalaf, F. Leymann, and S. Weerawarana. Exception handling in the BPEL4WS language. In v.d. Aalst et al. [122], pages 321–335.

[30] P. Dadam. *Distributed Databases and Client/Server systems.* Springer, 1996. (in German).

[31] P. Dadam and M. Reichert. Towards a new dimension in clinical information processing. In *Proc. MIE2000/GMDS 2000*, pages 295–301, Hannover, Sept. 2000.

[32] P. Dadam, M. Reichert, and K. Kuhn. Clinical workflows – the killer application for process-oriented information systems? In *Proc. Int'l Conf. on Business Information Systems (BIS'00)*, pages 36–59, Poznan, Poland, 2000.

[33] W. Deiters, T. Goesmann, K. Just-Hahn, T. Löffeler, and R. Rolles. Support for exception handling through workflow management systems. In *Proc. Workshop Towards Adaptive Workflow Systems (CSCW'98)*, Seattle, November 1998.

[34] W. Deiters and V. Gruhn. The FUNSOFT net approach to software process management. *Int'l Journal of Software Engineering and Knowledge Engineering*, 4(2):229–256, 1994.

[35] J. Desel, B. Pernici, and M. Weske, editors. *Business Process Management.*, LNCS 3080, Potsdam, Germany, June 2004.

[36] M. Dumas and A. H. M. ter Hofstede. UML activity diagrams as a workflow specification language. In *Proc. UML '01*, Toronto, Canada, 2001.

[37] D. Edmond and A.H.M. ter Hofstede. A reflective infrastructure for workflow adaptability. *Data and Knowledge Engineering*, 34(3):271–304, 2000.

[38] C. Ellis and K. Keddara. A workflow change is a workflow. In *Proc. Int'l Conf. on Business Process Management (BPM'00)*, pages 516–534, 2000.

[39] C.A. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In *Proc. ACM Conf. on Organizational Computing Systems (COOCS'95)*, pages 10–21, Milpitas, CA, August 1995.

[40] C.A. Ellis and C. Maltzahn. The Chautauqua workflow system. In *Proc. Int'l Conf. on System Science*, Maui, Hawaii, 1997.

[41] A. Fent, H. Reiter, and B. Freitag. Design for change: Evolving workflow specifications in ULTRAflow. In *Proc. Int'l Conf. on Advanced Information Systems Engineering (CAISE'02)*, pages 516–534, May 2002.

[42] FileNet Corporation. *FileNet Business Process Manager.*, 2003. (www.filenet.com).

[43] H. Frank and J. Eder. Equivalence transformations on statecharts. In *Proc. Int'l Conf. on Software Engineering and Knowledge Engineering (SEKE'00)*, pages 150–158, Chicago, July 2000.

[44] Gartner Group. Why e-business craves workflow technology. Technical Report T-09-4929, Gartner Group, December 1999.

[45] V. Guth and A. Oberweis. Delta analysis of petri net based models for business processes. In *Proc. Applied Informatics*, pages 23–32, 1997.

[46] M. Hammer and J. Champy. *Reengineering the Corporation.* Harper Collins, 1993.

[47] M. Hammori, J. Herbst, and N. Kleiner. Interactive workflow mining. In Desel et al. [35], pages 211–226.

[48] Y. Han. *Software Infrastructure for Configurable Workflow Systems.* PhD thesis, TU Berlin, 1995.

[49] P. Heimann, G. Joeris, C. Krapp, and B. Westfechtel. DYNAMITE: Dynamic task nets for software process management. In *Proc. Int'l Conf. Software Engineering (ICSE'06)*, pages 331–341, Berlin, March 1996.

[50] J. Herbst. An inductive approach to adaptive workflow systems. In *Proc. Workshop Towards Adaptive Workflow Systems (CSCW'98)*, Seattle, November 1998.

[51] IBM. *IBM WebSphere MQ Workflow V3.5 Release – Advanced production workflow with MQ and portal integration*, 2004. http://www-306.ibm.com/common/ssi/rep_ca/4/897/ENUS204-044/ENUS204-044.PDF.

[52] S. Jablonski, M. Böhm, and W. Schulze. *Workflow Management Development of Applications and Systems.* dpunkt, 1999. (in German).

[53] G. Joeris. Defining flexible workflow execution behaviors. In *Proc. GI-Workshop, Enterprise-wide and Cross-enterprise Workflow-Management (Informatik'99)*, pages 49–55, October 1999.

[54] G. Joeris and O. Herzog. Managing evolving workflow specifications. In *Proc. Int'l Conf. on Cooperative Information Systems (CoopIS'98)*, pages 310–321, New York City, August 1998.

[55] B. Joos, R.M. Katzsch, A. Meier, and C. Wernet. A practical comparison of three workflow management systems: Workflow, Staffware und InConcert. *Theorie und Praxis der Wirtschaftsinformatik*, 193:81–103, 1997. (in German).

[56] D. Jungnickel. *Graphs, Networks and Algorithms.* BI Wissenschaftsverlag, 1994. (in German).

[57] M. Kamath, G. Alonso, R. Günthör, and C. Mohan. Providing high availability in very large workflow management systems. In *Proc. Int'l Conf. on Extending Database Technology (EDBT'96)*, pages 427–442, Avignon, March 1996.

[58] G. Kappel. Reorganizing object behavior by behavior composition – coping with evolving requirements in office systems. In *Proc. Datenbanksysteme in Büro, Technik und Wissenschaft (BTW'91)*, pages 446–453, Kaiserslautern, March 1991.

[59] B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows.* PhD thesis, Queensland University of Technology, Brisbane, 2002. (available via *http://www.tm.tue.nl/it/research/patterns)*.

[60] B. Kiepuszewski, A.H.M. ter Hofstede, and C.J. Bussler. On structured workflow modelling. In *Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE'00)*, pages 431–445, 2000.

[61] M. Klein, C. Dellarocas, A. Bernstein, and (Eds.). Special issue on adaptive workflow systems. *Int'l Journal of Collaborative Comp.*, 9(3-4), 2000.

[62] N. Kleiner. Supporting usage–centered workflow design: Why and how?. In Desel et al. [35], pages 227–243.

[63] M. Kloppmann and G. Pfau. *WebSphere Application Server Enterprise Process Choreographer: Concepts and Architecture.*, 2002. http://www-136.ibm.com/developerworks/websphere/.

[64] J. Köbler, U. Schöning, and J. Toran. Graph isomorphism is low for PP. *Journal of Computational Complexity*, 2:301–330, 1992.

[65] K. Kochut, J. Arnold, A. Sheth, J. Miller, E. Kraemer, B. Arpinar, and J. Cardoso. IntelliGEN: A distributed workflow system for discovering protein-protein interactions. *Distributed and Parallel Databases*, 13(1):43–72, 2003.

[66] M. Kradolfer. *A Workflow Metamodel Supporting Dynamic, Reuse–Based Model Evolution.* PhD thesis, University of Zurich, 2000.

[67] M. Kradolfer and A. Geppert. Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In *Proc. Int'l Conf. in Cooperative Information Systems (CoopIS'99)*, pages 104–114, Edinburgh, September 1999.

[68] H.T. Kung and J.T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[69] J.A. Larson, S.B. Navathe, and R. Elmasri. A theory of attribute equivalence in databases with application to schema integration. *IEEE Transactions on Software Engineering*, 15(4):449–463, 1989.

[70] M. Lauer. Implementation aspects for adaptive process management systems. Master's thesis, University of Ulm, Computer Science Faculty, 2004. (to appear, in German).

[71] M. Lauer, S. Rinderle, and M. Reichert. Representation of schema and instance objects in adaptive process management systems. In *Proc. Workshop on Business Process Oriented Architectures (Informatik'04)*, LNI P-51, pages 555–560, Ulm, Germany, Sept. 2004. (in German).

[72] F. Leymann. Supporting business transactions via partial recovery in workflow management systems. In *Proc. Datenbanksysteme in Büro, Technik und Wissenschaft (BTW'95)*, pages 51–70, Dresden, March 1995.

[73] F. Leymann and W. Altenhuber. Managing business processes as an information ressource. *IBM Systems Journal*, 33(2):326–348, 1994.

[74] F. Leymann and D. Roller. *Production Workflow.* Prentice Hall, 2000.

[75] C. Liu and R. Conradi. Automatic replanning of task networks for process model evolution. In *Proc. European Software Engineering Conference*, pages 434–450, Garmisch-Partenkirchen, Germany, Sept. 1993.

[76] P. Mangan and S. Sadiq. A constraint specification approach to building flexible workflows. *Journal of Research and Practice in Information Technology*, 35(1):21–39, 2002.

[77] J.E. Mann. Workflow and EAI. *EAI Journal*, September/October:49–53, 1999.

[78] U. Martschat. Comparison and evaluation of production workflow-management-systems. Master's thesis, University of Ulm, Computer Science Faculty, 2001. (in German).

[79] Microsoft. *BizTalk Server*, 2004. http://www.microsoft.com/biztalk/.

[80] R. Müller. *Event-Oriented Dynamic Adaptation of Workflows.* PhD thesis, University of Leipzig, Germany, 2002.

[81] R. Müller, U. Greiner, and E. Rahm. AgentWork: A workflow system supporting rule–based workflow adaptation. *Data & Knowledge Engineering*, 51(2):223–256, 2004.

[82] P. Muth, J. Weissenfels, M. Gillmann, and G. Weikum. Workflow history management in virtual enterprises using a light-weight workflow management system. In *Proc. Int'l Workshop on Research Issues in Data Engineering (RIDE'99)*, pages 148–155, March 1999.

[83] A. Oberweis. *Modeling and Execution of Workflows with Petri Nets.* Teubner, 1996. (in German).

[84] G. Piccinelli and S. Lane Williams. Workflow: A language for composing web services. In v.d. Aalst et al. [122], pages 13–24.

[85] P. McBrien; A. Poulovassilis. A formalism of semantic schema integration. *Information Systems*, 23(5):390–334, 1998.

[86] The AristaFlow Project. www.aristaflow.de, 2004.

[87] M. Reichert. *Dynamic Changes in Workflow-Management-Systems.* PhD thesis, University of Ulm, Computer Science Faculty, 2000. (in German).

[88] M. Reichert and P. Dadam. ADEPT$_{flex}$ - supporting dynamic changes of workflows without losing control. *JIIS*, 10(2):93–129, 1998.

[89] M. Reichert and S. Rinderle. Change authorizations in adaptive workflow management systems. In *Proc. Conf. Sichere Geschäftsprozesse*, pages 30–42, St. Leon-Rot, Germany, September 2002. (in German).

[90] M. Reichert, S. Rinderle, and P. Dadam. ADEPT workflow management system: Flexible support for enterprise-wide business processes (tool presentation). In v.d. Aalst et al. [122], pages 370–379.

[91] M. Reichert, S. Rinderle, and P. Dadam. On the common support of workflow type and instance changes under correctness constraints. In *Proc. Int'l Conf. on Cooperative Information Systems (CoopIS'03)*, LNCS 2888, pages 407–425, Catania, Italy, November 2003.

[92] M. Reichert, S. Rinderle, and P. Dadam. On the modeling of correct service flows with BPEL4WS. In *Proc. Conf. on Development Methods for Information Systems and their Application (EMISA'04)*, pages 117–128, Luxembourg, October 2004.

[93] M. Reichert, U. Wiedemuth-Catrinescu, and S. Rinderle. Evolution of access control in information systems. In *Proc. Conf. Elektronische Geschäftsprozesse (EGP'04)*, pages 100–114, Klagenfurt, 2004. (in German).

[94] W. Reisig. *Petri Nets: An Introduction.* Springer, 1991. (in German).

[95] S. Rinderle and P. Dadam. Schema evolution in workflow management systems. *Informatik Spektrum*, 26(1):17–19, 2003. (in German).

[96] S. Rinderle, M. Reichert, and P. Dadam. Evaluation of correctness criteria for dynamic workflow changes. In v.d. Aalst et al. [122], pages 41–57.

[97] S. Rinderle, M. Reichert, and P. Dadam. On dealing with semantically conflicting business process changes. Technical Report UIB-2003-04, University of Ulm, June 2003.

[98] S. Rinderle, M. Reichert, and P. Dadam. Correctness criteria for dynamic changes in workflow systems – a survey. *Data and Knowledge Engineering, Special Issue on Advances in Business Process Management*, 50(1):9–34, 2004.

[99] S. Rinderle, M. Reichert, and P. Dadam. Disjoint and overlapping process changes: Challenges, solutions, applications. In *Proc. Int'l Conf. on Cooperative Information Systems (CoopIS'04)*, pages 101–120, Larnaca, Cyprus, October 2004.

[100] S. Rinderle, M. Reichert, and P. Dadam. Flexible support of team processes by adaptive workflow systems. *Distributed and Parallel Databases*, 16(1):91–116, 2004.

[101] S. Rinderle, M. Reichert, and P. Dadam. On dealing with structural conflicts between process type and instance changes. In Desel et al. [35], pages 274–289.

[102] B. Rosenstengel and U. Winand. *Petri Nets: An Application–Oriented Introduction.* Vieweg, 1991. (in German).

[103] S. Sadiq. Handling dynamic schema changes in workflow processes. In *Proc. 11th Australian Database Conference*, January 2000.

[104] S. Sadiq, O. Marjanovic, and M. Orlowska. Managing change and time in dynamic workflow processes. *IJCIS*, 9(1&2):93–116, 2000.

[105] S. Sadiq and M. Orlowska. Dynamic modification of workflows. Technical Report 442, University of Queensland, Brisbane, Australia, October 1998.

[106] S. Sadiq and M. Orlowska. Architectural considerations in systems supporting dynamic workflow modification. In *Proc. Workshop Software Architectures for Business Process Management*, Heidelberg, June 1999.

[107] S. Sadiq, W. Sadiq, and M. Orlowska. Pockets of flexibility in workflow specifications. In *Proc. Int'l Entity–Relationship Conf. (ER'01)*, pages 513–526, Yokohama, 2001.

[108] SAP. *Webflow Engine Driving Continous Improvement In Business Processes.*, 2003. http://www.sap.com/solutions/netweaver/brochures/.

[109] SERprocess. (www.ser.com).

[110] R. Siebert, T. Kindler, and T. Soyez. Integrated workflow and telecooperation support for the german government. In *Proc. Symposium on Applied Computing (SAC'1997)*, pages 177–179, San Jose, March 1997.

[111] Staffware. *Staffware Process Suite Brochure*, 2004. http://www.staffware.com/downloads/.

[112] D.M. Strong and S.M. Miller. Exceptions and exception handling in computerized information processes. *ACM–TOIS*, 13(2):206–233, 1995.

[113] R. v. Glabbeek and U. Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37(4–5):229–327, 2001.

[114] W.M.P. v.d. Aalst. *Timed Coloured Petri Nets and their Application to Logistics.* PhD thesis, Eindhoven University of Technology, 1992.

[115] W.M.P. v.d. Aalst. Exterminating the dynamic change bug: A concrete approach to support worfklow change. *Information Systems Frontiers*, 3(3):297–317, 2001.

[116] W.M.P. v.d. Aalst. Inheritance of business processes: A journey visiting four notorious problems. In *Proc. Petri Net Technology for Communication Based Systems*, LNCS 2472, pages 383–408, 2003.

[117] W.M.P. v.d. Aalst and T. Basten. Identifying commonalities and differences in object life cycles using behavorial inheritance. In *Proc. Int'l Conf. on Application and Theory of Petri Nets (ICATPN'01)*, LNCS 2075, pages 32 – 52, Newcastle, UK, 2001.

[118] W.M.P v.d. Aalst and T. Basten. Inheritance of workflows: An approach to tackling problems related to change. *Theoret. Comp. Science*, 270(1-2):125–203, 2002.

[119] W.M.P. v.d. Aalst and P. Berens. Beyond workflow management: Product-driven case handling. In *Proc. Conference On Supporting Group Work*, pages 42–51, New York, 2001.

[120] W.M.P. v.d. Aalst and S. Jablonski. Dealing with workflow change: Identification of issues an solutions. *Int'l Journal of Comp. Systems, Science and Engineering*, 15(5):267–276, 2000.

[121] W.M.P v.d. Aalst and M. Song. Mining social networks. uncovering interaction patterns in business processes. In Desel et al. [35], pages 244–260.

[122] W.M.P. v.d. Aalst, A.H.M. ter Hofstede, and M. Weske, editors. *Business Process Management.*, LNCS 2678, Eindhoven, The Netherlands, June 2003.

[123] W.M.P v.d Aalst, A.H.M ter Hofstede, and M. Weske. Business process management: A survey. In v.d. Aalst et al. [122], pages 1–12.

[124] W.M.P v.d. Aalst, B.F van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M Weijters. Workflow mining: A survey of issues and approaches. *Data and Knowledge Engineering*, 27(2):237–267, 2003.

[125] W.M.P. v.d. Aalst and K. van Hee. *Workflow Management.* MIT Press, 2002.

[126] W.M.P v.d. Aalst, M. Weske, and D. Grünbauer. Case handling: A new paradigm for business process support. *Data & Knowledge Engineering*, 2004. (to appear).

[127] W.M.P v.d. Aalst, M. Weske, and G. Wirtz. Advanced topics in workflow management: Issues, requirements, and solutions. *Int'l Journal of Integrated Design and Process Science*, 7(3), 2003.

