

PhD Dissertation

---



International Doctorate School in Information and  
Communication Technologies

DIT - University of Trento

CORRECTIVE EVOLUTION OF ADAPTABLE  
PROCESS MODELS

Adina Sirbu

Advisor:

Prof. Marco Pistore

Fondazione Bruno Kessler

---

April 2013



*“We shall not cease from exploration  
And the end of all our exploring  
Will be to arrive where we started  
And know the place for the first time.”*

T. S. Eliot

To Uwe



# Acknowledgements

First of all, I would like to thank my advisor Marco Pistore for giving me the opportunity to work under his guidance. Marco's invaluable insights from his broad knowledge and experience, as well as his problem-oriented view (what is the message?) have always kept me focused on achieving concrete results. This thesis would not have been possible without his support, and I am deeply grateful for having had the experience to work together.

Many thanks to Luciano Baresi and Annapaola Marconi for our numerous discussions which helped to shape the research problem presented in this thesis. This work has greatly improved over time due to their many fruitful suggestions and comments. I would like to thank Luciano Baresi, Paolo Giorgini, and Barbara Weber for accepting to act as my doctoral examination committee, for taking the time to read this thesis and to prepare many interesting questions and comments.

I would not have considered starting this journey if not for my experience of working with Jörg Hoffmann, for which I will always be grateful. Under his supervision, I have learned how to be a researcher, not only from our interactions, but also by observing Jörg's integrity in everything he undertakes, as well as his deep commitment to research. His passion for designing scalable techniques and experiments to prove scalability is contagious, and I hope to be able to pass it on when it will be my turn to be a mentor.

---

Special thanks go to my colleagues Asli Zengin, Heorhi Raik, and Piergiorgio Bertoli, who over the years have become close friends. Many thanks go to all my friends, and especially to Nathalie Steinmetz, for their many encouragements during these last years. I know how fortunate I am to have them in my life, and I apologize for not listing them all here.

I would like to express my deepest gratitude to my family, who have always believed in me and have supported me unconditionally in all my endeavors. Last but most importantly, I would like to thank Uwe, my partner in all the adventures, to whom I dedicate this thesis. Throughout many years now, he has always challenged me to be the best version of myself, and it is because of his love, wisdom, and support that I am the person I am today.

# Abstract

*Modeling business processes is a complex and time-consuming task, which can be simplified by allowing process instances to be structurally adapted at runtime, based on context (e.g., by adding or deleting activities). The process model then no longer needs to include a handling procedure for every exception that can occur. Instead, it only needs to include the assumptions under which a successful execution is guaranteed. If a design-time assumption is violated, the exception handling procedure matching the context is selected at runtime. However, if runtime structural adaptation is allowed, the process model may later need to be updated based on the logs of adapted process instances. Evolving the process model is necessary if adapting at run-time is too costly, or if certain adaptations fail and should be avoided.*

*An issue that is insufficiently addressed in the previous work on process evolution is how to evolve a process model and also ensure that the evolved process model continues to achieve the goal of the original model. We refer to the problem of evolving a process model based on selected instance adaptations, such that the evolved model satisfies the goal of the original model, as corrective evolution. Automated techniques for solving the corrective evolution problem are necessary for two reasons. First, the more complex a process model is, the more difficult it is to be changed manually. Second, there is a need to verify that the evolved model satisfies the original goal.*

*To develop automated techniques, we first formalize the problem of corrective evolution. Since we use a graph-based representation of processes,*

---

*a key element in our formal model is the notion of trace. When plugging an instance adaptation at a particular point in the process model, there can be multiple paths in the model for reaching this point. Each of these paths is uniquely identified by a trace, i.e., a recording of the activities executed up to that point. Depending on traces, an instance adaptation can be used to correct the process model in three different ways. A correction is strict if the adaptation should be plugged in on a precise trace, relaxed if on all traces, and relaxed with conditions if on a subset of all traces. The choice is driven by competing concerns: the evolved model should not introduce untested behavior, but it should also remain understandable.*

*Using our formal model, we develop automated techniques for solving the corrective evolution problem in two cases. The first case is also the most restrictive, when all corrections are strict. This case does not require verification, since the process model and adaptations are assumed to satisfy the goal, as long as the adaptations are applied on the corresponding traces. The second case is when corrections are either strict or relaxed. This second case requires verification, and for this reason we develop an automated technique based on planning.*

*We implemented the two automated techniques as tools, which are integrated into a common toolkit. We used this toolkit to evaluate the tradeoffs between applying strict and relaxed corrections on a scenario built on a real event log.*