[128] E. Verbeek. *Verification of WF–Nets.* PhD thesis, Technical University of Eindhoven, 2004.

[129] H.M.W. Verbeek and W.M.P. v.d. Aalst. Woflan 2.0 A petri-net-based workflow diagnosis tool. In *Proc. Int'l Conf. on Application and Theory of Petri Nets (ICATPN'00)*, LNCS 1825, pages 455–464, Aarhus, June 2000.

[130] G. Vossen and J. Becker. *Business Process Modeling and Workflow Management.* Thomson Publisher, 1996. (in German).

[131] J. Wäsch and W. Klas. History merging as a mechanism for concurrency control in cooperative environments. In *Proc. Int'l Workshop on Research Issues in Data Engineering (RIDE–NDS'96)*, pages 76–85, New Orleans, February 1996.

[132] B. Weber, W. Werner, and R. Breu. CCBR–enabled adaptive workflow management. In *Proc. European Conf. on Case-Based Reasoning (ECCBR'04)*, LNCS 3155, pages 434–448, Madrid, 2004.

[133] M. Weber, T. Illmann, and A. Schmidt. Webflow: Decentralized workflow management in the world wide web. In *Proc. Int'l Conf. on Applied Informatics (AI'98)*, February 1998.

[134] M. Weber, G. Patsch, A. Scheller-Huoy, J. Schweitzer, and G. Schneider. Flexible real-time meeting support for workflow management systems. In *Proc. Hawaii Int'l Conf. on System Sciences (HICSS'97)*, Maui, Hawaii, January 1997.

[135] M. Weske. Workflow management systems: Formal foundation, conceptual design, implementation aspects. University of Münster, Germany, 2000. Habilitation Thesis.

[136] M. Weske. Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In *Proc. Hawaii International Conference on System Sciences (HICSS-34)*, 2001.

[137] U. Wiedemuth-Catrinescu. Evolution of organizational models in workflow management systems. Master's thesis, University of Ulm, Computer Science Faculty, 2002. (in German).

[138] D.E Wilkins, K.L. Myers, J.D Lowrance, and L.P Wesley. Planning and reacting in uncertain and dynamic environments. *Experimental and Theoretical AI*, 7(1):197–227, 1995.

[139] Workflow Management Coalition. Terminology & glossary. Technical Report WFMC-TC-1011, WfMC, 1999.

Appendix A

Abbreviations

Table A.1: List of Abbreviations

Abbreviation	**Meaning**	**Chapter**
CP	Changing the Past Problem	cf. Chapter 2
DMBS	Database Management System	cf. Chapter 2
DS	Dangling States Problem	cf. Chapter 2
GCD	Greates Common Divisor	cf. Chapter 6
GI	Graph Isomorphism	cf. Chapter 6
ICN	Information Control Network	cf. Chapter 2
LT	Loop Tolerance Problem	cf. Chapter 2
MCS	Minimal Critical Specification	cf. Chapter 2
NM	Net Model	cf. Chapter 2
OC	Order Changing Problem	cf. Chapter 2
PAIS	Process–Aware Information Systems	cf. Chapter 1
PI	Parallel Insertion Problem	cf. Chapter 2
PMS	Process Management System	cf. Chapter 1
SCOC	Synthetic Cut Over Change	cf. Chapter 2
SM	Sequential Model	cf. Chapter 2
WSM Net	Well–Structured Marking Net	cf. Chapter 3

Appendix B

Definitions and Functions

Table B.1: A Selection of Important Functions Based On WSM Nets [87]

Let S = (N, D, ...) be a correct WSM Net (cf. Definitions 1 and 2).

`c_succ(S,n)` / `c_pred(S,n)`	set of all *direct* successors / predecessors of activity `n` considering only edges `e` ∈ `CtrlE` in `S`
`c_succ`*`(S,n)` / `c_pred`*`(S,n)`	set of all *direct* or *indirect* successors / predecessors of activity `n` considering only edges `e` ∈ `CtrlE` in `S`
`succ(S,n)` / `pred(S,n)`	set of all *direct* successors / predecessors of activity `n` referring to edges `e` ∈ (`CtrlE` ∪ `SyncE`) in `S`
`succ`*`(S,n)` / `pred`*`(S,n)`	set of all *direct* and *indirect* successors / predecessors of activity `n` referring to edges `e` ∈ (`CtrlE` ∪ `SyncE`) in `S` `succ`*`(S,n)` = {n* ∈ N | n* ∈ `succ`(S,n) ∨ (∃ n** ∈ `succ`(S,n): n* ∈ `succ`*(S,n**))}
$MinBlock(S, N^*)$	minimal control block in S embracing all activities contained in N^* ($N^* \subseteq N$)
$BranchNodes(S, n_1, n_2)$	determines split and join activity nodes (s, j) of the smallest branching block embracing n_1 and n_2 in S
$join^S(s)$ $(split^S(j))$	With split (join) node s (j) associated join (split) node in WSM Net S.
$endloop^S(L_S)$ $(startloop^S(L_E))$	With loop start (end) node L_S (L_E) associated loop end (start) node in WSM Net S.

Rules 1 (ADEPT Marking and Execution Rules) *Let S = (N, D, NT, CtrlE, SyncE, LoopE, DataE, DP, EC) be a correct WSM Net (cf. Definitions 1 and 2) and let* $I = (S, \Delta_I, M^{S_I}, Val^{S_I}, \Pi_I^{S_I})$ *be a process instance running according to S with node and edge markings* $M^{S_I} = (NS^{S_I}, ES^{S_I})$ *(cf. Definition 3).*

Execution Rule E_1 (Executing Activity Nodes)

$\forall n \in N$ *with* $NT(N) \in \{$`EndFlow, Activity, AndJoin, AndSplit, XOrSplit`$\} \wedge$
$NS^{S_I}(n) =$ `NotActivated`*:*

$NS^{S_I}(n) =$ `NotActivated` *can be changed into* $NS^{S_I} =$ `Activated` $\Longleftrightarrow$
($\forall cE \in CtrlE$ *with* $cE = (x, n)$ $(x \in N)$: $ES^{S_I}(e) =$ `TrueSignaled`*)* $\wedge$
($\forall sE \in SyncE$ *with* $sE = (y, n)$ $(y \in N)$: $ES^{S_I}(sE) \in \{$`TrueSignaled, FalseSignaled`$\}$*)*

Execution Rule E_2 (Executing `XOr`–Joins)

$\forall n \in N$ *with* $NT(N) =$ `XOrJoin` $\wedge$ $NS^{S_I}(n) =$ `NotActivated`*:*

$NS^{S_I}(n) =$ `NotActivated` *can be changed into* $NS^{S_I} =$ `Activated` $\Longleftrightarrow$
($\dot{\exists} cE \in CtrlE$ *with* $cE = (x, n)$ $(x \in N)$: $ES^{S_I}(e) =$ `TrueSignaled`*)* $\wedge$
($\forall sE \in SyncE$ *with* $sE = (y, n)$ $(y \in N)$: $ES^{S_I}(sE) \in \{$`TrueSignaled, FalseSignaled`$\}$*)*

Execution Rule E_3 (Executing Loop Start Nodes)

$\forall n \in N$ *with* $NT(N) =$ `StartLoop` $\wedge$ $NS^{S_I}(n) =$ `NotActivated`*:*

$NS^{S_I}(n) =$ `NotActivated` *can be changed into* $NS^{S_I} =$ `Activated` $\Longleftrightarrow$
($\forall cE \in CtrlE$ *with* $cE = (x, n)$ $(x \in N)$: $ES^{S_I}(e) =$ `TrueSignaled`*)* $\wedge$
($\forall sE \in SyncE$ *with* $sE = (y, n)$ $(y \in N)$: $ES^{S_I}(sE) \in \{$`TrueSignaled, FalseSignaled`$\}$*)* $\wedge$
($lE \in LoopE$ *with* $lE = (L_E, n)$ $(L_E \in N, NT(L_E) =$ `EndLoop`*):*
$ES^{S_I}(lE) \in \{$`NotSignaled, TrueSignaled` $\}$*)*

Marking Rule M_1 (Completing Activities)

$\forall n \in N$ *with* $NT(N) \in \{$`StartFlow, EndFlow, Activity, AndJoin, AndSplit, XOrJoin`$\} \wedge$
$NS^{S_I}(n) =$ `Running`*:*

$NS^{S_I}(n) =$ `Running` *can be changed into* $NS^{S_I} =$ `Completed` $\Longleftrightarrow$
n is successfully terminated. Then:

($NT(n) =$ `EndFlow` $\Longrightarrow$ *I is terminated)* $\vee$
($\forall e \in CtrlE \cup SyncE$ *with* $e = (n, x)$ $(x \in N)$: $ES^{S_I} :=$ `TrueSignaled`*)*

Marking Rule M_2 (Completing XOr–Splits)

$\forall n \in N$ *with* $NT(N) =$ `XOrSplit`*, decision parameter* DP^n*, and* $NS^{S_I}(n) =$ `Running`*:*

$NS^{S_I}(n) =$ `Running` *can be changed into* $NS^{S_I} =$ `Completed` $\Longleftrightarrow$
n is successfully terminated. Then:

$(e_{choice} \in CtrlE$ *with* $e_{choice} = (n, x)$ $(x \in N) \wedge SC^{e_{choice}} = DP^n$*:* $ES^{S_I}(e_{choice}) :=$ `TrueSignaled`$) \wedge$
$(e \in CtrlE$ *with* $e = (n, y)$ $(y \in N) \wedge SC^e \neq DP^n$*:* $ES^{S_I} :=$ `FalseSignaled` $\wedge$
$\forall z \in (\{y\} \cup c\_succ^*(y)) \cap c\_pred^*(join^S(n))$*:*
$NS^{S_I}(z) =$ `Skipped` *and* $\forall\ e_{false} = (z, t)$ $(t \in N)$*:* $ES^{S_I}(e_{false})$*:=* `FalseSignaled`$) \wedge$
$(\forall sE \in SyncE$ *with* $sE = (n, w)$ $(w \in N)$*:* $ES^{S_I} :=$ `TrueSignaled`$)$

Marking Rule M_3 (Completing EndLoops)

$\forall L_E \in N$ *with* $NT(N) =$ `EndLoop`*,* $lE \in LoopE$ *with* $lE = (L_E, L_S, lC)$ $(L_S = startloop^S(L_E))$
$\wedge\ NS^{S_I}(n) =$ `Running`*:*

$NS^{S_I}(L_E) =$ `Running` *can be changed into* $NS^{L_E} =$ `Completed` $\Longleftrightarrow$
L_E *is successfully terminated. Then:*

$(lC =$ `True` $\Longrightarrow$
$\forall n \in \{L_S, L_E\} \cup (c\_succ^*(L_S) \cap c\_pred^*(L_E))$*:*
$NS^{S_I}(n)$*:=*`NotActivated` *and* $\forall e \in CtrlE \cup SyncE \cup Loop$ *with* $e = (n, x)$ $(x \in N)$*:*
ES^{S_I}*:=*`NotSignaled` $\wedge$ $ES^{S_I}(lE)$*:=*`TrueSignaled`$) \vee$
$lC =$ `False` $\Longrightarrow$
$(ES^{S_I}(lE)$*:=*`FalseSignaled`$) \wedge$
$(\forall e \in CtrlE \cup SyncE$ *with* $e = (L_E, x)$ $(x \in N)$*:* ES^{S_I}*:=*`TrueSignaled`$)$

Auxiliary Definition 2 (Assigning Change Operations to Change Types) *Let* Δ *be a change. Let further*

1. *Change be a set of change operations (cf. Tables 3.1, 3.2, and 3.3) with*

   $Change:=\{[serial|parallel|branch]InsertActivity(\ldots)$, $deleteActivity(\ldots)$, $[serial|parallel|branch]MoveActivitiy(\ldots)$, $insertSyncEdge(\ldots)$, $deleteSyncEdge(\ldots)$, $insertLoopEdge(\ldots)$, $deleteBlock(\ldots)$, $addDataElements(\ldots)$, $deleteDataElements(\ldots)$, $addDataEdges(\ldots)$, $deleteDataEdges(\ldots)$, $changeActivityAttributes(\ldots)$, $changeEdgeAttributes(\ldots)\}$[1]

   and

2. *OpType be a set of operation types with* $OpType:=\{ins\_Act$, del_Act, $move\_Act$, ins_Sync, del_Sync, ins_Loop, del_Loop, $data$, $attrChange\}$

3. *optype be a function which assigns to each change operation in Change its specific operation type in OpType. Formally:*

$$optype : Change \mapsto OpType$$

with

$$\begin{aligned}
&optype([serial|parallel|branch]MoveActivitiy(\ldots)) = ins_Act\\
&optype(deleteActivity(\ldots)) = del_Act\\
&optype([serial|parallel|branch]MoveActivitiy(\ldots)) = move_Act\\
&optype(insertSyncEdge(\ldots)) = ins_Sync\\
&optype(deleteSyncEdge(\ldots)) = del_Sync\\
&optype(addDataElements(\ldots)) = data\\
&optype(deleteDataElements(\ldots)) = data\\
&optype(addDataEdges(\ldots)) = data\\
&optype(deleteDataEdges(\ldots)) = data\\
&optyp(changeActivityAttributes(\ldots) = attrChange\\
&optyp(changeEgdeAttributes(\ldots) = attrChange
\end{aligned}$$

[1] One may wonder whether block operations are taken into account or not. Block operations are implicitely treated since we base projections onto purged change log Δ^{purged}. Within this logical view on Δ block operations are split into their single operations.

Appendix C

Proofs

C.1 Proof (Theorem 1)

At first, we repeat Lemma 1 since it is needed in order to prove Theorem 1. It states that all predecessors of a running or completed activity $\mathtt{n}^*$ must have one of the markings `COMPLETED` or `SKIPPED`.

Lemma 1 (Correctness Property for Process Instance Markings) *Let $I = (S, \Delta_I, M^{S_I}, \ldots)$ be a process instance based on a (correct) WSM Net $S = (N, D, \ldots)$. Marking M^{S_I} has been achieved by applying the ADEPT marking and execution rules (cf. Rules 1, Appendix B). Then for $M^{S_I} = (NS^{S_I}, ES^{S_I})$ the following condition holds:*

$$\forall n \in N \text{ with } NS^{S_I}(n) \in \{\texttt{Activated, Running, Completed, Skipped}\} \Longrightarrow$$
$$(\forall\, n^* \in pred^*(S, n)\text{: } NS(n^*) \in \{\texttt{Completed, Skipped}\})$$

We now have done all necessary prepartory work for proving Theorem 1:

Theorem 1 (Compliance Conditions When Inserting Activities) *Let $S = (N, D, NT, CtrlE, SyncE, LoopE, DataE)$ be a correct process type schema (represented by a WSM–Net) and I be a process instance on S with reduced execution history $\Pi^S_{I\,red}$ and with marking $M^S = (NS^S, ES^S)$. Assume further that change operation Δ transforms S into a correct process type schema $S' = (N', D', NT', CtrlE', SyncE', LoopE', DataE')$.*
Thereby Δ inserts an activity n_{insert} (with associated control and sync edges) into S, i.e., $\Delta = [serial|parallel|branch]InsertActivity(S, n_{insert}, \ldots)$ (cf. Table 3.2). Then:

I is compliant with $S' \Leftrightarrow$
$\forall\, n \in \{x \in N \mid n_{insert} \rightarrow x \in (CtrlE' \cup SyncE')\}$:
$NS(n) \in \{\texttt{NotActivated, Activated, Skipped}\} \vee$
n_{insert} is inserted into an already skipped branch of an XOR-branching

The proposition of Theorem 1 can be more formally described as follows:

I is compliant with S' $\Leftrightarrow B_1 \vee B_2 \vee B_3$ with
$B_1 \equiv$ [$\forall n \in$ `succ(S',`n_{insert}`)`:
NS(n) $\in$ {`NotActivated, Activated, Skipped`}]
$B_2 \equiv$ [$\forall$ n $\in$ `c_pred(S',` n_{insert}`)`: NS(n) = `Skipped`]
$B_3 \equiv$ [n_{insert} is inserted into a skipped, empty branch]

(The statement "n_{insert} is inserted into an already skipped branch" corresponds to $B_2 \vee B_3 \vee$ [$\forall n \in$ `c_succ(S',`n_{insert}`)`: NS(n) = `Skipped`] where the last term is already included by B_1.)

"$\Rightarrow$" I is compliant with S' $\Rightarrow B_1 \vee B_2 \vee B_3$
Proof by Contradiction, we show: $\neg(B_1 \vee B_2 \vee B_3) \Rightarrow$ I is not compliant with S'

Assumption: $\neg(B_1 \vee B_2 \vee B_3)$ holds
$\neg(B_1 \vee B_2 \vee B_3) \equiv \neg B_1 \wedge \neg B_2 \wedge \neg B_3$
$\equiv$ [$\exists n^* \in$ `succ`(S', n_{insert}): NS(n^*) $\in$ {`Running, Completed`} $\wedge$
[$\exists n^{**} \in$ c_pred(S', n_{insert}): NS(n^{**}) $\neq$ `Skipped`] $\wedge$
[n_{insert} is not inserted into a skipped, empty branch]
With $\neg B_1$ and Lemma 1 we obtain NS'(n_{insert}) $\in$ {`Completed, Skipped`}. Consequently, the marking NS(n_{insert}) must be `Skipped`. After re-evaluating the marking of the modified instance a newly inserted activity will be either marked as `Skipped` (insertion into a skipped branch) or as `NotActivated` or `Activated`.

Taking the above assumption, n_{insert} must therefore have been inserted into an already skipped branch of an XOR-branching with split node `s` and join node `j`. Because of $\neg B_3$ this branch cannot be empty. Based on this, it either follows that n_{insert} is not a direct successor of `s` – then $\forall$ n $\in$ `c_pred`(S', n_{insert}): NS(n) = `Skipped` – or n_{insert} is not a direct predecessor of `j` – $\forall$ n $\in$ `c_succ`(S', n_{insert}): NS(n) = `Skipped`. The first statement can not be true because of $\neg B_2$ and the latter because of $\neg B_1$. This is contradicting to our assumption. □

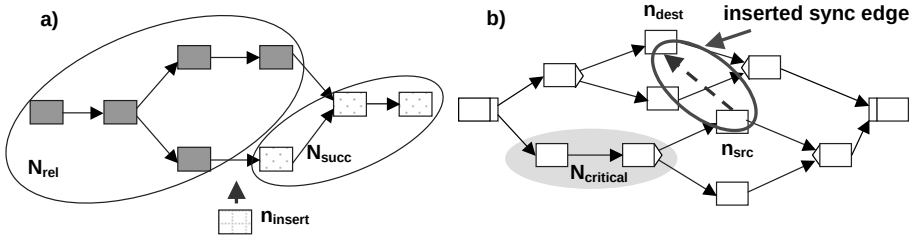


Figure C.1: Important Sets of a Process Schema Referring to n_{insert}

Let now statements C_1 and C_2 be as follows:
$C_1 \equiv$ [$\forall$ n $\in$ `succ(S',`n_{insert}`)`:
NS(n) $\in$ {`NotActivated, Activated, Skipped`}]
$C_2 \equiv$ [n_{insert} is inserted into a skipped branch of an XOR-branching]

"⇐": $C_1 \vee C_2 \Rightarrow$ I is compliant with S' (according to Criterion 7)
We first prove $C_1 \Rightarrow$ I is compliant with S'.
Assumption:

$C_1 \equiv$ [$\forall$ n $\in$ succ(S',n_{insert}):
NS(n) $\in$ {NotActivated, Activated, Skipped}]
$\Rightarrow \forall$ n $\in$ succ(S',n_{insert}):
$\nexists\ e_i \in \Pi^S_{I\,red}$ with $e_i \in$ {Start(n), End(n)}
$\Rightarrow \forall$ n $\in$ succ*(S',n_{insert}):
$\nexists\ e_i \in \Pi^S_{I\,red}$ with $e_i \in$ {Start(n), End(n)} ($\diamond$)

That means that the history $\Pi^S_{I\,red}$ contains no entry of a direct or indirect successor of n_{insert}. Furthermore, a re-evaluation of the instance marking results in

NS'(n_{insert}) $\in$ {NotActivated, Activated, Skipped}
$\Rightarrow\ \nexists\ e_i \in \Pi^S_{I\,red}$ with $e_i \in$ {Start(n_{insert}), End(n_{insert})} ($\diamond\diamond$)

We now show that I is compliant with S', i.e., the previous execution events $e_0, \ldots, e_k$ stored in $\Pi^S_{I\,red}$ can be applied to S' in the given order.

Let N_{rel} be the set of all activity nodes of N' which can be executed before n_{insert} is started(see Figure C.1a). So, N_{rel} contains all activity nodes positioned before or parallel to n_{insert}. Formally:

N_{rel} := pred*(S', n_{insert}) $\cup$
{n $\in$ N' | n $\notin$ pred* (S', n_{insert}) $\wedge$ n $\notin$ succ*(S', n_{insert})}

With ($\diamond$) and ($\diamond\diamond$) it follows:

$\forall\ e_i \in \Pi^S_{I\,red}$ with e_i = Start(n) $\vee$ e_i = End(n): n $\in N_{rel} \subseteq$ N

Thus all entries of $\Pi^S_{I\,red}$ have been written by activity nodes which are – in principle – executable before n_{insert} referring to S'. Since the subgraph of S induced by the node set N_{rel} (cf. Figure C.1a) is not affected by the insertion and therefore remains unchanged, $e_1, \ldots, e_k$ can be carried out on this subgraph in the given order and therefore on S' as well.
Referring to the second part [$C_2 \Rightarrow$ I is compliant with S'] it is clear that n_{insert} is inserted into a skipped branch, i.e., we obtain NS'(n_{insert}) = SKIPPED. Therefore n_{insert} has not yet written any entry into the execution history. Consequently, the previous execution history $\Pi^S_{I\,red}$ is producible on S' as well. □

C.2 Proof (Theorem 3)

Theorem 3 (Compliance Conditions When Inserting Control And Sync Edges) *Let S = (N, D, NT, CtrlE, SyncE, LoopE, DataE) be a correct process type schema (represented by a WSM–Net) and I be a process instance on S with reduced execution history $\Pi^S_{I\,red}$ and with marking $M^S = (NS^S, ES^S)$. Assume further that change operation Δ transforms S into a correct process type schema S' = (N', D', NT', CtrlE', SyncE', LoopE', DataE).*

(a) Δ inserts a control edge ctrlE = $n_{src} \rightarrow n_{dest}$ into S, i.e, Δ = addCtrlEdge(S, ctrlE). Then:

I is compliant with S' ⇔ NS(n_{dest}) ∈ {`NotActivated, Activated, Skipped`*}*

(b) Δ inserts a sync edge syncE = $\mathtt{n}_{src} \rightarrow \mathtt{n}_{dest}$ into S (n_{src} and n_{dest} ordered parallel so far), i.e., Δ = insertSyncEdge(S, syncE). Then:

I is compliant with S' ⇔

[NS(n_{dest}) ∈ {`NotActivated, Activated, Skipped`*}] ∨*
[NS(n_{src}) = `Completed` *∧ NS(n_{dest}) ∈ {*`Running, Completed`*} with*
∃e_i = `End`*(n_{src}), e_j =* `Start`*(n_{dest}) ∈ $\mathcal{H}_{red}$ ∧ $i < j$))] ∨*
[NS(n_{src}) = `Skipped` *∧ NS(n_{dest}) ∈ {*`Running, Completed`*}) with*
∀ n ∈ $N_{critical}$ with NS(n) ≠ `Skipped`*:*
∃e_i = `Start`*($\mathtt{n}_{dest}$), e_j =* `End`*(n) ∈ $\mathcal{H}_{red}$ with $j < i$),*
where $N_{critical}$ = (c_pred(S, n_{src}) ¬ c_pred*(S, n_{dest}))*
and c_pred(S, n)) denotes all direct/indirect predecessors of n in S concerning control edges]*

In the following, we first prove part (b) of Theorem 3 (insertion of sync edges into S) since part (a) (insertion of control edges) is less complex and can be proven in a similar way.

(b) Δ inserts a sync edge $\mathtt{n}_{src} \rightarrow \mathtt{n}_{dest}$ into S (n_{src} and n_{dest} ordered parallel so far).

First let
$A_1 \equiv$ [NS(n_{dest}) ∈ {`NotActivated, Activated, Skipped`}]
$A_2 \equiv$ [(NS(n_{src}) = `Completed` ∧ NS(n_{dest}) ∈ {`Running, Completed`})
with ∃$e_i, e_j \in \Pi^S_{I\,red}$: $i < j$ ∧ e_i = `End`(n_{src}), e_j = `Start`(n_{dest})]
$A_3 \equiv$ [(NS(n_{src}) = `Skipped` ∧ NS(n_{dest}) ∈ {`Running, Completed`})
with ∀$n \in N_{critical}$ with NS(n) ≠ `Skipped`:
∃$e_k, e_l \in \Pi^S_{I\,red}$: $l < k$ ∧ e_k = `Start`(n_{dest}), e_l = `End`(n)]
where $N_{critical}$ = (c_pred*(n_{src}) ¬ c_pred*(n_{dest})) (cf. Fig. C.1b)

The negation of A_1, A_2 and A_3 yields
$\neg A_1 \equiv$ [NS(n_{dest}) ∈ {`Running, Completed`}]
$\neg A_2 \equiv$ [NS(n_{src}) ≠ `Completed` ∨
NS(n_{dest}) ∈ {`NotActivated, Activated, Skipped`} ∨
∄$e_i, e_j \in \Pi^S_{I\,red}$: $i < j$ ∧ e_i = `End`(n_{src}), e_j = `Start`(n_{dest})]
$\neg A_3 \equiv$ [NS(n_{src}) ≠ `Skipped` ∨
NS(n_{dest}) ∈ {`NotActivated, Activated, Skipped`} ∨
∃$n \in N_{critical}$ with NS(n) ≠ `Skipped`:
∄$e_k, e_l \in \Pi^S_{I\,red}$: $l < k$ ∧ e_k = `Start`(n_{dest}), e_l = `End`(n)]

"⇒": I is compliant with S' ⇒ $A_1 \vee A_2 \vee A_3$
Proof by contradiction, we show:
¬($A_1 \vee A_2 \vee A_3$) ⇒ I is not compliant with S'

Assumption: $\neg(A_1 \vee A_2 \vee A_3)$ holds.

$$
\begin{aligned}
&\neg(A_1 \vee A_2 \vee A_3) \equiv \neg A_1 \wedge \neg A_2 \wedge \neg A_3 \equiv (\neg A_1 \wedge \neg A_2) \wedge \neg A_3 \\
&\equiv [(\mathrm{NS}(n_{dest}) \in \{\texttt{Running, Completed}\} \wedge \mathrm{NS}(n_{src}) \neq \texttt{Completed}) \\
&\qquad \vee (\mathrm{NS}(n_{dest}) \in \{\texttt{Running, Completed}\} \wedge \\
&\qquad\quad \nexists e_i, e_j \in \Pi^S_{I\,red}\colon i < j \wedge e_i = \texttt{End}(n_{src}),\ e_j = \texttt{Start}(n_{dest}))] \\
&\quad \wedge \neg A_3 \\
&\equiv [(\mathrm{NS}(n_{dest}) \in \{\texttt{Running, Completed}\} \wedge \mathrm{NS}(n_{src}) \neq \texttt{Completed}) \\
&\qquad \vee ((\exists e_j \in \Pi^S_{I\,red}\colon e_j = \texttt{Start}(n_{dest})) \wedge \\
&\qquad\quad ((\nexists e_i \in \Pi^S_{I\,red}\colon e_i = \texttt{End}(n_{src})) \vee \\
&\qquad\quad (\exists e_i \in \Pi^S_{I\,red}\colon e_i = \texttt{End}(n_{src}) \wedge i > j)))] \wedge \neg A_3 \\
&\equiv [(\mathrm{NS}(n_{dest}) \in \{\texttt{Running, Completed}\} \wedge \mathrm{NS}(n_{src}) \neq \texttt{Completed}) \\
&\qquad \vee ((\exists e_j \in \Pi^S_{I\,red}\colon e_j = \texttt{Start}(n_{dest}) \wedge \\
&\qquad\quad \nexists e_i \in \Pi^S_{I\,red}\colon e_i = \texttt{End}(n_{src})) \vee \\
&\qquad (\exists e_i, e_j \in \Pi^S_{I\,red}\colon e_j = \texttt{Start}(n_{dest}),\ e_i = \texttt{End}(n_{src}) \wedge i > j))] \\
&\quad \wedge \neg A_3 \\
&\equiv [(\exists e_j \in \Pi^S_{I\,red}\colon e_j = \texttt{Start}(n_{dest}) \wedge \nexists e_i \in \Pi^S_{I\,red}\colon e_i = \texttt{End}(n_{src})) \\
&\qquad \vee (\exists e_i, e_j \in \Pi^S_{I\,red}\colon e_j = \texttt{Start}(n_{dest}),\ e_i = \texttt{End}(n_{src}) \wedge i > j)] \\
&\quad \wedge \neg A_3 \\
&\equiv\colon (E_1 \vee E_2) \wedge \neg A_3 \ (\equiv (E_1 \wedge \neg A_3) \vee (E_2 \wedge \neg A_3))
\end{aligned}
$$

Because of $n_{src} \in \mathrm{pred}(\mathrm{S'}, n_{dest})$ and due to the compliance of I with S' the end entry of n_{src} cannot be situated before the start entry of n_{dest} in the execution history $\Pi^S_{I\,red}$; i.e., E_2 and therefore $(E_2 \wedge \neg A_3)$ cannot hold. Accordingly, $(E_1 \wedge \neg A_3)$ must hold.

$$
\begin{aligned}
&(E_1 \wedge \neg A_3) \\
&\equiv [\exists e_j \in \Pi^S_{I\,red}\colon e_j = \texttt{Start}(n_{dest}) \wedge \nexists e_i \in \Pi^S_{I\,red}\colon e_i = \texttt{End}(n_{src})] \wedge \\
&\quad [\mathrm{NS}(n_{src}) \neq \texttt{Skipped} \\
&\qquad \vee \mathrm{NS}(n_{dest}) \in \{\texttt{NotActivated, Activated, Skipped}\} \\
&\qquad \vee \exists n \in N_{critical}, \mathrm{NS(n)} \neq \texttt{Skipped}\colon \\
&\qquad\quad \nexists\ e_k, e_l \in \Pi^S_{I\,red}\colon l < k \wedge e_k = \texttt{Start}(n_{dest}),\ e_l = \texttt{End(n)}] \\
&\equiv [\exists e_j \in \Pi^S_{I\,red} : e_j = \texttt{Start}(n_{dest}) \wedge \\
&\qquad \nexists e_i \in \Pi^S_{I\,red}\colon e_i = \texttt{End}(n_{src}) \\
&\qquad \wedge \mathrm{NS}(n_{src}) \neq \texttt{Skipped}] \\
&\quad \vee [(\exists e_j \in \Pi^S_{I\,red}\colon e_j = \texttt{Start}(n_{dest}) \\
&\qquad \wedge \nexists e_i \in \Pi^S_{I\,red}\colon e_i = \texttt{End}(n_{src})) \\
&\qquad \wedge \mathrm{NS}(n_{dest}) \in \{\texttt{NotActivated, Activated, Skipped}\}) \vee \\
&\qquad (\exists e_j \in \Pi^S_{I\,red}\colon e_j = \texttt{Start}(n_{dest}) \wedge \\
&\qquad \nexists e_i \in \Pi^S_{I\,red}\colon e_i = \texttt{End}(n_{src}) \\
&\qquad \wedge (\exists n \in N_{critical}, \mathrm{NS(n)} \neq \texttt{Skipped}\colon \\
&\qquad\quad \nexists\ e_k, e_l \in \Pi^S_{I\,red}\colon l < k \wedge e_k = \texttt{Start}(n_{dest}),\ e_l = \texttt{End(n)}))] \\
&\equiv [\exists e_j \in \Pi^S_{I\,red}\colon e_j = \texttt{Start}(n_{dest}) \\
&\qquad \wedge \nexists e_i \in \Pi^S_{I\,red}\colon e_i = \texttt{End}(n_{src}) \wedge \mathrm{NS}(n_{src}) \neq \texttt{Skipped}] \\
&\quad \vee [(\exists e_j \in \Pi^S_{I\,red}\colon e_j = \texttt{Start}(n_{dest}) \wedge \nexists e_i \in \Pi^S_{I\,red}\colon e_i = \texttt{End}(n_{src})
\end{aligned}
$$

$\wedge$ ($\exists n \in N_{critical}$, NS(n) $\neq$ `Skipped`:
$\nexists\ e_k, e_l \in \Pi^S_{I\,red}$: $l < k \wedge e_k =$ `Start(`n_{dest}`)`, $e_l =$ `End(n)`))]
$\equiv$: $C_1 \vee C_2$

C_1 results in
NS(n_{dest}) $\in$ {`Running, Completed`} $\wedge$ NS(n_{src}) $\notin$ {`Completed, Skipped`}.
In this case I cannot be compliant with S'. Therefore C_2 must hold.