## **Keywords**

process evolution, adaptable processes, automated verification



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Contributions . . . . .	7
1.1.1	Formal Model . . . . .	7
1.1.2	Automated Process Evolution Approach . . . . .	8
1.1.3	Experimental Results . . . . .	10
1.2	Thesis Outline . . . . .	11
<b>2</b>	<b>State of the Art</b>	<b>13</b>
2.1	Adaptable Process Models . . . . .	13
2.1.1	Design-time Adaptation Approaches . . . . .	15
2.1.2	Runtime Adaptation Approaches . . . . .	19
2.2	Ensuring Process Compliance to Goals and Constraints . . . . .	25
2.2.1	Design-time Compliance . . . . .	26
2.2.2	Runtime Compliance . . . . .	30
2.3	Related Work . . . . .	32
2.3.1	Analyzing Execution and Adaptation Logs . . . . .	32
2.3.2	Managing Process Variants . . . . .	34
2.3.3	Refactoring Process Models . . . . .	36
2.4	Discussion . . . . .	38
<b>3</b>	<b>Evolution Framework</b>	<b>39</b>
3.1	Process-Based Application Lifecycle . . . . .	39

## CONTENTS

---

3.1.1	Execution Phase . . . . .	41
3.1.2	Analysis Phase . . . . .	43
3.1.3	Evolution Phase . . . . .	44
3.2	Process-Based Application Representation . . . . .	47
3.2.1	Domain Objects . . . . .	48
3.2.2	Goals . . . . .	50
3.2.3	Process Models . . . . .	52
3.3	Execution Concepts . . . . .	57
3.3.1	Traces . . . . .	57
3.3.2	Configurations . . . . .	58
3.3.3	Executions . . . . .	59
3.3.4	Goal Satisfaction . . . . .	61
3.4	Adaptation Concepts . . . . .	62
3.4.1	Adaptation Operation . . . . .	63
3.4.2	Adapted Traces and Executions . . . . .	65
3.5	Discussion . . . . .	68
<b>4</b>	<b>Corrective Evolution</b>	<b>71</b>
4.1	Corrections . . . . .	71
4.1.1	Adaptation Plug-in Point and Condition . . . . .	72
4.1.2	Correction Applicability . . . . .	76
4.1.3	Correction Effects . . . . .	78
4.2	Corrective Evolution Problem . . . . .	80
4.2.1	Ordering Corrections . . . . .	81
4.2.2	Problem Definition . . . . .	82
4.2.3	Solution Properties . . . . .	83
4.2.4	Multiple Corrections at Once vs. One at a Time . . . . .	85
4.3	Discussion . . . . .	87

---

<b>5</b>	<b>Encoding into State Transition Systems</b>	<b>89</b>
5.1	Basic STS Notions . . . . .	90
5.2	Encoding the Process Models . . . . .	92
5.3	Encoding the Adaptations . . . . .	95
5.4	Encoding the Partial Traces . . . . .	97
5.5	Encoding the Conditions . . . . .	99
5.6	Encoding the Domain Objects . . . . .	102
5.7	Encoding the Goals . . . . .	103
<b>6</b>	<b>Strict Corrective Evolution</b>	<b>107</b>
6.1	Description of the Approach . . . . .	107
6.2	Correctness of the Approach . . . . .	110
6.2.1	Correspondence Between Executions of a Corrected Model and Runs of the Parallel Product STS . . . .	112
6.2.2	Correctness . . . . .	122
6.2.3	Completeness . . . . .	123
6.3	Discussion . . . . .	124
<b>7</b>	<b>Relaxed Corrective Evolution</b>	<b>127</b>
7.1	Description of the Approach . . . . .	128
7.2	Correctness of the Approach . . . . .	133
7.2.1	Correspondence Between Executions of a Corrected Model and Runs of the Parallel Product STS . . . .	134
7.2.2	Correspondence Between Complete Executions of a Corrected Model and Complete Runs of the Controller	139
7.2.3	Correctness . . . . .	142
7.2.4	Completeness . . . . .	144
7.3	Discussion . . . . .	145

<b>8</b>	<b>Evaluation</b>	<b>147</b>
8.1	Overview of the Event Log . . . . .	148
8.2	Experimental Setup . . . . .	150
8.2.1	Designing the Domain Objects . . . . .	150
8.2.2	Annotating the Activities . . . . .	153
8.2.3	Obtaining the Original Process Model . . . . .	154
8.2.4	Generating Corrections . . . . .	156
8.3	Evaluating Tradeoffs Between Strict and Relaxed Corrections	159
8.3.1	Fitness . . . . .	159
8.3.2	Behavior . . . . .	161
8.3.3	Structure . . . . .	162
8.4	Discussion . . . . .	164
<b>9</b>	<b>Conclusion</b>	<b>167</b>
9.1	Achieved Results . . . . .	167
9.2	Future Directions . . . . .	170
	<b>Bibliography</b>	<b>172</b>

# List of Tables

3.1	Configurations along one execution in the car logistics scenario	61
5.1	Encoding process model elements as STSs . . . . .	93



# List of Figures

3.1	Evolution framework . . . . .	40
3.2	Process-based application representation . . . . .	48
3.3	Modeling elements in the process lifecycle . . . . .	49
3.4	Domain objects in the car logistics scenario . . . . .	51
3.5	Car process model . . . . .	56
3.6	Adaptation operations . . . . .	67
5.1	STS encoding of the process model and adaptations . . . . .	98
5.2	STS encoding of traces: (a) $\pi_1$ ; (b) $\pi_2$ . . . . .	100
5.3	STS encoding of conditions: (a) $\varphi_1$ ; (b) $\varphi_2$ . . . . .	101
5.4	STS encoding of the <i>Car Health</i> domain object . . . . .	104
5.5	STS encoding of a goal statement . . . . .	105
6.1	Strict corrective evolution: solution overview . . . . .	108
6.2	Corrected process model: both corrections are strict . . . . .	110
6.3	Example of a correspondence between an execution and a run	111
7.1	Relaxed corrective evolution: solution overview . . . . .	129
7.2	Corrected process model: correction $C_1$ is relaxed, $C_2$ is strict	132
7.3	Corrected process model: correction $C_1$ is strict, $C_2$ is relaxed	132
7.4	Corrected process model: both corrections are relaxed . . . . .	133
8.1	Mining result on the raw event log . . . . .	149
8.2	Domain objects for the loan application process . . . . .	151

LIST OF FIGURES

---

8.3	A rough process model . . . . .	157
8.4	Fitness: corrections applied (1) individually; (2) incrementally	160
8.5	Behavioral precision: corrections applied (1) individually; (2) incrementally . . . . .	162
8.6	Corrected process models for the 4th correction . . . . .	163
8.7	Structural precision: corrections applied (1) individually; (2) incrementally . . . . .	164
8.8	Corrected process models for the 8th correction . . . . .	165



# Chapter 1

## Introduction

The success of an enterprise depends on its ability to adapt frequently and rapidly in response to changes in the business environment [20]. Since business activities are increasingly integrated with information technologies, this flexibility must also be present in the information system of the enterprise. In response to this challenge, service-oriented computing emerged as a major trend in business engineering and software technology [71, 121]. With service-oriented computing, the main idea is to capture business relevant functionality into independent modular units called services, which expose only the information which is necessary to be used by customers. Services therefore allow to abstract away from implementation details and focus on business requirements and functionalities. Technologies and standards have been developed for building service-based applications as compositions of individual services, possibly offered by different third-party organizations. The flexibility offered by service-based applications comes from the fact that software services can be discovered, selected and composed dynamically, while the application is running.

With service-oriented computing, there came also a surge of interest in Business Process Management (BPM), since business processes can be used to dynamically compose and coordinate services. Traditional work-

flow management technologies were designed to support business processes requiring the coordination of human participants and IT tools and applications. This mainly involved providing support for designing and managing the execution of these business processes, which were seen as static and repetitive. In contrast, in BPM business processes do not necessarily involve human participants, and often span different organisations [118]. Moreover, the support provided with workflow management is enhanced in BPM to include also business process monitoring, diagnosis, and redesign [110].

By using business processes to represent compositions of services, and BPM to manage these business processes, an enterprise can more easily react to changes in the business environment. The support provided by BPM in terms of monitoring and diagnosis can be used to observe the changes in the environment. Then, the ability to implement the runtime adaptation of business processes and the redesign of these processes with relatively little effort, by re-composing existing services, allows to rapidly respond to the observed changes.

### **From process modeling to adaptation**

Systems which support BPM need to be process-aware. In other words, the logic of the process in terms of activities to be performed, together with the control and data flow between these activities, must be modeled explicitly. Process modeling is a complex and time-consuming task, which requires discussions with domain experts and business analysts, as well as a deep knowledge of the process modeling language. The fact that change is easy to implement due to the use of services can also be used for simplifying the process modeling task. In particular, this task can be simplified by allowing process instances to be structurally adapted at runtime, based on context (e.g., by adding or deleting activities). The process model then no longer

---

needs to include a handling procedure for every exceptional situation that can occur. Instead, it only needs to include the assumptions under which a successful execution is guaranteed. If a design-time assumption is violated, an exception is triggered and the exception handling procedure matching the context is selected or constructed at runtime (e.g., [12, 21, 28]).

Being able to adapt process instances at runtime is especially useful if the exceptional situation is unlikely, but must nevertheless be considered in the process model (e.g., due to legal concerns), and including the exception handling procedure would significantly increase the complexity of the process model. Another case where runtime adaptation is useful is if the appropriate exception handling procedure is not known at the time of designing the process model. If process knowledge can be discovered also at runtime, we may want to postpone the decision of how to handle an exceptional situation until it is actually needed, in order to use all the available process or fragment models. Finally, there may be different handling procedures which are suitable for a particular exceptional situation, and the process designer may not know upfront which handling procedure would be successful for the application. Also in this case, we may want to choose an appropriate exception handling procedure at runtime.