C_2
$\equiv$[$\exists e_j \in \Pi^S_{I\,red}$: $e_j =$ `Start(`n_{dest}`)`, $\nexists e_i \in \Pi^S_{I\,red}$: $e_i =$ `End(`n_{src}`)` $\wedge$
($\exists n \in N_{critical}$ with NS(n) $\neq$ `Skipped`:
$\nexists\ e_k, e_l \in \Pi^S_{I\,red}$: $l < k \wedge e_k =$ `Start(`n_{dest}`)`, $e_l =$ `End(n)`)]
$\equiv$ ($\exists e_j \in \Pi^S_{I\,red}$: $e_j =$ `Start(`n_{dest}`)` $\wedge$ $\nexists e_i \in \Pi^S_{I\,red}$: $e_i =$ `End(`n_{src}`)`) $\wedge$
($\exists n \in N_{critical}$, NS(n) $\neq$ `Skipped` $\wedge$
($\nexists e_l \in \Pi^S_{I\,red}$: $e_l =$ `End(n)` $\vee$
$\exists e_l \in \Pi^S_{I\,red}$: $e_l =$ `End(n)` $\wedge$ $j < l$))
$\equiv$ [($\exists e_j \in \Pi^S_{I\,red}$: $e_j =$ `Start(`n_{dest}`)` $\wedge$ $\nexists e_i \in \Pi^S_{I\,red}$: $e_i =$ `End(`n_{src}`)`)
$\wedge$ ($\exists n \in N_{critical}$, NS(n) $\neq$ `Skipped`
$\wedge$ $\nexists e_l \in \Pi^S_{I\,red}$: $e_l =$ `End(n)`)] $\vee$
[($\exists e_j \in \Pi^S_{I\,red}$: $e_j =$ `Start(`n_{dest}`)` $\wedge$ $\nexists e_i \in \Pi^S_{I\,red}$: $e_i =$ `End(`n_{src}`)`)
$\wedge$ ($\exists n \in N_{critical}$, NS(n) $\neq$ `Skipped`
$\wedge$ $\exists e_l \in \Pi^S_{I\,red}$: $e_l =$ `End(n)` $\wedge$ $j < l$)]
$\equiv$: $D_1 \vee D_2$

Because of D_1 it follows that there is a predecessor node n $\in N_{critical}$ of n_{src} which is neither marked as `Completed` nor as `Skipped` (see Figure C.1b). Referring to S' this node is also a predecessor of n_{dest} since S' contains the additional edge $n_{src} \rightarrow n_{dest}$. Accordingly, I cannot be compliant with S'.
D_2 yields that a predecessor node n $\in N_{critical}$ of n_{src} with NS(n) = `Completed` exists whose end entry is situated after the start entry of n_{dest} in the execution history $\Pi^S_{I\,red}$. Since n is a predecessor of n_{dest} in S' it follows that I is not compliant with S'. □

"$\Leftarrow$": $A_1 \vee A_2 \vee A_3 \Rightarrow$ I is compliant with S'
With A_1 it follows that $\Pi^S_{I\,red}$ still does not contain an entry related to n_{dest}. Therefore $\Pi^S_{I\,red}$ could have been produced on S' as well; i.e., I is compliant with S'. The same applies to A_2 because the end entry of n_{src} had been written into $\Pi^S_{I\,red}$ before the start entry of n_{dest} was logged.

After insertion of $n_{src} \rightarrow n_{dest}$, in any case, n_{src} has to be either executed or skipped before n_{dest} is activated or skipped. In addition, other (predecessor) nodes of n_{src}, which could have been executed parallel to n_{dest} so far may now have to be executed or skipped before n_{dest} can be marked. This node set is determined by $N_{critical}$ (see Figure C.1b). Only if each activity node of $N_{critical}$ has either been marked as `Skipped` or has written its end entry before the start entry of n_{dest} into $\Pi^S_{I\,red}$, the execution history can be produced on the new schema S' as well.

This follows directly from A_3. □

C.3 Proof (Theorem 4)

Theorem 4 (Deletion of Activities/Control Edges/Sync Edges) *Let S = (N, D, NT, CtrlE, SyncE, LoopE, DataE) be a correct process type schema (represented by a WSM–Net) and I be a process instance on S with reduced execution history* $\Pi^S_{I\,red}$ *and with marking* $M^S = (NS^S, ES^S)$*. Assume further that change operation* Δ *transforms S into a correct process type schema S' = (N', D', NT', CtrlE', SyncE', LoopE', DataE).*

(a) Δ *deletes an activity* n_{delete} *from S (including the re-linking of control edges), i.e.,* Δ *= deleteActivity(S,* n_{delete}*) (cf. Table 3.2). Then:*

I is compliant with S' $\Leftrightarrow$
$NS(n_{delete}) \in \{$`NotActivated, Activated, Skipped`$\}$

(b) Δ *deletes a control or sync edge c_sEdge =* (n_{src}, n_{dest}) *from S, i.e.,* Δ *= delete[Ctrl|Sync]Edges(S, c_sEdge) (cf. Table 3.1 + 3.2). Then:*

I is compliant with S'

(a) Δ deletes an activity n_{delete} from S (including the re-linking of control edges)

"$\Rightarrow$": I is compliant with S' $\Longrightarrow$ NS(n_{delete}) $\in$ {`NotActivated, Activated, Skipped`}

Proof by contradiction, we show:
NS(n_{delete}) $\notin$ {`NotActivated, Activated, Skipped`} $\Longrightarrow$ I is not compliant with S'

Assumption:
NS(n_{delete}) $\notin$ {`NotActivated, Activated, Skipped`}
$\equiv$ NS(n_{delete}) $\in$ {`Running, Completed`}
$\Longrightarrow \exists e_i \in \Pi^S_{I\,red}$ with e_i = `Start`(n_{delete})

Because of $n_{delete} \notin N'$ for each process instance on S' with execution history $\Pi^{\tilde{S'}}_{I\,red}$:

$\nexists e_k \in \Pi^{\tilde{S'}}_{I\,red}$ with e_k = `Start`(n_{delete})
$\Longrightarrow \Pi^S_{I\,red}$ cannot be produced on S'

$\Longrightarrow$ I is not compliant with S' □

"$\Leftarrow$": NS(n_{delete}) $\in$ {`NotActivated, Activated, Skipped`} $\Longrightarrow$ I is compliant with S'

<u>Case 1:</u> NS(n_{delete}) $\in$ {`NotActivated, Activated`}
$\Longrightarrow \forall$ n $\in$ c_succ*(S, n_{delete}) $\cup$ {n_{delete}}: NS(n) $\in$ {`NotActivated, Activated`}

$\Longrightarrow \forall$ n $\in$ c_succ*(S, n_{delete}) $\cup$ $\{n_{delete}\}$: $\nexists e_i \in \Pi^S_{I\,red}$ with e_i = `Start`(n) $\vee$ e_i = `End`(n)
$\Longrightarrow \forall e_i \in \Pi^S_{I\,red}$ with $e_i \in$ {`Start(n), End(n)`}: n $\notin$ c_succ*(S, n_{delete}) $\cup$ $\{n_{delete}\}$

All previous history entries have been produced by activities which either were predecessors of n_{delete} or were executable parallel to n_{delete}, i.e., the previous execution history can also be produced on S'.

Case 2: NS(n_{delete}) = `Skipped`
In this case n_{delete} has not yet written any history entry. Consequently the previous execution history can also be produced on S'. □

(b) Δ deletes a control or sync edge c_sEdge = (n$_{src}$, n$_{dest}$) from S

The only effect of deleting control or sync edge c_sEdge = (n$_{src}$, n$_{dest}$) (c_sEdge $\in$ *CtrlE* $\cup$ *SyncE*) is that previous predecessor activities of n_{dest} can be parallel executed with n_{dest} now. In principle, these parallel ordered activities can be finished before starting n_{dest} when executing a respective process instance on S'. Therefore an execution history $\Pi^S_{I\,red}$ can be also produced on S'. In particular, this holds if NS(n_{src}) = `Skipped` or NS(n_{dest}) = `Skipped`. □

C.4 Proof (Theorem 6)

Theorem 6 (Moving Activities) *Let S = (N, D, NT, CtrlE, SyncE, LoopE, DataE) be a correct process type schema (represented by a WSM–Net) and I be a process instance on S with reduced execution history* $\Pi^S_{I\,red}$ *and with marking* $M^S = (NS^S, ES^S)$. *Assume further that change operation* Δ *transforms S into a correct process type schema S' = (N', D', NT', CtrlE', SyncE', LoopE', DataE) by moving activity* n_{move} *from its current position to its new position within S'* *Thereby* Δ *adds set of control edges*

$CtrlE^{add}_{\Delta_T} = \{(n_1, n_2) | n_1 \in c\_pred(S, n_{move}),\ n_2 \in c\_succ(S, n_{move})\} \cup$
$\{(n_1, n_{move}) | n_1 \in c\_pred(S', n_{move})\} \cup$
$\{(n_{move}, n_2) | n_2 \in c\_sucee(S', n_{move})\}$
i.e., $\Delta = [serial|parallel|branch]moveActivity(S, n_{move}, \ldots)$ *(cf. Table 3.3). Then:*

I is compliant with S' ⇔
$\forall (n_{src}, n_{dest}) \in CtrlE^{add}_{\Delta_T}$:
$NS(n_{dest}) \in$ {`NotActivated, Activated, Skipped`} $\vee$
$[NS(n_{src})$ = `Completed` $\wedge\ NS(n_{dest}) \in$ {`Running, Completed`} *with*
$(\exists e_i$ = `End`$(n_{src}),\ e_j$ = `Start`$(n_{dest}) \in \Pi^S_{I\,red} \wedge i < j)]$

To prove Theorem 6 we draft the following Lemma 2. It states that for a sequence of deleting and inserting control and sync edges a serialization of these edge operations can be found for which each resulting intermediate process schema is correct, formally:

Lemma 2 (Serialization of Control / Sync Edge Insertions / Deletions) *Let S be a (correct) process type schema. Assume that S is transformed into another (correct) process type schema S' by applying control edge insertion and deletion operations $op_1, \ldots, op_k$, i.e., $op_i \in \{$addCtrlEdge(S, ...), deleteCtrlEdge(S, ...), insertSyncEdge(S, ...), deleteSyncEdge(S, ...)$\}$. Then:*

There is a serialization $op_{i_1}, \ldots, op_{i_k}$ of $op_1, \ldots, op_k$ such that a correct intermediate process schema results when applying each operation $op_{i_j} (j = 1, \ldots, k)$, formally:

$$S = S_0[op_{i_1} > S_1 \ldots S_{k-1}[op_{i_k} > S_k = S'$$

with S_μ is a correct process schema ($\mu = 1, \ldots, k$).

Proof of Lemma 2:

We show the proposition by induction over the number of applied edge operations k.

$k = 1$ (Inductive Beginning): $S = S_0[op_{i_1} > S_1 = S'$

Trivially, this is correct since S and S' are correct process schemes according to lemma assumption.

$k \rightarrow k + 1$ (Inductive Step): $S = S_0[op_{i_1} > S_1 \ldots S_k[op_{i_{k+1}} > S_{k+1} = S'$

Let $op_1, \ldots, op_{k+1}$ be edge operations which transform S into correct process schema S'. In particular, correctness of a process schema means that it is free of deadlock–causing cycles, i.e., cycles over control or sync edges besides desired loops over loop egdes. To show: $\exists i_1, \ldots, i_{k+1}$ with $\{1, \ldots, k+1\}$ such that:

$S = S_0[op_{i_1} > S_1 \ldots S_k[op_{i_{k+1}} > S_{k+1} := S'$ with S_μ correct for all $\mu = 1, \ldots, k+1$.

Regarding edge operations $op_1, \ldots, op_{k+1}$ we distinguish between three cases:

Case 1: All edge operations are additive, i.e., each time a control or sync edge is added.

In this case we get that S_k is correct since $S_{k+1} := S'$ is correct regarding assumption and therefore free of deadlock–causing cycles. S' results from adding a control or sync edge to S_k, i.e., S_k has exactly one control or sync edge less than S'. Since S' is free of deadlock–causing cycles S_k is free of deadlock–causing cycles. With inductive assumption $S_1, \ldots, S_{k-1}$ are free of deadlock–causing cycles.

Case 2: All edge operations are subtractive, i.e., each time a control or sync edge is deleted.

With inductive assumption we know that S_k is correct. S_{k+1} results from S_k by deleting a control or sync edge. Therefore if S_k is free of deadlock–causing cycles S_{k+1} is free of deadlock–causing cycles as well.

Case 3: There are at least one subtractive and one additive edge operation.

Let $op^* \in \{op_1, \ldots, op_k\}$ be a edge deletion operation. Let $op_{i_1} = op^*$. We immediately obtain that S_1 is correct. Furthermore, applying $\{op_1, \ldots, op_k\} \setminus \{op^*\}$ to S_1 results in correct

schema $S_{k+1} := S'$. With inductive assumption also intermediate schemes $S_2, \ldots, S_k$ have to be correct. □

Now we can prove Theorem 6.

Let $op_1, \ldots, op_n$ be the edge operations transforming S into S' with set of newly inserted control edges $CtrlE^{add}_{\Delta_T}$ and set of deleted control or sync edges $E^{del}_{\Delta_T}$. Accordingly, each insertion operation generates an entry in $CltrE^{add}_{\Delta_T}$ and each deletion operation an entry in $E^{del}_{\Delta_T}$.

We show Theorem 6 by induction over the number of applied edge operations n.

$n = 1$ (Inductive Beginning): $S := S_0[op_1 > S_1 := S'$

Case 1: op_1 adds a control edge, i.e., $CltrE^{add}_{\Delta_T} = \{n_{src} \rightarrow n_{dest}\}$ with $n_{src}, n_{dest} \in N$, $E^{del}_{\Delta_T} = \emptyset$. In this case the proposition follows with Theorem 3.

Case 2: op_1 deletes a control or sync egde, i.e., $E^{del}_{\Delta_T} = \{n_{src} \rightarrow n_{dest}\}$ with $n_{src}, n_{dest} \in N$, $CtrlE^{add}_{\Delta_T} = \emptyset$. In this case the proposition holds with $CtrlE^{add}_{\Delta_T} = \emptyset$ and Theorem 4.

$n \longrightarrow n + 1$ (Inductive Step): $S := S_0[op_1 > S_1 \ldots S_n[op_{i_{n+1}} > S_{n+1} := S'$

With Lemma 2 there is a serialization $i_1, \ldots, i_{n+1}$ with
$S := S_0[op_{i_1} > S_1 \ldots S_n[op_{i_{n+1}} > S_{n+1} >:= S'$ with S_μ is a correct intermediate schema $(\mu = 1, \ldots, n + 1)$.

Let $CtrlE^{add^{(n)}}_{\Delta_T}$ be the set of all control edges added by $op_{i_1}, \ldots, op_{i_n}$. With inductive assumption follows:

I is compliant with S' $\Longleftrightarrow$

$\forall n_{src} \rightarrow n_{dest} \in CtrlE^{add^{(n)}}_{\Delta_T}$:
$\quad \mathrm{NS}(n_{dest}) \in \{\texttt{NotActivated, Activated, Skipped}\} \vee$
$\quad (\mathrm{NS}(n_{dest}) \in \{\texttt{Running, Completed}\} \wedge \mathrm{NS}(n_{src}) = \texttt{Completed} \wedge$
$\qquad ((\exists e_i = \texttt{End}(n_{src}),\ e_j = \texttt{Start}(n_{dest})) \in \Pi^S_{I\,red} \wedge i < j))$ (*)

Case 1: $op_{i_{n+1}}$ adds a control edge $n_{src} \rightarrow n_{dest}$
Then the proposition directly follows with (*) and Theorem 3.

Case 2: $op_{i_{n+1}}$ adds a control edge $n_{src} \rightarrow n_{dest}$
Then the proposition directly follows with (*) and Theorem 4 ($CtrlE^{add^{(n+1)}}_{\Delta_T} = CtrlE^{add^{(n)}}_{\Delta_T}$).□

C.5 Proof (Theorem 8)

Theorem 8 (Compliance Conditions For Data Flow Changes) *Let S = (N, D, NT, CtrlE, SyncE, LoopE, DataE) be a correct process type schema (represented by a WSM Net) and I be a process instance on S with reduced execution history* $\Pi^S_{I\,red}$ *and with marking* $M^S = (NS^S, ES^S)$. *Assume further that change operation* Δ *transforms S into a correct*

process type schema S' = (N', D', NT', CtrlE', SyncE', LoopE', DataE).

(a) Δ inserts a data element d into S, i.e., Δ = addDataElements(S, {d}, ...). Then:

I is compliant with S'.

(b) Δ deletes a data element d from S, i.e., Δ = deleteDataElements(S, {d}, ...). Then:

I is compliant with S' ⇔
No read or write access on d by an activity with state `Running` *or* `Completed`

(c) Δ inserts or deletes a read edge (d, n, read),
i.e., Δ ∈ {addDataEdges(S, {(d, n, read)}), deleteDataEdges(S, {(d, n, read)}}. Then:

I is compliant with S' ⇔ NS(n) ∈ {`NotActivated, Activated, Skipped`}

(d) Δ inserts or deletes a write edge (d, n, write),
i.e., Δ ∈ {addDataEdges(S, {(d, n, write)}), deleteDataEdges(S, {(d, n, write)}}. Then:

I is compliant with S' ⇔ NS(n) ≠ `Completed`

(a) Adding a data element d to S has no effects on read or write accesses on existing data elements. In particular, $\Pi_I^{S^{red}}$ is also producable on S'.