### **From adaptation to evolution**

If runtime structural adaptation is allowed, the process model may later need to be updated based on the logs of adapted process instances. If an instance adaptation fails when applied in a certain situation, the process model can be updated to ensure that the failing adaptation is no longer used for that situation. Updating the process model is necessary also when it is too costly to deal at execution time with an exceptional situation, for example because the situation occurs frequently.

Evolving the process model based on process instance adaptations or

process variants is not new, and different aspects of this research problem been addressed in, e.g., [15, 37, 53, 77, 87, 95, 117]. Many approaches deal with the question of how to recommend process model changes based on frequent and successful process instance adaptations, e.g., [77, 87, 117]. There are also approaches, e.g., [53, 87], which generate the evolved process model automatically. Once a process model is evolved, an important problem that arises is how to migrate the running process instances to the new process model. This problem is investigated in, e.g., [15, 86, 95].

Although many approaches have been proposed to tackle the problem of process evolution, an issue that is insufficiently addressed in these approaches is how to evolve a process model and also ensure that the evolved process model continues to comply with the goal and constraints of the original process model. Process models often have to comply with internally defined directives, such as business strategies, as well as externally imposed directives, such as legal regulations and contracts. Enforcing compliance with external directives is particularly challenging, since such directives cannot simply be incorporated in the design of the process models [93]. This issue is also relevant for process evolution, especially since instance adaptations can be used to update a process model in multiple ways, but it can be the case that only some of the evolved process models which result comply with the goal and constraints of the original process model. We refer to the problem of evolving a process model based on selected process instance adaptations such that the evolved model satisfies the goal of the original model as *corrective evolution*.

When plugging an instance adaptation at a particular point in the process model, we need to consider that there can be multiple paths to reach this point in the model. Each of these paths is uniquely identified by a trace, i.e., a recording of the activities executed up to that point. For each instance adaptation that should be plugged into the process model, there

---

are three options, depending on which traces the adaptation should be plugged in. To the best of our knowledge, this distinction based on traces is not present in the literature.

- The first option is to plug in the adaptation at a particular point in the process model, only for the trace corresponding to the process instance where the adaptation was used. We call this option a *strict correction*. The advantage of strict corrections is that the resulting process model will not contain any unknown behavior.
- The second option, or *relaxed correction*, is to plug in the adaptation again at a particular point in the process model, but on all the traces that lead to the specified point. Although more behavior is introduced, the resulting process model should be smaller than the model obtained with strict corrections. The reason is that applying strict corrections requires unfolding the process model, such that the adaptation is plugged in only for the right trace, and this unfolding may require duplicating process steps.
- Plugging in adaptations as relaxed corrections is not always possible, since we also need to ensure that the resulting process model satisfies the goal. The option in this case, *relaxed correction with conditions*, is to plug in the adaptation only for some of the traces leading to the specified point. This option covers the cases in between the first two options.

### **Motivating scenario**

To illustrate the need for corrective evolution, we consider a vehicle logistics scenario, which will be used as a reference scenario throughout this dissertation. This scenario is inspired by the real-life operations of the Bremerhaven sea port. In this scenario, cars arrive from manufacturers

at the sea port, must be unloaded and eventually delivered to a retailer. We focus on the process model for handling the car from the moment it arrives at the terminal, and until it is delivered. First, the car is stored, and remains in the storage area until a delivery order is issued. Then, it undergoes the treatments specified in the delivery order (e.g., painting, installation of electronic equipment, washing). When the treatments are completed, the car is loaded on a truck and delivered to the retailer. The goal of the process is to deliver the car to the retailer in perfect condition.

Cars may be damaged at any point while at the sea port. They may be scratched, may have dents on the surface, flat tires, or low oil levels. The damage can be handled by repairing the car immediately or postponing the repair to a more convenient time. How to handle the damage depends on the context of the car. For example, it may depend on the availability of the resources necessary to repair the car, such as treatment stations or skilled workers. It may also depend on how urgent it is to repair the car, as well as on the urgency of the other cars waiting to be repaired.

To consider at every step in the process that the car can be damaged, as well as all the possible contexts in which the damage can occur, would complicate the process model significantly and obscure the original purpose of the process model. Instead, we specify a constraint on every process step, that the car should not be damaged. If this constraint is violated, the execution of the process instance will be interrupted, and the appropriate handling procedure will be selected based on the context.

When analyzing the execution and adaptation logs, we determine that if the car is damaged while being stored, and other cars are waiting to be repaired, the repair should be postponed since it is not urgent (the car has not yet been ordered), and will only delay other cars. However, if the car is damaged a second time in the storage area, unless many cars are already waiting to be repaired, the repair should not be postponed

anymore. Although the repairs are not urgent, they should be resolved such that the car is delivered on time. We therefore need to evolve the process model, such that the repair is postponed in the first situation, and performed immediately in the second. Moreover, we need to ensure that the evolved process model satisfies the goal to deliver the car in perfect condition.

Depending on the type of the two corrections, we will obtain different process models. If both corrections are strict, the second adaptation is applied only for the trace when the car is damaged a second time, after having postponed the repair for the first damage. If, however, the second adaptation is plugged in as a relaxed correction, it will be applied also for the traces where the car is damaged for the first time.

## 1.1 Contributions

In this dissertation, we extend the existing work on process evolution by investigating the problem of updating a process model based on process instance adaptations, such that the evolved model continues to achieve the goal of the original process model. We propose a formal model for representing processes and their goals, and use this formal model to develop two automated approaches for process evolution. Finally, we evaluate these two approaches on a scenario built on a real-life event log.

### 1.1.1 Formal Model

Our representation of process-based applications is based on two distinctions. First, we distinguish between domain knowledge, or knowledge about properties of entities in the domain, and process knowledge, or knowledge about business logic. Second, we distinguish between concrete, context-dependent knowledge, and knowledge that is common, abstract,

independent of context.

We consider the domain knowledge to be stable and abstract, and model this knowledge using domain objects. Each domain object corresponds to a property of a physical or computational entity in the domain, and consists of a set of possible values and a set of events representing value changes. The stable part of the process knowledge is represented using goals, defined based on the domain objects. Finally, the dynamic and concrete part of process knowledge is represented using adaptable process models. Process models are directed graphs in which the nodes are either steps in the process (i.e., activities) or control connectors. We relate the process models to the domain by defining annotations on process steps based on domain objects.

We use this formal model to investigate the problem of corrective evolution, that is, the problem of updating a process model based on process instance adaptations, such that the evolved model satisfies the goal of the original model. To formally specify that process models satisfy goals, we introduce process execution concepts, and define goal satisfaction criteria based on these concepts. We then introduce process instance adaptation concepts. Based on these concepts, we define evolutionary correction as the operation of changing a process model to include an instance adaptation at a certain point in the model and for a certain condition. Finally, we define the corrective evolution problem as the problem of applying a sequence of corrections, such that the resulting process model satisfies the original goal.

### **1.1.2 Automated Process Evolution Approach**

There are two main reasons for solving the corrective evolution problem automatically. First, the more complex a process model is, the more difficult it becomes to be changed manually, since changing a model requires a thorough understanding of its behavior. How understandable a model



is depends not only on the size of the model, but also on features such as level of concurrency and modularity of the model [18, 64]. Second, unless all corrections are strict, there is a need to verify that the resulting process model satisfies the original goal.

In this dissertation, we develop automated techniques for solving two cases of the corrective evolution problem: when all the corrections are strict, and when the corrections are either strict or relaxed. Unlike the general case, these two cases do not require searching for the right traces where to plug in the adaptations. In the first case, the traces are given as input. In the second case, if a trace is not given, i.e., a correction is relaxed, the adaptation should be plugged in on all traces leading to the plug-in point. The difference between these two cases is that strict corrective evolution also does not require verification, since we assume that both process model and adaptations satisfy the goal, as long as adaptations are applied only on the corresponding traces.

If all corrections are strict, the challenge is not to find a solution to the problem, since a solution to the problem can be constructed naively. However, a naively constructed solution will be unnecessarily large, due to the duplication of many process steps. Therefore, the challenge in this case is to automatically find a solution which contains as few duplicated process steps as possible.

If the corrections are all either strict or relaxed, finding a solution to the problem is no longer trivial. While constructing a corrected process model is still relatively simple, this process model does not necessarily satisfy the goal of the original process model. Therefore, this case adds the new challenge, to automatically verify that every execution of the corrected process model satisfies the goal.

### 1.1.3 Experimental Results

The two automated techniques for solving corrective evolution problems have been implemented into a prototype tool. We used this prototype tool to conduct a series of experiments, in order to show the tradeoffs between applying strict and relaxed corrections.

To conduct our experiments, we set up a scenario based on a real event log. The traces in this log correspond to instances of a single process, an application process for a personal loan or overdraft. To set up our scenario, we recreated all the elements required by our approach: the domain objects, the original process model and its goal, as well as the corrections to the process model. This task was particularly challenging, since the event log we used is an overwhelmingly complex mass of data. The complexity of the log stems from a great variation in how the loan applications are processed, in terms of the order of executing activities, as well as the number of times that certain sequences are re-executed.