(b) Deleting data element d

"$\Longrightarrow$": I is compliant with $S' \Longrightarrow$
[No read or write access on d by an activity with state `Running` or `Completed`]

We show the above propostion by contradiction with contradictionary assumption:

[$\exists n^*$ with NS(n^*) ∈ {`Running, Completed`} ∧
there has been a read or write access or n^* on d] $\Longrightarrow$ I not compliant with S'

Assumption:
[$\exists n^* \in N$ with NS(n^*) ∈ {`Running, Completed`} ∧
there has been a read or write access or n^* on d]
$\equiv \exists e_\mu^{((d_1^{(\mu)},v_1^{(\mu)}),\ldots,(d_m^{(\mu)},v_m^{(\mu)}))} \in \Pi^S_{I\,red}$ with e_μ ∈ {`Start(`n^*`), End(`n^*`)`}
$\wedge\ \exists l \in \{1,\ldots,m\}$ with $d_l^{(\mu)} = d$, $n^* \in N$

With this we get that I cannot be compliant with S' since d is not present in S'. □

"$\Longleftarrow$":

$A \equiv$ [$\nexists$ activity X (with NS(X) ∈ `Running` or `Completed` with read or write access on d]

$A \Longrightarrow I$ is compliant with S'

I is compliant with S' if $\Pi^S_{I\,red}$ can be produced on S' as well, in particular:

$\nexists e_\mu^{((d_1^{(\mu)},v_1^{(\mu)}),\ldots,(d_m^{(\mu)},v_m^{(\mu)}))} \in \Pi_{I\,red}^S$ with $e_\mu \in \{\texttt{Start}(n^*), \texttt{End}(n^*)\}$
$\wedge\ \exists l \in \{1,\ldots,m\}$ with $d_l^{(\mu)} = d$, $n^* \in N$

This is true due to the validity of A. □

(c) Adding or Deleting a read data edge

(c1) Adding a read data edge $(d, n, read)$

"$\Longrightarrow$": I is compliant with $S' \Longrightarrow$ NS(n) $\in \{\texttt{NotActivated}, \texttt{Activated}, \texttt{Skipped}\}$

We proof this proposition by contradiction using the following contradictionary assumption:

NS(n) $\in \{\texttt{Running}, \texttt{Completed}\} \Longrightarrow I$ is not compliant with S'

Assumption: NS(n) $\in \{\texttt{Running}, \texttt{Completed}\} \Longrightarrow$
$\exists e_\mu^{((d_1^{(\mu)},v_1^{(\mu)}),\ldots,(d_m^{(\mu)},v_m^{(\mu)}))} \in \Pi_{I\,red}^S$ with $e_\mu = \texttt{Start}(n^*)$

Since activity n has been already started when read data edge $(d, n, read)$ is inserted n has already performed its read data accesses but no read access on d, i.e., $\nexists l \in \{1,\ldots,m\}$ with $d_\mu^{(1)} = d$. Then I cannot be compliant with S'. □

"$\Longleftarrow$": NS(n) $\in \{\texttt{NotActivated}, \texttt{Activated}, \texttt{Skipped}\} \Longrightarrow I$ is compliant with S'

If I should be compliant with S' $\Pi_{I\,red}^S$ has to be also producable on S'. In particular, there should be no $e_\mu^{((d_1^{(\mu)},v_1^{(\mu)}),\ldots,(d_m^{(\mu)},v_m^{(\mu)}))} \in \Pi_{I\,red}^S$ with $e_\mu = \texttt{Start}(n^*)$. Otherwise $d_\mu^{(l)} \neq d \forall l$ would hold what is in conflict to assumption NS(n) $\in \{\texttt{NotActivated}, \texttt{Activated}, \texttt{Skipped}\}$. □

(c2) When deleting a read data edge $(d, n, read)$ we can argue similarily to case (c2). Reason is that if $(d, n, read)$ is removec from S n must not have read d since this read access is not longer possible based on S'. Read accesses are always performed when starting activities what results in the fact that n has not been started yet. □

(d) When inserting and deleting write data edges we can argue similarily to case (c). The only difference is that affected activity n can already be started. Reason is that write data accesses take place not until completion of an activity. Therefore modifications of an activity's write data edges can be carried out as long as this activity is not completed. □

C.6 Proof (Theorem 9)

Theorem 9 (Correctness of Marking Adaptation Approach) *Let S be a process type schema and I an unbiased process instance running on S with instance marking $M^S = (NS^S, ES^S)$ and execution history Π_I^S. Let further Δ_T be a process type change which transforms S into another (correct) process schema S' and let I be compliant with S'. Then the instance markings resulting from replaying Π_I^S on S', i.e., $M_{replay}^{S'} = (NS_{replay}^{S'}, ES_{replay}^{S'})$ conincides with the marking*

resulting from applying Algorithms 1 + 2, i.e., $M_{adapt}^{S'} = (NS_{adapt}^{S'}, ES_{adapt}^{S'})$. *Fomrally:*

$$NS_{replay}^{S'} = NS_{adapt}^{S'} \wedge ES_{replay}^{S'} = ES_{adapt}^{S'}$$

Generally, the following proposition ($\star$) holds:
Algorithm 2 initially adopts previous markings for all activities and edges which are present in original schema S and changed schema S'. Then all activities and edges which are concerned by change operation Δ_T are newly evaluated, e.g., by skipping activities which are inserted into already skipped branches.

We show the proposition of Theorem 9 by induction over the number t of entries within Π_I^S.

(Inductive Assumption): Let S be a (correct) process type schema and I a process instance on S with execution history $\Pi_I^S =< e_0, \ldots, e_t >$. Let $\Delta_T = op_1, \ldots, op_n$ be a process type change which transforms S into another (correct) process schema S'. Let $\mathrm{M}_{replay}^{S'} = (\mathrm{NS}_{replay}^{S'}, \mathrm{ES}_{replay}^{S'})$ and $\mathrm{M}_{adapt}^{S'} = (\mathrm{NS}_{adapt}^{S'}, \mathrm{ES}_{adapt}^{S'})$ be as described in Theorem 9.
Then $\mathrm{NS}_{replay}^{S'} = \mathrm{NS}_{adapt}^{S'} \wedge \mathrm{ES}_{replay}^{S'} = \mathrm{ES}_{adapt}^{S'}$ (§).

$t = 1$ (Inductive Beginning): $\Pi_I^S =< e_1 >$ with $e_1 =$ `Start(X)` $\wedge$ NT(X) = `StartFlow` ($\diamondsuit$)

Since I is compliant with S' we obtain that $X \in N'$ and NT(X) = `StartFlow` holds for S' ($\clubsuit$). Then replaying Π_I^S on S' results in marking $\mathrm{M}_{replay}^{S'}$ where
$\mathrm{NS}_{replay}^{S'}$(X) = `Running`, $\mathrm{NS}_{replay}^{S'}$(n) = `NotActivated` $\forall n \in N' \setminus \{X\}$,
and $\mathrm{ES}_{replay}^{S'}$(e) = `NotSignaled` $\forall e \in E'$[1].

Because of ($\diamondsuit$) regarding original schema S it follows that
NS^S(X) = `Running`, NS^S(n) = `NotActivated` $\forall n \in N \setminus \{X\}$,
and ES^S(e) = `NotSignaled` $\forall e \in E$ hold.

Due to ($\clubsuit$) Δ_T concerns not yet executed and therefore not yet marked regions of S and S'. Therefore marking $\mathrm{MS}^S = (\mathrm{NS}^S, \mathrm{ES}^S)$ is unalteredly transferred to S' by Algorithm 2. □

$t \longrightarrow t + 1$ (Inductive Step): $\Pi_I^S =< e_1, \ldots, e_{t+1} >$ with $e_{t+1} \in$ `Start(X), End(X)` (¶)

Since I is compliant with S' we obtain that $X \in N'$ holds ($\heartsuit$).

Case 1: $e_{t+1} =$ `Start(X)`
Replaying Π_I^S*:* With ($\heartsuit$) we obtain that $\mathrm{NS}_{replay}^{S'}$(X) = `Running` holds. Since markings of all other activities remain unaltered when replaying Π_I^S it follows that $\mathrm{NS}_{replay}^{S'}$(n) = $\mathrm{NS}^{S'}$(n) $\forall n \in N' \setminus \{X\}$ holds. Furthermore replaying a `Start` event on a process schema obviously has no effects on edge markings and therefore: $\mathrm{ES}_{replay}^{S'}$(e) = $\mathrm{ES}^{S'}$(e) $\forall e \in E'$.
Applying Algorithm 2: Applying inductive assumption (§) Algorithm 2 yields the same marking as results from replaying Π_I^S on S' if X has not been started yet. With ($\heartsuit$) and Π_I^S we obtain that $\mathrm{NS}^S(X)$ = `Running` holds. This marking is transferred to S' applying proposition ($\star$). Looking at Algorithm 2 we see that it never changes marking of running activities. Therefore marking $\mathrm{NS}^S(X)$ = `Running` is also unalteredly transferred to S'. Furthermore starting activity

[1] E = (CtrlE ∪ SyncE ∪ LoopE) and E' = (CtrlE' ∪ SyncE' ∪ LoopE')

X has no influence on edge markings, i.e., $\mathrm{ES}^S = \mathrm{ES}^{S'}_{adapt}$ holds. □

Case 2: e_{t+1} = `End(X)`

Case 2.1: $\forall(X, n_{dest}, [\mathrm{lCond}]) \in E : (X, n_{dest}) \in E'$, i.e., all outgoing edges and therefore all direct successors n_{dest} of X in S' have been already present in S.
Replaying Π^S_I*:* With (♡) we obtain that $\mathrm{NS}^S(X) = NS^{S'}_{replay}(X)$ = `Completed` holds (since I is compliant with S' there must be a `Start` entry for X in Π^S_I). The completion of X only affects the markings of direct successors of X (in S and S'). More precisely, for all direct successors of X in S' Rules 1 (cf. Appendix B) are executed in the same way for S and S' since all direct successors are present in S and S'. Therefore we obtain that $\mathrm{NS}^{S'}_{replay} = \mathrm{NS}^S$ and consequently $\mathrm{ES}^{S'}_{replay} = \mathrm{ES}^S$ holds.

Applying Algorithm 2: Since $(\{X\} \cup \{n | (X, n) \in E\}) \subseteq N' \wedge \forall(X, n_{dest}) \in E : (X, n_{dest}) \in E'$ holds and I is compliant with S', Δ_T does not concern X or one of its direct successors. Of course, Δ_T also does not affect any predecessors of X since I is compliant with S'. Therefore again Algorithm 2 takes over marking MS^S on S and does not carry out any further changes.

Case 2.2: At least one of the outgoing edges of X was not present in S (we summarize these edges in E_{new}).

After replaying Π^S_I on S', X is marked as `Completed` and all outgoing edges are marked according to Rules 1 (cf. Appendix B) (and consequently all edges in E_{new}). Since the outgoing edges are newly marked the respective destination activities are also evaluated according to Rules 1 (cf. Appendix B). According to inductive assumption (§), $\mathrm{MS}^{S'}_{replay} = \mathrm{MS}^{S'}_{adapt}$ holds before completion of X. According to proposition ($\star$) Algorithm 2 adopts $\mathrm{NS}^S(X)$ = `Completed` and the markings of all activities and edges present in S within the initialization phase. Since $E_{new} \not\subseteq E$ holds we conclude that $E_{new} \subseteq E_{check}(\Delta_T)$ holds what results in a re–evaluation of edges contained in E_{new}. This re–evaluation executed by Algorithm 2 also follows Rules 1 (cf. Appendix B) . Therefore all edges contained in E_{new} are equally marked when replaying Π^S_I in S' or applying Algorithm 2. For edges in $E_{check}(\Delta_T) \setminus E_{new}$ the necessary marking adaptations are already covered by inductive assumption (§). □

C.7 Proof (Proposition 1)

Proposition 1 (Avoiding Missing Input Data and Lost Updates) *Let S be a WSM Net and I be a biased instance with starting schema S and instance–specific schema* $S_I := S + \Delta_I = (N_I, D_I, NT_I, CtrlE_I, SyncE_I, \ldots)$*. Assume that type change* Δ_T *transforms S into a correct schema S' = (N', D', NT', CtrlE', SyncE', ...). Then: Propagating* Δ_T *to I neither results in missing input data nor in lost updates if*

$\forall\ mDL_1 = (d_1, mode_1, [\text{"add"}|\text{"delete"}]) \in \mathcal{AD}(S, \Delta_T) \cup \mathcal{DD}(S, \Delta_T)$,
$\forall\ mDL_2 = (d_2, mode_2, [\text{"add"}|\text{"delete"}]) \in \in \mathcal{AD}(S, \Delta_I) \cup \mathcal{DD}(S, \Delta_I)$

with $mode_i \in \{read, write\}$ *(i = 1, 2):*
$d_1 \neq d_2 \vee$
$mode1 = mode2 = read \vee$
$mDL_1 = (d_1, "read", "delete") \vee mDL_2 = (d_2, read, "delete")$ (♣)

whereas

- $\mathcal{AD}(S, \Delta_T) := \{(d, mode, "add") \in DataE' \setminus DataE, mode \in \{read, write\}\}$
- $\mathcal{DD}(S, \Delta_T) := \{(d, mode, "delete") \in DataE \setminus DataE', mode \in \{read, write\}\}$
- $\mathcal{AD}(S, \Delta_I) := \{(d, mode, "add") \in DataE_I \setminus DataE, mode \in \{read, write\}\}$
- $\mathcal{DD}(S, \Delta_I) := \{(d, mode, "delete") \in DataE \setminus DataE_I, mode \in \{read, write\}\}$

Additional Suppositions:

1. S, S', and S_I are correct WSM Nets (cf. Definition 2)
2. $\Delta_T \cap \Delta_I = \emptyset$

Let $S'_I := (S + \Delta_I) + \Delta_T$ with $S'_I = (N'_I, D'_I, DataE'_I, CtrlE'_I, \ldots)$ be the instance–specific schema resulting from propagating type change Δ_T to instance–specific schema S_I.

Formalization Of Proposition:

1. **Propostion 1:** S'_I contains no activities with missing input data, i.e.,
   $(\forall X \in N'_I$ with obligatory input data: $\exists d \in D'_I$ with $(d, X, read) \in DataE'_I)(\theta) \wedge$
   (Let A_X be the set of all action sets leading to activation of X $(X \in N'_I)$. Then:
   $\forall X \in N'_I, d \in D'_I$ with $\exists e = (X, d, read) \in DataE'_I$ with obligatory input data:
   $\forall V \in A_X : \exists X^* \in V \wedge \exists e^* = (X^*, d, write) \in DataE'_I)$ (λ)
2. **Proposition 2:** S'_I contains no write data edges causing lost updates at runtime, i.e.
   $\forall n_1, n_2 \in \mathrm{N}^*$, $n_1 \neq n_2$, $\mathrm{d} \in \mathrm{D}^* \wedge \exists\ e_1 = (n_1, \mathrm{d}, \mathrm{write})$, $e_2 = (n_2, \mathrm{d}, \mathrm{write}) \in \mathrm{DataE}^* \wedge$
   $(split, join) := BranchNodes(S, n_1, n_2)$[2] $\neq$ (`Undefined, Undefined`):
   $(\mathrm{NT}(join) \neq$ `AndJoin`$) \vee (n_1 \in \mathrm{succ}^*(n_2) \vee n_1 \in \mathrm{pred}^*(n_2))$

We first prove part (θ) of Propostion 1, i.e.,
$(\clubsuit) \Longrightarrow \forall X \in N'_I$ with obligatory input data: $\exists d \in D'_I$ with $(d, X, read) \in DataE'_I$.
Due to supposition 1 condition (θ) holds for S, S_I and S'. Therefore if activity X has been already present in N there has been also a read data edge $(d, X, read) \in DataE$. If Δ_T or Δ_I delete $(d, X, read)$ they also have to delete X due to Supposition 1. Consequently, either X is not present in N'_I or read data edge $(d, X, read)$ is still present in $DataE'_I$. If otherwise X has not been present in N either Δ_I or Δ_T has inserted X (not both due to Supposition 2)

[2]Function *BranchNodes(...)* is defined in Table B.1 (cf. Appendix B)

together with read data edge $(d, X, read)$ (cf. Supposition 1). Let without loss of generality Δ_T be the change which has inserted activity X and data edge $(d, X, read)$. Then it is not possible that Δ_I deletes $(d, X, read)$ since Δ_I does not "know" activity X and the connected data edges. Consequently, read data egde $(d, X, read)$ is still present in $DataE'_I$. □

Condition (λ) of Propostion 1 is proven by contradiction with contractionary assumption:

$\exists$ X $\in$ N'_I, d $\in$ D'_I with $\exists$ e = (X, d, read) $\in$ DataE$'_I$ with obligatory input data:
$\exists$ V $\in$ A_X: $\nexists$ $\bar{X}$ $\in$ V with $\exists$ ($\bar{X}$, d, write) $\in$ DataE$'_I$

$\overset{Supp.1}{\Longrightarrow}$
[($\exists$ ($\bar{X}$, d, write) $\in$ DeletedDataEdges(S, Δ_T) $\wedge$ ($\exists$ (X, d, read) $\in$ AddedDataEdges(S, Δ_I)) $\vee$
($\exists$ ($\bar{X}$, d, write) $\in$ DeletedDataEdges(S, Δ_I) $\wedge$ ($\exists$ (X, d, read) $\in$ AddedDataEdges(S, Δ_T))][3] $\vee$
[($\exists$ $\tilde{e} = (\tilde{X}$, d, write) $\in$ DataE:
($\tilde{e}$ $\in$ DeletedEdges(S, Δ_I) $\wedge$ ($\bar{X}$, d, write) $\in$ DeletedEdges(S, Δ_T) $\vee$
($\tilde{e}$ $\in$ DeletedEdges(S, Δ_T) $\wedge$ ($\bar{X}$, d, write) $\in$ DeletedEdges(S, Δ_I))][4]

$\Longrightarrow$
Δ_T and Δ_I both work on same data element d $\wedge$
Δ_I or Δ_T both delete at least on write data edge

$\equiv \neg$ (λ) in Proposition 1 □

We now show Proposition 2:

[$\forall n_1, n_2 \in$ N*, $n_1 \neq n_2$, d $\in$ D* $\wedge$ $\exists$ $e_1 = (n_1$, d, write), $e_2 = (n_2$, d, write) $\in$ DataE* $\wedge$
(*split, join*) := *BranchNodes*(S, n_1, n_2) $\neq$ (`Undefined, Undefined`):
(NT(*join*) $\neq$ `AndJoin`) $\vee$ ($n_1 \in$ succ$^*(n_2)$ $\vee$ $n_1 \in$ pred$^*(n_2)$)] := C

Proof by Contradiction:

$\neg$ C $\equiv$
$\exists n_1, n_2 \in$ N*, $n_1 \neq n_2$, d $\in$ D* $\wedge$ $\exists$ $e_1 = (n_1$, d, write), $e_2 = (n_2$, d, write) $\in$ DataE* $\wedge$
(*split, join*):=*BranchNodes*(S, n_1, n_2) $\neq$ (`Undefined, Undefined`):
(NT(*join*) = `AndJoin`) $\wedge$ ($n_1 \notin$ succ$^*(n_2)$ $\wedge$ $n_1 \notin$ pred$^*(n_2)$)

$\overset{Supp.1}{\Longrightarrow}$ ($e_1 \in$ AddedEdges(S, Δ_I) $\wedge$ $e_2 \in$ AddedEdges(S, Δ_T) or vice versa)

$\Longrightarrow$ AddedWriteAccesses(S, Δ_I) $\cap$ AddedWriteAccesses(S, Δ_T) = {d}

$\equiv \neg$ Proposition 2 □

[3] $X \notin N$
[4] $X \in N$

C.8 Proof (Proposition 2)

Proposition 2 (Basic Deadlock Prevention) *Let S be a WSM Net and I be a biased instance with starting schema S and instance–specific schema*
$S_I := S + \Delta_I = (N_I, D_I, NT_I, CtrlE_I, SyncE_I, \ldots)$. *Assume that type change* Δ_T *transforms S into a correct schema S' = (N', D', NT', CtrlE', SyncE', ...).*

Then: $S'_I = (S + \Delta_I) + \Delta_T$ *does not contain deadlock-causing cycles if the following condition holds:*

$$\forall\ (s_1, d_1) \in \mathcal{AS}(S, \Delta_T), \forall\ (s_2, d_2)\ \mathcal{AS}(S, \Delta_I): \\ d_1 \notin (pred^*(S, s_2) \cup \{s_2\}) \vee d_2 \notin (pred^*(S, s_1) \cup \{s_1\})\ (\Psi)$$

whereas

- $\mathcal{AS}(S, \Delta_T) := SyncE' \setminus SyncE$
- $\mathcal{AS}(S, \Delta_I) := SyncE_I \setminus SyncE$

Additional Suppositions:

1. S, S', and S_I are correct WSM Nets (cf. Definition 2)
2. $\Delta_T \cap \Delta_I = \emptyset$

Let $S'_I := (S + \Delta_I) + \Delta_T$ with $S'_I = (N'_I, D'_I, DataE'_I, CtrlE'_I, \ldots)$ be the instance–specific schema resulting from propagating type change Δ_T to instance–specific schema S_I.

Formalization Of Proposition:

For $S'_{I\,fwd}$:=(N'_I, CtrlE$'_I$, SyncE$'_I$) holds: ($\forall$ n* $\in$ N$'_I$: n* $\notin$ succ*($S'_{I\,fwd}$, n*))

Proof by contradiction with contratictionary assumption:
$\neg$ ($\forall$ n* $\in$ N$'_I$: n* $\notin$ succ*($S'_{I\,fwd}$,n*))
$\equiv$ $\exists$ n* $\in$ N$'_I$: n* $\in$ succ*($S'_{I\,fwd}$,n*)
$\equiv$ $\exists$ edge sequence $EdgeSeq := e_1 \rightarrow \ldots \rightarrow e_{k-1}$ with
e_1= n* $\rightarrow n_1$, $e_{k-1} = n_k \rightarrow$ n* $\in$ (CtrlE$'_I$ $\cup$ SyncE$'_I$)

$\overset{Supp.1}{\Longrightarrow}$ Edge sequence $EdgeSeq$ is not present in S, S_I, and S' but $EdgeSeq$ is present in S'_I

$\Longrightarrow$ Δ_T as well as Δ_I must have inserted at least one edge within $EdgeSeq$, i.e.,
$\exists\ e_i = (s_i, d_i) \in (CtrlE' \cup SyncE') \setminus (CtrlE \cup SyncE) \wedge$
$\exists e_j = (s_j, d_j) \in (CtrlE_I \cup SyncE_I) \setminus (CtrlE \cup SyncE)$ with
$e_i, e_j \in \{e_1, \ldots, e_n | EdgeSeq = e_i \rightarrow \ldots \rightarrow e_n\}$

$\Longrightarrow$ ($d_i \in$ pred*($S'_{I\,fwd}$, s_j) $\cup\{s_j\}$) $\wedge$ ($d_j \in$ pred*($S'_{I\,fwd}$, s_i) $\cup\{s_i\}$)

$\Longrightarrow$ Contradiction to Proposition $\square$

C.9 Proof (Theorem 10)

Theorem 10 (Trace Equivalence By Process Schema Isomorphism) *Let S_1 and S_2 be two WSM Nets. Then S_1 and S_2 are trace equivalent if S_1 and S_2 are isomorphic according to Definition 12. Formally:*

$$S_1 \simeq S_2 \Longrightarrow S_1 \equiv_{trace} S_2$$

In the following, for process schema S let Ω_S comprise all producable execution histories on S.

Proposition: $S_1 \simeq S_2 \Longrightarrow S_1 \equiv_{trace} S_2$

We show this by induction over the length k of an arbitrary execution history
$\Pi_I^{S_i} =< e_1, \ldots, e_k > \in \Omega_{S_i}$ (i = 1, 2) with
$e_j \in \{\texttt{START}(\text{label(a)}), \texttt{END}(\text{label(a)})\}$; $j = 1, \ldots, k$; a $\in N_i$.

Inductive Assumption (IA):

$S_1 \simeq S_2 \Longrightarrow \forall\ \Pi_I^{S_1} =< e_1, ...e_k > \in \Omega_{S_1}$: $\Pi_I^{S_1} \in \Omega_{S_2}$,
i.e., $\Pi_I^{S_1}$ can be produced on S_2 as well.

Note that we restrict our considerations to the direction from S_1 to S_2. Trivially, the reverse direction – $\forall\ \Pi_I^{S_2} =< e_1, e_2, ...e_k > \in \Omega_{S_2}$: $\Pi_I^{S_2} \in \Omega_{S_1}$ – can be proven analogously.

Inductive Beginning (IB):

$\Pi_I^{S_1} =< e_1 > \in \Omega_{S_1}$ with $e_1 = \texttt{START}(\textit{label}(\text{a}))$

With condition (♣) from Definition 12 it follows that there is an image activity f(a) $\in N_2$ (N_2 denotes the node set of S_2) with $\textit{label}(\text{f(a)}) = \textit{label}(\text{a})$. Since activity "a" has written the first entry into $\Pi_I^{S_1}$ it must be a start activity of S_1, i.e., a node without incoming control edges. With (♣) from Definition 12 it directly follows that the image f(a) $\in N_2$ in S_2 has no incoming control edges as well (edge order preserving property). Consequently, activity execution order given by $\Pi_I^{S_1}$ can be produced on S_2 as well.

However, we still have to care about the possible read accesses produced by activity "a". Due to $e_1 = \texttt{START}(\textit{label}(\text{a}))$ activity "a" was already started, i.e., it has already read the set of process data elements $D_1^a \subseteq D_1$ to which it is linked via a set of read data edges $DataE_1^a \subseteq DataE_1$. With (♣) from Definition 12 it follows:

$\forall d_j \in D_1^a$: $\exists\ g(d_j) \in D_2^a$ with $\text{label}(d_j) = \text{label}(g(d_j)) \wedge$
$\forall dE_i \in DataE_1^a$: $\exists\ g(dE_i) \in DataE_2^a$ with $\text{label}(dE_i) = \text{label}(g(dE_i))$.
Consequently, all entries produced by read data accesses of a in $\Pi_I^{S_1}$ can be produced on S_2 as well.

Inductive Step (IS):

$\Pi_I^{S_1} = < e_1, e_2, ..., e_k, e_{k+1} > \in \Omega_{S_1}$

Let $a \in N_1$ be the activity which has written entry e_k into $\Pi_I^{S_1}$ and let $\tilde{a} \in N_1$ be the activity which has written e_{k+1} into $\Pi_I^{S_1}$.

Due to **(IA)** it follows that ${\Pi_I^{S_1}}' < e_1, e_2, ..., e_k > \in \Omega_{S_2}$. Now we analyze the order relation beween activities a and $\tilde{a}$ regarding S_1.

Case 1: $a = \tilde{a} \Longrightarrow e_k = \texttt{START}(\text{a}) \wedge e_{k+1} = \texttt{END}(\text{a})$ (*)

Taking **(IA)** and (*), trivially, $\Pi_I^{S_1} \in \Omega_{S_2}$ holds.

Case 2: $a \neq \tilde{a}$

For this case, activity a is either a direct predecessor of $\tilde{a}$ or a and $\tilde{a}$ are ordered in parallel regarding schema S_1.

Case 2.1: a is a direct predecessor of $\tilde{a}$

With (♣) from Definition 12 it follows:
$\exists$ f(a), f($\tilde{a}$) in N_2 with: f(a) is a direct predecessor of f($\tilde{a}$).
For this case, it directly follows that $\Pi_I^{S_1} \in \Omega_{S_2}$ holds.

Case 2.2: a and $\tilde{a}$ are ordered in parallel

With (♣) from Definition 12 it follows:
$\exists$ f(a), f($\tilde{a}$) in N_2 with: f(a) and f($\tilde{a}$) are ordered in parallel.
For this case, trivially, $\Pi_I^{S_1} \in \Omega_{S_2}$ holds.

However, we still have to care about the possible read and write data accesses produced by activity $\tilde{a}$. In doing so, we distinguish between the following cases:

Case 1: $e_{k+1} = \texttt{START}(\text{label}(\tilde{a}))$

Activity $\tilde{a}$ has read the set of process data elements ${D_1}^{\tilde{a}} \subseteq D_1$ to which it is linked via a set of read data edges ${DataE_1}^{\tilde{a}} \subseteq DataE_1$. With (♣) from Definition 12 it follows:
$\forall d_j \in {D_1}^{\tilde{a}}$: $\exists\ g(d_j) \in {D_2}^{\tilde{a}}$ with label($g(d_j)$) = label(d_j) $\wedge$
$\forall dE_i \in {DataE_1}^{\tilde{a}}$: $\exists\ g(dE_i) \in {DataE_2}^{\tilde{a}}$ with label($g(dE_i)$) = label(dE_i)

Case 2: $e_{k+1} = \texttt{END}(\text{label}(\tilde{a}))$

Since $\tilde{a}$ has been already completed, there were write data accesses of $\tilde{a}$ on a set of data elements ${D_1}^{\tilde{a}} \subseteq D_1$. For them we can argue analogously to read data accesses in Case 1. □

C.10 Proof (Theorem 11)

Theorem 11 (Equivalent Changes) *Let S be a (correct) process schema and let $\Delta_i, i = 1, 2$ be two changes which transform S into (correct) process schemes $S_i, i = 1, 2$. Then Δ_1 and Δ_2 are equivalent, i.e, $\Delta_1 \equiv \Delta_2$ if the following conditions (1) – (4) hold:*

1. **Change Operations on Activity Sets:**

   (a) $\Delta_1[ins\_Act] \equiv \Delta_2[ins\_Act] \Longleftrightarrow$
   $(N^{add}_{\Delta_1} = N^{add}_{\Delta_2} \wedge$
   $AnchorIns(S, \Delta_1) = AnchorIns(S, \Delta_2) \wedge$
   $OrderIns(S, \Delta_1) = OrderIns(S, \Delta_2))$

   (b) $\Delta_1[del\_Act] \equiv \Delta_2[del\_Act] \Longleftrightarrow N^{del}_{\Delta_1} = N^{del}_{\Delta_2}$

   (c) $\Delta_1[move\_Act] \equiv \Delta_2[move\_Act] \Longleftrightarrow$
   $(N^{move}_{\Delta_1} = N^{move}_{\Delta_2}) \wedge$
   $AnchorMove(S, \Delta_1) = AnchorMove(S, \Delta_2)) \wedge$
   $OrderMove(S, \Delta_1) = OrderMove(S, \Delta_2))$

   (d) $\Delta_1[ins/move\_Act] \equiv \Delta_2[ins/move\_Act] \Longleftrightarrow$
   $(N^{move}_{\Delta_1} = N^{move}_{\Delta_2}) \wedge$
   $\Delta_1[ins\_Act] \equiv \Delta_2[ins\_Act] \wedge$
   $\Delta_1[move\_Act] \equiv \Delta_2[move\_Act] \wedge$
   $OrderAgg(S, \Delta_1) = OrderAgg(S, \Delta_2))$

2. **Change Operations on Sync Edges:**

   (a) $\Delta_1[ins\_Sync] \equiv \Delta_2[ins\_Sync] \Longleftrightarrow SyncE^{add}_{\Delta_1} = SyncE^{add}_{\Delta_2}$

   (b) $\Delta_1[del\_Sync] \equiv \Delta_2[del\_Sync] \Longleftrightarrow SyncE^{del}_{\Delta_1} = SyncE^{del}_{\Delta_2}$

   (c) $\Delta_1[ins\_Loop] \equiv \Delta_2[ins\_Loop] \Longleftrightarrow LoopE^{add}_{\Delta_1} = LoopE^{add}_{\Delta_2}$

   (d) $\Delta_1[del\_Loop] \equiv \Delta_2[del\_Loop] \Longleftrightarrow LoopE^{del}_{\Delta_1} = LoopE^{del}_{\Delta_2}$

3. **Change Operations on Data Flow:**
   $\Delta_1[data] \equiv \Delta_2[data] \Longleftrightarrow$
   $(D^{add}_{\Delta_1} = D^{add}_{\Delta_2} \wedge D^{del}_{\Delta_1} = D^{del}_{\Delta_2}) \wedge$
   $(DataE^{add}_{\Delta_1} = DataE^{add}_{\Delta_2} \wedge DataE^{add}_{\Delta_1} = DataE^{add}_{\Delta_2})$

4. **Change Operations on Attributes:**
   $\Delta_1[attrChange] \equiv \Delta_2[attrChange] \Longleftrightarrow ChangedAttr_{\Delta_1} = ChangedAttr_{\Delta_2}$

   (Φ)

We show that:
$(\Phi) \Longrightarrow S_1 \simeq S_2 \overset{Theorem10}{\Longrightarrow} S_1 \equiv_{trace} S_2$

$(\Phi) \Longrightarrow$
[[$\exists$ bijective mapping f: $N_1 \mapsto N_2$ with
($label$(n) = $label$(f(n)) $\forall$ n $\in N_1$) $\wedge$
($\forall$ e= (u, v) $\in$ CtrlE$_1$: $\exists$ e* = (f(u), f(v)) $\in$ CtrlE$_2$ with $label$(e)= $label$(e*)
$\wedge \forall$ e* = (u*, v*) $\in$ CtrlE$_2$ $\exists$ e = (f^{-1}(u*), f^{-1}(v*)) $\in CtrlE_1$
with $label$(e*) = $label$(e)) $\wedge$
($\forall$ e= (u, v) $\in$ SyncE$_1$: $\exists$ e* = (f(u), f(v)) $\in$ SyncE$_2$ with $label$(e)= $label$(e*)
$\wedge \forall$ e* = (u*, v*) $\in$ SyncE$_2$ $\exists$ e = (f^{-1}(u*), f^{-1}(v*)) $\in SyncE_1$
with $label$(e*) = $label$(e))$\wedge$
($\forall$ e= (u, v) $\in$ LoopE$_1$: $\exists$ e* = (f(u), f(v)) $\in$ LoopE$_2$ with $label$(e)= $label$(e*)
$\wedge \forall$ e* = (u*, v*) $\in$ LoopE$_2$ $\exists$ e = (f^{-1}(u*), f^{-1}(v*)) $\in LoopE_1$
with $label$(e*) = $label$(e))]$\wedge$

[$\exists$ bijective mapping g: $D_1 \mapsto D_2$ with
($label$(d) = $label$(g(d)) $\forall$ d $\in D_1$) $\wedge$
($\forall$ dE = (n, d, mode) $\in$ DataE$_1$, n $\in N_1$:
$\exists$ dE* = (g(n), g(d), mode) $\in$ DataE$_2$: $label$(dE) = $label$(dE*)
$\wedge \forall$ dE* = (n*, d*, mode) $\in$ DataE$_2$
$\exists$ dE = (g^{-1}(n*), g^{-1}(d*), mode) $\in DataE_1$: $label$(dE*) = $label$(dE)]]

Without loss of generality we show the direction from S_1 to S_2 (the other direction from S_2 to S_1 runs analogously).

- to show: $\forall n \in N_1 : \exists f(n) \in N_2$ with label(n) = label(f(n)) (ξ_1)
  $N_1 = (N \cup N^{add}_{\Delta_1}) \setminus N^{del}_{\Delta_1}$, $N_2 = (N \cup N^{add}_{\Delta_2}) \setminus N^{del}_{\Delta_2} \Longrightarrow$
  $N_1 = N_2$ and consequently ξ_1 hold.