We first designed our domain objects based on the event log and the textual descriptions given as input, and used these domain objects to annotate all the activities that appear in the log. We filtered and mined the event log to obtain a rough process model and created the goal for this process model based on the domain objects. Finally, we computed the differences between the model and the traces in the log, and used the most frequent differences to generate strict and relaxed corrections.

We applied strict and relaxed corrections to our rough process model, and compared the resulting corrected models using metrics devised for evaluating process mining results. The experimental results we obtained provide a first confirmation that the choice of correction type influences both the behavior and the structure of the evolved process model. Moreover, we also obtained an indication that the process models which result

by applying relaxed corrections should require less future adaptations than the corresponding process models obtained with strict corrections.

## 1.2 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2 we give an overview of the state of the art in two fields related to our research problem. The first field is that of adaptable process models. Here, we present techniques ranging from design-time to runtime, and from manual to fully automatic, which are meant to increase the flexibility of process models to changes in the execution environment. The second field is that of ensuring the compliance of process models to goals and constraints. Here, we present techniques from the research area of business process management, as well as techniques from the related area of Web service composition. We then focus on approaches which are closely related to the problem of process evolution, and conclude with a discussion about the limitations of the approaches which are closest to the work presented in this thesis.

In Chapter 3, we introduce a framework supporting the evolution of process models based on execution and adaptation histories, and discuss each phase in the lifecycle of a process-based application using this framework. We also introduce the basic elements for modeling an application, as well as concepts related to process execution and adaptation. These concepts are then used in Chapter 4, where we formally define corrections and corrective evolution problems. This chapter also includes a discussion of the challenges involved in automating corrective evolution.

The following chapters present formal techniques and approaches realizing automated corrective evolution. We start with Chapter 5, where we encode all the inputs of a corrective evolution problem into labeled state transition systems. This encoding forms the basis of our automated

mechanism, and is used in the following two chapters for solving two cases of corrective evolution. In Chapter 6, we solve the special case when all corrections are strict. In Chapter 7, we continue with a second and more general case, when corrections are either strict or relaxed.

In Chapter 8, we describe our empirical evaluation of the tradeoffs between applying strict and relaxed correction. Concluding remarks and future directions are discussed in Chapter 9.

# Chapter 2

## State of the Art

The research problem addressed in this thesis, that of ensuring goal compliance of evolving process models, has its roots into two related fields. First, there is the field of adaptable process models, which deals with increasing the flexibility of process models to changes in the execution environment. The second field is that of ensuring the compliance of business process models to goals and constraints. In this chapter, we give an overview of each of these two related fields in Section 2.1, and respectively Section 2.2. Approaches which are closely related to our research problem are presented in detail in Section 2.3.

### 2.1 Adaptable Process Models

A critical success factor of a process-oriented information system is the ability to deal with change in an efficient way [52, 70]. One of the ways to deal with change is to increase the flexibility or adaptability of process models. As pointed out in [81], a business process is designed to maintain an equilibrium between stakeholder requirements, and therefore flexibility of a business process is not only about what should change, but also about what should stay the same. Consequently, process flexibility has been defined as the ability to deal with context changes by adapting the parts

of the process affected by the change, while maintaining the unaffected parts unchanged [97].

Different types of flexibility are required during the lifecycle of a process (see also [83, 118]):

- ability to deal with uncertainty by deferring certain decisions to execution time;
- ability to react to foreseen or un-foreseen exceptions by deviating from the predefined process model;
- ability to evolve process models over time.

To address these flexibility requirements, several competing paradigms have emerged, such as adaptable processes [82, 12], declarative processes [75], and data-driven processes [111, 68]. In the following, we give an overview of approaches designed to achieve process flexibility. Although we will choose examples for each of the competing paradigms, we will focus more on approaches which model processes using an imperative language, since this is also the direction taken in this thesis.

Process adaptation can be performed either at design-time or at runtime. Among design-time adaptation approaches, we distinguish between approaches which provide a set of specialized constructs for embedding the adaptation logic in the process model, and approaches which support the modeling of a reference process model from which different process variants can be configured.

Among runtime adaptation approaches, we distinguish between approaches which support loosely specified process models, where the underspecified parts of the model are filled at runtime, and approaches which allow process instances to be structurally adapted at runtime. A last category of runtime adaptation is that of process evolution approaches, where the process

model is changed as well. Here, we consider also the problem of propagating the process model changes to the running process instances.

### 2.1.1 Design-time Adaptation Approaches

#### Special Modeling Constructs

We first consider the case when the process modeling language contains constructs which provide the ability to incorporate alternative execution paths within the process model at design-time. Here, an important contribution is the work on workflow patterns [109], which provide a means to assess the expressiveness of process modeling languages [124, 92].

The approach in [60] proposes a set of modeling constructs, which allow to route the execution of the process based on contextual information. The built-in adaptation constructs include:

- the *conditional if*, which is used to specify conditional branches guarded by context conditions;
- the *context handler*, which allows to react automatically to the violation of context conditions during the execution of the process;
- the *conditional one-of*, which is used to specify a set of alternative paths, each handling a specific execution context, and which allows to react to context changes by jumping at runtime from one path to another;
- the *cross-context link*, which allows such jumps also when it is not possible or desirable to undo the work done.

For each built-in adaptation construct, the authors provide a graphical representation similar to the Business Process Modeling Notation (BPMN) [1], and define a Business Process Execution Language (BPEL) [72] extension, with a clear syntax and operational semantics.

Another complex form of design-time adaptation is proposed by Modafferi et al. in [67]. The authors introduce the notion of context-sensitive region as a part of the business process which can have different behaviors depending on context. Regions are associated to several configurations, each configuration having an entry condition over the user context. At execution time, the process instance can be migrated from one configuration to another (possibly compensating completed activities), depending on context. To prevent excessive loss of work due to compensation, the notion of migration arcs is introduced, these allowing to switch between configurations.

Wieland et al. [123] extend BPEL in order to explicitly model the influence of the execution context on the workflow. The context-aware workflows are modeled using context events, context queries, and context decisions. With context events, the workflow waits asynchronously for a special environment state. The context queries are used to filter or select objects based on spatial predicates. Lastly, context decisions are used to route the process flow based on internal/external context data.

### **Process Configuration**

For a particular business process, several process variants may exist. There are many situations when process variants are necessary. For example, if the process model should be used in different countries or regions, these countries may have different regulations. Another example is if there are different policies for different seasons, different groups of customers, etc. What is specific to process variants is that for each particular context the steps to be followed are known and well understood.

The modeling and management of process variants is not supported adequately by commercial business process management tools [118]. Such variants have to be specified either as separate process models, or in one



process model, using conditional branching. In both cases, the maintenance efforts are high. Separate, very similar process models are difficult to maintain due to the high redundancy. On the other hand, a process model which includes all the variants will be difficult to maintain and understand due to its size and complexity.

Several directions have been proposed to reduce the efforts necessary for maintaining process variants. One such direction is to use process configuration. With process configuration, the idea is to capture all the process variants into a reference process model, such that at configuration-time (an intermediary phase between design-time and runtime) this reference model can be configured into one of the process variants.

The main characteristic of such reference process models is that they include variation points, in order to distinguish between parts which are common to all variants and parts which are variant-specific. For example, such variation points are represented as configurable nodes in [88, 33, 107], node labels in [85], and adjustment points in Provop [37].

With configurable nodes, the idea is to mark selected activities and control connectors as configurable, and associate them with configuration alternatives. Moreover, it should be possible to define constraints on the set of configurable nodes, in order to restrict the creation of variants. A first example of process configuration using configurable nodes is that of Configurable Event-driven Process Chains (C-EPCs) [88]. With C-EPCs, the EPC functions and decision nodes can be annotated to indicate if they are mandatory or optional. A similar approach is proposed by Gottschalk et al. in [33], where the hiding and blocking operators known for the inheritance of process behavior [106] are applied to the input and output ports of activities. The approach is demonstrated on the concrete process modeling language YAWL [108], resulting in the so-called configurable YAWL (C-YAWL). Also using the hiding and blocking operators, the approach

by Aalst et al. [107] allows to configure processes incrementally, ensuring at every step that the configured variants are correct with respect to both syntactic and behavioral semantics.

Another means to support the configuration of process variants from a reference process model is by annotating the nodes of the reference model. With aggregated EPCs [85], the EPCs are connected to a product hierarchy by annotating the functions and events with simple labels. The labels are then used for extracting from the aggregated EPC the process model corresponding to a given product or product group. A similar approach has been developed in the project PESOA [79, 96], where process models are extended with stereotype annotations to obtain *variant-rich process models*. Through process configuration, each variation point is realized with one or more variants, depending on its type. The approach also makes use of the feature diagrams proposed in [45]. Although feature diagrams were introduced to facilitate configuration in software product lines, they can also be used to configure process models. In particular, the variants can be linked to features from a feature diagram. This way, variants are included or excluded from the process model depending on the selection of features in the feature configuration.

In Provop [37], process variants can be configured from a base process model, by applying groups of change operations, called options. The regions to which changes can be applied are restricted using adjustment points. Similar to [88] and [33], Provop allows to define constraints on the application of different adjustments (e.g., two options can be mutually exclusive). However, different from the process configuration approaches presented so far, for which the reference model needs to contain all the possible behavior, Provop allows also to move or add process elements.