- to show: $\forall e = (u, v) \in CtrlE_1 : \exists e* = (f(u), f(v)) \in CtrlE_2$ with label(e) = label(e*) (ξ_2)
  $CtrlE_1 = (CtrlE \cup CtrlE^{add}_{\Delta_1}) \setminus CtrlE^{del}_{\Delta_1}$, $CtrlE_2 = (CtrlE \cup CtrlE^{add}_{\Delta_2}) \setminus CtrlE^{del}_{\Delta_2} \Longrightarrow$
  $CtrlE_1 = CtrlE_2$ and consequently ξ_2 hold.

- to show: $\forall e = (u, v) \in SyncE_1 : \exists e* = (f(u), f(v)) \in SyncE_2$ with label(e) = label(e*) (ξ_3)
  $SyncE_1 = (SyncE \cup SyncE^{add}_{\Delta_1}) \setminus SyncE^{del}_{\Delta_1}$, $SyncE_2 = (SyncE \cup SyncE^{add}_{\Delta_2}) \setminus SyncE^{del}_{\Delta_2} \Longrightarrow$
  $SyncE_1 = SyncE_2$ and consequently ξ_3 hold.

- to show: $\forall e = (u, v) \in LoopE_1 : \exists e* = (f(u), f(v)) \in LoopE_2$ with label(e) = label(e*) (ξ_4)
  $LoopE_1 = (LoopE \cup LoopE^{add}_{\Delta_1}) \setminus LoopE^{del}_{\Delta_1}$, $LoopE_2 = (LoopE \cup LoopE^{add}_{\Delta_2}) \setminus LoopE^{del}_{\Delta_2} \Longrightarrow$
  $LoopE_1 = LoopE_2$ and consequently ξ_4 hold.

- to show: $\forall d \in D_1 : \exists g(n) \in D_2$ with label(n) = label(g(n)) (ξ_5)
  $D_1 = (D \cup D^{add}_{\Delta_1}) \setminus D^{del}_{\Delta_1}$, $D_2 = (D \cup D^{add}_{\Delta_2}) \setminus D^{del}_{\Delta_2} \Longrightarrow$
  $D_1 = D_2$ and consequently ξ_5 hold.

- to show: $\forall dE = (n, d, mode) \in DataE_1 : \exists dE* = (g(n), g(d), mode) \in DataE_2$ with label(n) = label(g(n)) (ξ_6)
  $DataE_1 = (DataE \cup DataE^{add}_{\Delta_1}) \setminus DataE^{del}_{\Delta_1}$, $DataE_2 = (DataE \cup DataE^{add}_{\Delta_2}) \setminus DataE^{del}_{\Delta_2} \Longrightarrow$
  $DataE_1 = DataE_2$ and consequently ξ_6 hold. □

Appendix D

Algorithms

Algorithm 9 Purge(S, S', $\Delta = (op_1, \ldots, op_n)$, $N_\Delta^{add}, N_\Delta^{del}, SyncE_\Delta^{add}, SyncE_\Delta^{del}$, $D_\Delta^{add}, D_\Delta^{del}$, $DataE_\Delta^{add}, DataE_\Delta^{del}$) $\longrightarrow$ (Δ^{purged}, N_Δ^{move}, $ChangedAttr_\Delta$)

```
//Initialization

VisitedActivities:=∅;
VisitedSyncEdges:=∅;
VisitedDataElements:∅;
VisitedDataEdges:=∅;
VisitedAttributes:=∅;
Δ^purged := ∅; N_Δ^move := ∅, ChangedAttr_Δ := ∅
//Scan Change Log in Reverse Direction

for i = n to 1 do

  switch (op_i)

  //Insertion Of Activities
   case serialInsertActivity(S, X, src, dest)):
     if (X ∉ VisitedActivities)
       VisitedActivities = VisitedActivities ∪ {X}; //X not considered so far
      if(X ∉ N_Δ^add) //X actually not inserted ⟶ hidden move
        if (src ≠ c_pred(S, X) ∧ dest ≠ c_succ(S, X))//We have to check whether X
         //has been moved to another position than its original one.
           Δ^purged.addFirst(serialMoveActivity(S, X, src, dest))//This function
           //adds an entry to the first position of the change transaction.;
           N_Δ^move = N_Δ^move ∪ {X};
        fi
      else
         Δ^purged.addFirst(serialInsertActivity(S, X, src, dest));
      fi
    fi
  break
```

```
case parallelInsertActivity(S, X, (b,e)):
  if (X ∉ VisitedActivities)
    VisitedActivities = VisitedActivities ∪ {X}; //X not considered so far
    if(X ∉ N_Δ^add) //X actually not inserted ⟶ hidden move
       Δ^purged.addFirst(parallelMoveActivity(S, X, (b, e)))//For parallel and
      //branch insert operations it is not necessary
      // to compare original and destination context
       N_Δ^move = N_Δ^move ∪ {X};
    else
       Δ^purged.addFirst(parallelInsertActivity(S, X, (b, e)));
    fi
  fi
break;

case branchInsertActivity(S, X, split, join, sc):
   if (X ∉ VisitedActivities)
     VisitedActivities = VisitedActivities ∪ {X}; //X not considered so far
    if(X ∉ N_Δ^add) //X actually not inserted ⟶ hidden move
       Δ^purged.addFirst(branchMoveActivity(S, X, split, join, sc));
       N_Δ^move = N_Δ^move ∪ {X};
    else
     Δ^purged.addFirst(branchInsertActivity(S, X, split, join, sc));
     fi
  fi
break

//Insertion Of Blocks

case serialInsertBlock(S, block, src, dest):
  forall (X ∈ block \ VisitedActivities) do
     VisitedActivities = VisitedActivities ∪ {X}; //X not considered so far
     if(X ∉ N_Δ^add) //X actually not inserted ⟶ hidden move
       Δ^purged.addFirst(serialMoveActivity(S, X, src, dest));
       N_Δ^move = N_Δ^move ∪ {X};
     else
       Δ^purged.addFirst(serialInsertActivity(S, X, src, dest));
     fi
   od
break
```

```
  case parallelInsertBlock(S, block, (b,e)):
     forall (X ∈ block \ VisitedActivities) do
       VisitedActivities = VisitedActivities ∪ {X}; //X not considered so far
      if(X ∉ N_Δ^add) //X actually not inserted ⟶ hidden move
         Δ^purged.addFirst(parallelMoveActivity(S, X, (b, e)));
         N_Δ^move = N_Δ^move ∪ {X};
      else
         Δ^purged.addFirst(parallelInsertActivity(S, X, (b, e)));
      fi
    od
  break

  case branchInsertBlock(S, block, split, join, sc):
     forall (X ∈ block \ VisitedActivities) do
       VisitedActivities = VisitedActivities ∪ {X}; //X not considered so far
      if(X ∉ N_Δ^add) //X actually not inserted ⟶ hidden move
         Δ^purged.addFirst(branchMoveActivity(S, X, split, join, sc));
         N_Δ^move = N_Δ^move ∪ {X};
      else
         Δ^purged.addFirst(branchInsertActivity(S, X, split, join, sc));
      fi
    od
  break

//Moving Activities

  case serialMoveActivity(S, X, src, dest):
     if (X ∉ VisitedActivities)
       VisitedActivities = VisitedActivities ∪ {X};
      if (X ∈ N_Δ^add) //X has been newly inserted and moved afterwards
         Δ^purged.addFirst(serialInsertActivity(S, X, src, dest));
      else
        if (src ≠ c_pred(S, X) ∧ dest ≠ c_succ(S, X))
           Δ^purged.addFirst(serialMoveActivity(S, X, src, dest));
           N_Δ^move = N_Δ^move ∪ {X};
        fi
      fi
    fi
  break

  case parallelMoveActivity(S, X, (b, e)):
     if (X ∉ VisitedActivities)
       VisitedActivities = VisitedActivities ∪ {X};
      if(X ∈ N_Δ^add) //X has been newly inserted and moved afterwards
         Δ^purged.addFirst(parallelInsertActivity(S, X, (b, e)));
      else
         Δ^purged.addFirst(parallelMoveActivity(S, X, (b, e)));
           N_Δ^move = N_Δ^move ∪ {X};
      fi
    fi
  break
```

```
  case branchMoveActivity(S, X, split, join, sc):
    if (X ∉ VisitedActivities)
       VisitedActivities = VisitedActivities ∪ {X};
      if(X ∈ N_Δ^add) //X has been newly inserted and moved afterwards
         Δ^purged.addFirst(branchInsertActivity(S, X, split, join, sc));
      else
         Δ^purged.addFirst(branchMoveActivity(S, X, split, join, sc));
           N_Δ^move = N_Δ^move ∪ {X};
      fi
    fi
  break

//Moving Blocks

  case serialMoveBlock(S, block, src, dest):
     forall (X ∈ block \ VisitedActivities) do
       VisitedActivities = VisitedActivities ∪ {X};
      if (X ∈ N_Δ^add) //X has been newly inserted and moved afterwards
         Δ^purged.addFirst(serialInsertActivity(S, X, src, dest));
      else
         Δ^purged.addFirst(serialMoveActivity(S, X, src, dest));
         N_Δ^move = N_Δ^move ∪ {X}
      fi
    od
  break

  case parallelMoveBlock(S, block, (b, e)):
    forall (X ∈ block \ VisitedActivities) do
       VisitedActivities = VisitedActivities ∪ {X};
      if (X ∈ N_Δ^add) //X has been newly inserted and moved afterwards
         Δ^purged.addFirst(parallelInsertActivity(S, X, (b, e)));
      else
         Δ^purged.addFirst(parallelMoveActivity(S, X, (b, e)));
         N_Δ^move = N_Δ^move ∪ {X};
      fi
    od
  break

  case branchMoveBlock(S, block, split, join, sc):
     forall (X ∈ block \ VisitedActivities) do
       VisitedActivities = VisitedActivities ∪ {X};
      if (X ∈ N_Δ^add) //X has been newly inserted and moved afterwards
         Δ^purged.addFirst(branchInsertActivity(S, X, (b, e)));
      else
         Δ^purged.addFirst(branchMoveActivity(S, X, (b, e)));
         N_Δ^move = N_Δ^move ∪ {X};
      fi
    od
  break
```

```
//Deleting Activities

  case deleteActivity(S, X):
    if (X ∉ VisitedActivities)
       VisitedActivities = VisitedActivities ∪ {X};
      if(X ∈ N_Δ^del)
         Δ^purged.addFirst(deleteActivity(S, X));
      fi
    fi
  break

//Deleting Blocks

  case deleteBlock(S, block):
    forall (X ∈ block \ VisitedActivities) do
       VisitedActivities = VisitedActivities ∪ {X};
      if(X ∈ N_Δ^del)
         Δ^purged.addFirst(deleteActivity(S, X));
      fi
    od
  break

//Inserting And Deleting Sync Edges

  case insertSyncEdge(S, edge):
    if (edge ∉ VisitedSyncEdges)
       VisitedSyncEdges = VisitedSyncEdges ∪ {edge};
      if(edge ∈ SyncE_Δ^add)
         Δ^purged.addFirst(insertSyncEdge(S, edge));
      fi
    fi
  break

  case deleteSyncEdge(S, edge):
    if (edge ∉ VisitedSyncEdges)
       VisitedSyncEdges = VisitedSyncEdges ∪ {edge};
      if(edge ∈ SyncE_Δ^del)
         Δ^purged.addFirst(deleteSyncEdge(S, edge));
      fi
    fi
  break
```

```
//Inserting And Deleting Data Elements

  case addDataElements(S, dE, dom, defVal):
    forall (d ∉ dE ∩ VisitedDataElements) do
       VisitedDataElements = VisitedDataElements ∪ {d};
      if(d ∈ D_Δ^add)
         Δ^purged.addFirst(addDataElement(S, d, dom, defVal));
      fi
    od
  break

  caes deleteDataElements(S, dE ):
    forall (d ∉ dE ∩ VisitedDataElements) do
       VisitedDataElements = VisitedDataElements ∪ {d};
      if(d ∈ D_Δ^del)
         Δ^purged.addFirst(deleteDataElements(S, d));
      fi
    od
  break

//Inserting And Deleting Data Edges

  case addDataEdges(S, dE):
    forall (d ∉ dE ∩ VisitedDataEdges) do
       VisitedDataEdges = VisitedDataEdges ∪ {d};
      if(d ∈ DataE_Δ^add)
         Δ^purged.addFirst(addDataEdges(S, d, dom, defVal));
      fi
    od
  break

  case deleteDataEdges(S, dE):
    forall (d ∉ dE ∩ VisitedDataEdges) do
       VisitedDataEdges = VisitedDataEdges ∪ {d};
      if(d ∈ DataE_Δ^del)
         Δ^purged.addFirst(deleteDataEdges(S, d));
      fi
    od
  break
//Changing Activity And Edge Attributes

  case changeActivityAttribute(S, X, attr, nV):
    if (X ∉ VisitedAttributes)
       VisitedAttributes = VisitedAttributes∪ {X};
       Δ^purged.addFirst(changeActivityAttribute(S, X, attr, nV));
       ChangedAttr_Δ = ChangedAttr_Δ ∪ {(X, attr, nV)};
    fi
  break

  case changeEdgeAttribute(S, edge, attr, nV):
    if (edge ∉ VisitedAttributes)
       VisitedAttributes = VisitedAttributes∪ {edge};
       Δ^purged.addFirst(changeEdgeAttribute(S, edge, attr, nV));
       ChangedAttr_Δ = ChangedAttr_Δ ∪ {(edge, attr, nV)};
    fi
  break

return (Δ^purged, N_Δ^move, ChangedAttr_Δ);
```

Algorithm 10 (Calculating Instance–Specific Change $\Delta_I(S') := \Delta_I \setminus \Delta_T$) *Let S be a (correct) process type schema and $I = (S, \Delta_I, \ldots)$ a process instance with instance–specific schema $S_I := S + \Delta_I$. Let further Δ_T be a process type change which transforms S into another (correct) process type schema S'. Assume that the degree of overlap between Δ_T and Δ_I has been determined. Then instance–specific change $\Delta_I(S')$ resulting after re–linking I to S' can be determined as follows:*

CalcBias(S, S', S_I, $N^{add}_{\Delta_T}, N^{del}_{\Delta_T}, N^{add}_{\Delta_I}, N^{del}_{\Delta_I}, N^{move}_{\Delta_T}$, $N^{move}_{\Delta_I}$, $SyncE^{add}_{\Delta_T}, SyncE^{del}_{\Delta_T} SyncE^{add}_{\Delta_I}, SyncE^{del}_{\Delta_I}$
$LoopE^{add}_{\Delta_T}, LoopE^{del}_{\Delta_T} LoopE^{add}_{\Delta_I}, LoopE^{del}_{\Delta_I}$, $DataE^{add}_{\Delta_T}, DataE^{del}_{\Delta_T} DataE^{add}_{\Delta_I}, DataE^{del}_{\Delta_I}$, $D^{add}_{\Delta_T}, D^{del}_{\Delta_T}$,
$D^{add}_{\Delta_I}, D^{del}_{\Delta_I}$, $E^{conc\_context}_{\Delta_T}[ins], E^{conc\_context}_{\Delta_T}[move]$, $E^{conc\_context}_{\Delta_I}[ins], E^{conc\_context}_{\Delta_I}[move]$,) $\longrightarrow$ ($\Delta_I(S')$)

```
//Initialization

Δ_I(S') := ∅;

//Δ_T is subsumption equivalent with Δ_I, i.e., Δ_T ≺ Δ_I

if (Δ_T ≺ Δ_I)
  find Add:=N^add_ΔI \ N^add_ΔT;
  Single:={X ∈ Add | ∄B ∈ OrderIns(S_I, Δ_I) with (X, Y) ∈ B ∨ (Y, X) ∈ B};
  Row:= {< X_1, ..., X_n > |{X_1, ..., X_n} ∈ AnchorGroupsIns(S, Δ_I) ∧ ∃(X_1, X_2), ..., (X_{n-1}, X_n) ∈ CtrlE_I};
   forall X ∈ Single do
     Δ_I(S') := Δ_I(S') ∪ {insertBetweenNodeSets(S', X, {c_pred(S_I, X)}, {c_succ(S_I, X})};
  od
  forall B =< X_1, ..., X_n >∈ Row do
    find (A_1, X_1, A_2) ∈ AnchorIns(S, Δ_I);
     Δ_I(S') := Δ_I(S') ∪ {insertBetweenNodeSets(S', X_1, A_1, A_2)};
    for (i = 2 to n) do
       Δ_I(S') := Δ_I(S') ∪ {serialInsertActivity(S', X_i, A_{i-1}, A_2)};
    od
  od

  forall X ∈ N^del_ΔI \ N^del_ΔT do
     Δ_I(S') := Δ_I(S') ∪ {deleteActivity(S', X);
  od

  find Move:=N^move_ΔI \ N^move_ΔT;
  Single:={X ∈ Move| ∄B ∈ OrderMove(S_I, Δ_I) with (X, Y) ∈ B ∨ (Y, X) ∈ B};
  Row:= {< X_1, ..., X_n > |{X_1, ..., X_n} ∈ AnchorGroupsMove(S, Δ_I) ∧ ∃(X_1, X_2), ..., (X_{n-1}, X_n) ∈ CtrlE_I};
   forall X ∈ Single do
     Δ_I(S') := Δ_I(S') ∪ {moveBetweenNodeSets(S', X, {c_pred(S_I, X)}, {c_succ(S_I, X})}[1];
  od
  forall B =< X_1, ..., X_n >∈ Row do
    find (A_1, X_1, A_2) ∈ AnchorMove(S, Δ_I);
     Δ_I(S') := Δ_I(S') ∪ {moveBetweenNodeSets(S', X_1, A_1, A_2)};
    for (i = 2 to n) do
       Δ_I(S') := Δ_I(S') ∪ {serialMoveActivity(S', X_i, A_{i-1}, A_2)};
    od
  od

  forall edge ∈ SyncE^add_ΔI \ SyncE^add_ΔT do
     Δ_I(S') := Δ_I(S') ∪ {insertSyncEdge(S', edge)};
  od

  forall edge ∈ SyncE^del_ΔI \ SyncE^del_ΔT do
     Δ_I(S') := Δ_I(S') ∪ {deleteSyncEdge(S', edge)};
  od
```

```
  forall edge ∈ LoopE^add_ΔI \ LoopE^add_ΔT do
     Δ_I(S'):=Δ_I(S') ∪ {insertLoopEdge(S', edge, ...)};
  od

  forall edge = (L_S, L_E) ∈ LoopE^del_ΔI \ LoopE^del_ΔT do
     Δ_I(S'):=Δ_I(S') ∪ {deleteBlock(S', (L_S, L_E))};
  od

  forall d ∈ D^add_ΔI \ D^add_ΔT do
     Δ_I(S'):=Δ_I(S') ∪ {addDataElements(S', {d})};
  od

  forall edge ∈ DataE^add_ΔI \ DataE^add_ΔT do
     Δ_I(S'):=Δ_I(S') ∪ {addDataEdges(S', {edge})};
  od

  forall edge ∈ DataE^del_ΔI \ DataE^del_ΔT do
     Δ_I(S'):=Δ_I(S') ∪ {deleteDataEdges(S', {edge})};
  od

  forall d =∈ D^del_ΔI \ D^del_ΔT do
     Δ_I(S'):=Δ_I(S') ∪ {deleteDataElements(S', {d}))};
  od

  inherit attribute changing operations from Δ_I which are not present in Δ_I;

else

//Δ_I is partially equivalent with Δ_T regarding deletion of activities and sync edges

if (Δ_I[projection] ◊ Δ_T[projection]∧ projection ∈ {del_Act, del_Sync})
  forall X ∈ N^del_ΔI \ N^del_ΔT do
     Δ_I(S'):=Δ_I(S') ∪ {deleteActivity(S', X)};
  od
  forall edge ∈ SyncE^del_ΔI \ SyncE^del_ΔT do
     Δ_I(S'):=Δ_I(S') ∪ {deleteSyncEdge(S', edge)};
  od

else

//Δ_I and Δ_T have a concurrent target context, i.e., Δ_T ⅄ Δ_I

if (Δ_T ⅄ Δ_I)
   forall (s, d) ∈ (E^conc_context_ΔT[ins] ∪ E^conc_context_ΔT[move]) ∩ (E^conc_context_ΔI[ins] ∪ E^conc_context_ΔI[move]) do
    determine concerned activity X ∈ N^add_ΔI ∪ N^move_ΔI;
    if (X ∈ N^add_ΔI)
       Δ_I(S'):=Δ_I(S') ∪ {insertBetweenNodeSets(S', X, {s}, {d});
    else if (X ∈ N^move_ΔI)
       Δ_I(S'):=Δ_I(S') ∪ {moveBetweenNodeSets(S', X, {s}, {d})[1];
    fi
  od
fi

return Δ_I(S');
[1]The high-level change moveBetweenNodeSets(S, X, ...) can be defined analogously to high-level change
insertBetweenNodeSets(S, ...), i.e., by first moving X parallel to a minimal block around the
defined node sets and then setting the desired order relations via sync edges.
```

Algorithm 11 (Calculating $\Delta_T(S_I) := \Delta_T \setminus \Delta_I$) *Let S be a (correct) process type schema and $I = (S, \Delta_I, \ldots)$ a process instance with instance–specific schema $S_I := S + \Delta_I$. Let further Δ_T be a process type change which transforms S into another (correct) process type schema S'. Assume that $\Delta_I \prec \Delta_T$ holds. Then the parts of Δ_T for which state–related compliance has to be checked on S_I (notation: $(\Delta_T(S_I) := \Delta_T \setminus \Delta_I)$ can be determined as follows:*

CalcStateChecks(S, S', S_I, $N^{add}_{\Delta_T}, N^{del}_{\Delta_T}, N^{add}_{\Delta_I}, N^{del}_{\Delta_I}, N^{move}_{\Delta_T}$, $N^{move}_{\Delta_I}$, $SyncE^{add}_{\Delta_T}, SyncE^{del}_{\Delta_T} SyncE^{add}_{\Delta_I}, SyncE^{del}_{\Delta_I}$
$LoopE^{add}_{\Delta_T}, LoopE^{del}_{\Delta_T} LoopE^{add}_{\Delta_I}, LoopE^{del}_{\Delta_I}$, $DataE^{add}_{\Delta_T}, DataE^{del}_{\Delta_T} DataE^{add}_{\Delta_I}, DataE^{del}_{\Delta_I}$, $D^{add}_{\Delta_T}, D^{del}_{\Delta_T}$, $D^{add}_{\Delta_I}, D^{del}_{\Delta_I}$,) $\longrightarrow$ $(\Delta_T(S_I))$

```
//Initialization

ΔT(SI) := ∅;

find Add:=N^add_ΔT \ N^add_ΔI;
Single:={X ∈ Add | ∄B ∈ OrderIns(S', ΔT) with (X, Y) ∈ B ∨ (Y, X) ∈ B};
Row:= {< X1, ..., Xn > |{X1, ..., Xn} ∈ AnchorGroupsIns(S, ΔT) ∧ ∃(X1, X2), ..., (Xn−1, Xn) ∈ CtrlE'};
  forall X ∈ Single do
     ΔT(SI) := ΔT(SI)∪ {insertBetweenNodeSets(SI, X, {c_pred(S', X)}, {c_succ(S', X})};
  od
  forall B =< X1, ..., Xn >∈ Row do
    find (A1, X1, A2) ∈ AnchorIns(S, ΔT);
     ΔT(SI) := ΔT(SI)∪ {insertBetweenNodeSets(SI, X1, A1, A2)};
    for (i = 2 to n) do
       ΔT(SI) := ΔT(SI)∪ {serialInsertActivity(SI, Xi, Ai−1, A2)};
    od
  od

  forall X ∈ N^del_ΔT \ N^del_ΔI do
     ΔT(SI):=ΔT(SI) ∪ {deleteActivity(SI, X);
  od

  find Move:=N^move_ΔT \ N^move_ΔI;
  Single:={X ∈ Move| ∄B ∈ OrderMove(S', ΔT) with (X, Y) ∈ B ∨ (Y, X) ∈ B};
  Row:= {< X1, ..., Xn > |{X1, ..., Xn} ∈ AnchorGroupsMove(S, ΔT) ∧ ∃(X1, X2), ..., (Xn−1, Xn) ∈ CtrlE'};
  forall X ∈ Single do
     ΔT(SI) := ΔT(SI)∪ {moveBetweenNodeSets(SI, X, {c_pred(S', X)}, {c_succ(S', X})};
  od
  forall B =< X1, ..., Xn >∈ Row do
    find (A1, X1, A2) ∈ AnchorMove(S, ΔT);
     ΔT(SI) := ΔT(SI)∪ {moveBetweenNodeSets(SI, X1, A1, A2)};
    for (i = 2 to n) do
       ΔT(SI) := ΔT(SI)∪ {serialMoveActivity(SI, Xi, Ai−1, A2)};
    od
  od

  forall edge ∈ SyncE^add_ΔT \ SyncE^add_ΔI do
     ΔT(SI):=ΔT(SI) ∪ {insertSyncEdge(SI, edge)};
  od

  forall edge ∈ SyncE^del_ΔT \ SyncE^del_ΔI do
     ΔT(SI):=ΔT(SI) ∪ {deleteSyncEdge(SI, edge)};
  od
```

```
  forall edge ∈ LoopE^add_ΔT \ LoopE^add_ΔI do
     Δ_T(S_I):=Δ_T(S_I) ∪ {insertLoopEdge(S_I, edge, ...)};
  od

  forall edge = (L_S, L_E) ∈ LoopE^del_ΔT \ LoopE^del_ΔI do
     Δ_T(S_I):=Δ_T(S_I) ∪ {deleteBlock(S_I, (L_S, L_E))};
  od

  forall d ∈ D^add_ΔT \ D^add_ΔI do
     Δ_T(S_I):=Δ_T(S_I) ∪ {addDataElements(S_I, {d})};
  od

  forall edge ∈ DataE^add_ΔT \ DataE^add_ΔI do
     Δ_T(S_I):=Δ_T(S_I) ∪ {addDataEdges(S_I, {edge})};
  od

  forall edge ∈ DataE^del_ΔT \ DataE^del_ΔI do
     Δ_T(S_I):=Δ_T(S_I) ∪ {deleteDataEdges(S_I, {edge})};
  od

  forall d =∈ D^del_ΔT \ D^del_ΔI do
     Δ_T(S_I):=Δ_T(S_I) ∪ {deleteDataElements(S_I, {d})};
  od

  inherit attribute changing operations from Δ_I which are not present in Δ_I;

return Δ_T(S_I);
```

Zusammenfassung

Für Unternehmen gewinnt die elektronische Unterstützung ihrer Geschäftsprozesse zunehmend an Bedeutung. Sowohl für traditionelle Applikationssysteme (z. B. ERP-Systeme) als auch für Anwendungen im sich rasch entwickelnden E-Business-Bereich (z. B. E-Procurement, Supply Chain Management) wird von Anwenderseite in verstärktem Maße eine aktive Prozessunterstützung gewünscht. Dasselbe trifft auf Technologien zur unternehmensweiten und -übergreifenden Anwendungsintegration zu. All diese Entwicklungen haben die Prozess-Management-Idee – Trennung von Prozesslogik und Anwendungscode sowie explizite Steuerung der Prozesse durch ein *Prozess-Management-System (PMS)* – wieder stark in den Fokus des allgemeinen Interesses gerückt.

Sollen PMS in umfassender Weise für die rechnerbasierte Verwaltung und Steuerung von Geschäftsprozessen einsetzbar sein, müssen die von ihnen verwalteten Prozessschemata und Prozessinstanzen bei Bedarf rasch anpassbar sein. Solche Änderungen können einen Prozesstyp (bzw. sein Schema) als Ganzes oder auch nur einzelne Instanzen betreffen. Bei Änderungen auf Prozesstyp-Ebene wird man in der Regel fordern, dass die auf Basis des alten Prozessschemas erzeugten Instanzen auch nach Änderung dieses Schemas ungestört weiterlaufen können. Dies lässt sich z. B. durch geeignete Versionskonzepte erreichen. Dieser einfache Ansatz ist für Prozesse kurzer Dauer meist ausreichend sein, wirft aber im Zusammenhang mit langlaufenden Prozessen, wie sie z. B. im Krankenhaus-, Engineering- oder Finanz-Bereich auftreten, Probleme auf. Bei dem dann resultierenden „Mix" von Instanzen alter und neuer Form muss gegebenenfalls für längere Zeit ein Durcheinander in Produktion oder Dienstleistungen in Kauf genommen werden. Abgesehen davon ist eine Fortführung der Prozesse auf Grundlage des alten Prozessschemas aus verschiedenen Gründen nicht immer akzeptabel, etwa wenn dadurch gesetzliche Vorschriften oder Geschäftsregeln des Unternehmens (z. B. Behandlungsrichtlinien eines Krankenhauses) verletzt werden.

Aus diesen Gründen besteht von Anwenderseite der Wunsch, die auf Prozesstyp-Ebene festgelegten Änderungen – wo sinnvoll und möglich – auch auf die bereits (vielleicht in großer Zahl) laufenden Prozessinstanzen zu übertragen. Wir sprechen in diesem Zusammenhang auch von der *Propagation* einer Prozesstyp-Änderung auf laufende Prozessinstanzen bzw. von der *Migration verträglicher* Prozessinstanzen auf das geänderte Prozessschema. Dies bei Bedarf zu können, und zwar ohne, dass es dadurch auf Instanzebene in der Folge zu Inkonsistenzen oder Fehlern kommt, ist ungemein wichtig, wenn ein PMS breit und umfassend einsetzbar sein soll.

Neben Änderungen auf Prozesstyp-Ebene muss ein flexibles PMS auch die (Ad-hoc) Modifikation einzelner Prozessinstanzen zur Laufzeit erlauben. Kommt es dann nachfolgend zu einer Prozesstyp-Änderung stellt sich die grundlegende Frage, ob und wann die bereits individuell modifizierten Prozessinstanzen auf das geänderte Prozessschema migriert werden können. Das bedeutet dann, dass ein flexibles Prozess-Management-System auch das *Zusammenspiel von Prozesstyp- und Prozessinstanz-Änderungen* adäquat unterstützen muss.

Heutige auf dem Markt verfügbare PMS erlauben es jedoch entweder gar nicht, die Änderungen eines Prozesstyps auf bereits laufende Prozessinstanzen zu übertragen oder aber dies kann in der Folge zu Inkonsistenzen oder gar Systemabstürzen führen. Dieser Mangel ist ein wesentlicher Grund für die immer noch geringe Verbreitung dieser Systeme. Auch Ansätze aus der Forschung springen in vielerlei Hinsicht zu kurz, etwa hinsichtlich Benutzerfreundlichkeit oder Effizienz. Außerdem existiert weder ein kommerzielles System noch ein theoretischer Ansatz, der das Zusammenspiel von Prozesstyp- und Prozessinstanz-Änderungen erlaubt.

Ziel dieser Arbeit ist es, ein umfassendes formales Rahmenwerk für die Unterstützung von Prozesstyp- und Prozessinstanz-Änderungen im laufenden Betrieb zu erarbeiten. Darauf basierend sollen Prozesstyp-Änderungen in korrekter und effizienter Weise auf die möglicherweise große Zahl von sich in Ausführung befindlichen Prozessinstanzen propagiert werden. Dies erfordert einerseits entsprechende Korrektheitsüberprüfungen zur Laufzeit, andererseits darf die Performanz des Gesamtsystems nicht wesentlich leiden. Darüber hinaus sollen alle präsentierten Konzepte benutzerfreundlich anwendbar sein.

Grundlegend ist die Unterscheidung zwischen zwei Arten von Prozessinstanzen: solchen, die noch basierend auf ihrem Original-Prozessschema laufen (*unverzerrte Prozessinstanzen*) und solchen, die bereits individuell modifiziert worden sind (*verzerrte Prozessinstanzen*). Das Fundament unserer Betrachtungen bildet der Ansatz zur Migration von unverzerrten Prozessinstanzen auf das geänderte Prozessschema. Kernstück ist hierbei ein von uns (weiter-) entwickeltes, umfassendes Korrektheitskriterium, dessen Gültigkeit durch Auswertung einfacher Bedingungen effizient überprüft werden kann. Zusätzlich werden Algorithmen zur automatischen Anpassung der verträglichen Prozessinstanzen nach ihrer Migration auf das geänderte Prozessschema präsentiert (was für bestimmte Formalismen, z. B. Petri-Netze im Allgemeinen nicht möglich ist).

Ausgehend vom Basisfall der Migration unverzerrter Prozessinstanzen sollen Prozesstyp-Änderungen auch auf verzerrte Prozessinstanzen anwendbar sein. Dazu werden die verzerrten Prozessinstanzen zunächst entlang des Überlappungsgrades zwischen instanz-spezifischer Änderung („Verzerrung") und der Prozesstyp-Änderung klassifiziert. Der Grund hierfür ist, dass Prozessinstanzen, für die instanz-spezifische Änderung und Prozesstyp-Änderung disjunkt sind, eine andere *Migrationsstrategie* erfordern als Prozessinstanzen, für welche diese Änderungen *überlappende* „Effekte" auf das ursprüngliche Prozessschema haben. Für Prozessinstanzen der ersten Kategorie (disjunkte Prozessinstanz- und Prozesstyp-Änderungen), wird eine geeignete Erweiterung des (Basis-) Korrektheitskriteriums vorgenommen, welche neben Statuskonflikten nun auch strukturelle Konflikte berücksichtigt. Um die Existenz von strukturellen Konflikten

effizient auszuschließen, werden schnell überprüfbare Struktur-Konflikttests entwickelt. Schließlich werden geeignete Strategien für die Migration von Prozessinstanzen mit disjunkter Verzerrung präsentiert.

Für Prozessinstanzen, deren Verzerrung die Prozesstyp-Änderung überlappt, wird eine weiterführende Klassifikation entwickelt, die von sich nur teilweise überlappenden bis hin zu äquivalenten Änderungen reicht. Für alle diese Klassen werden (automatische) Migrationsstrategien und Verhaltensregeln zur Benutzerunterstützung erarbeitet. Diese Strategien beinhalten nicht nur das automatische "Umhängen" der verträglichen Prozessinstanzen auf das geänderte Prozessschema, sondern auch die erforderlichen Struktur- und Zustandsanpassungen. Alle Begriffe werden definiert und alle wichtigen Aussagen formal dargestellt und bewiesen. Zudem sind die präsentierten Konzepte prototypisch (im Rahmen eines Demonstrators) implementiert. Sie werden in die Entwicklung des neuen ADEPT2 Prozess-Management-Systems einfließen.