Since configuring process variants is an error-prone task, guiding the end-user in making the configuration decisions can be very useful. This is-

sue is addressed in [50], where the configuration process is guided through questionnaires. Each question refers to a set of domain facts, which can be set to true or false. It is possible to define domain constraints as propositional logic formulas over facts, as well as ordering dependencies between questions and facts. The questions are then posed in an order consistent with these dependencies. For each question, the space of allowed choices is pruned, in order to prevent the violation of domain constraints.

### 2.1.2 Runtime Adaptation Approaches

#### Loosely Specified Process Models

While specifying a process model, we may foresee at design-time that more execution paths will be necessary at runtime, which are not present in the model. Also, it may be the case that the process model cannot be completely specified, and that the activities to be performed at certain points in the process model will become clear only during the execution of a process instance. In both cases, a possible solution is to have loosely specified process models. Such models are incomplete, in the sense that they do not contain enough information to allow them to be executed to completion, but they can be completed at runtime by providing a concrete realization for the underspecified parts.

This concrete realization is selected dynamically in case of late binding (e.g., [3, 14]), modeled in the case of late modeling (e.g., [94]), and composed from fragments in the case of late composition (e.g., [75, 103]). The realization can be performed either before reaching the underspecified part, for example when the process is instantiated (e.g., [103]), or when executing the underspecified part (e.g., [3, 94]). Moreover, the realization can be performed only once for all instances, during the first execution (e.g., [3]), or it can be repeated for every new process instance (e.g., [94, 103]).

The approach in [3] supports *late binding* through small, self-contained processes called 'worklets'. The process model contains enabled workitems, which have associated sets of rules. Worklets are selected and bound at runtime using these rules and the relevant context. Both rules and worklets can be manually added at any time, even during process execution. Also supporting late binding, Burmeister et al. [14] model a business process by defining a goal hierarchy, a list of context variables, and a set of business plans linked to subgoals. Business plans are then selected and executed depending on goals and context. Late binding can also be achieved using an aspect-oriented approach, by annotating process parts with aspects (e.g., [19, 42]). If the aspect matches the context information, the normal execution of a workflow can be interrupted and interweaved with additional business logic.

An approach which supports *late modeling* is presented by Sadiq et al. in [94], based on the notion of 'pockets of flexibility'. During execution, process instances are progressively built based on constraints which specify how and when the fragments can be composed. Validation is used to ensure that the constraint set does not carry conflicts or redundancy, and also that the dynamically modeled fragment conforms to the constraints.

*Late composition* of process fragments can be achieved using a declarative process modeling language. DECLARE [75] is a constraint-based framework supporting multiple declarative languages. A process model specified using a declarative language contains constraints which restrict the task execution options, i.e., the more constraints, the less execution paths are allowed. At runtime, process instances can be composed by allowing all the behaviors which do not violate the constraints. Using DECLARE, it is also possible to combine declarative and imperative specifications, since DECLARE works together with a workflow management system supporting YAWL [108].

### Dynamic Adaptation of Process Instances

When executing a process instance, it may be necessary to deviate from the execution paths defined in the process model in response to changes in the execution environment. Examples of such deviations are adding, skipping, re-executing, or compensating activities. Such adaptations only alter the precise process instance, without affecting the process model or other process instances. In the following, we discuss representative approaches supporting the dynamic adaptation of process models and Web service compositions. We start with approaches which consider adaptable process models. However, we also consider an approach where this dynamicity is built in the representation of processes. All these approaches are reactive, in the sense that the instance is locally adapted to react to changes in the execution environment. We therefore present also an approach which goes in the direction of proactive adaptation.

Many process management systems have been proposed which support the user in defining process instance changes, e.g., ADEPT2 [82, 83], WASA<sub>2</sub> [120], *eFlow* [16], CAKE II [66]. In [116], Weber et al. provide a catalogue of 18 change patterns and 7 change support features, and use this catalogue to evaluate the ability to deal with change of several systems from both academia and industry. Regarding the runtime structural adaptations of process instances, in [116] the authors observe that these can be based on change primitives, such as add/remove node, add/remove edge [120, 66]. When applying change primitives, the soundness of the resulting process model cannot be guaranteed, and must be checked explicitly. Alternatively, the structural adaptations can be based on high-level change operations, e.g., insert task/fragment between two sets of nodes [82]. The high-level change operations can then be associated with pre and post-conditions, which guarantee the soundness of the resulting model.

Several approaches have been proposed for automating the structural adaptation of process instances. Such approaches are often based on planning, as is the case of the works presented in [99], [12], and [28]. In [99], Schuschel and Weske present an integrated planning and enactment system, where alternating phases of planning and coordination are repeatedly traversed whenever an unanticipated effect occurs. This approach is extended in [100], where the replanning is triggered automatically.

Bucchiarone et al. [12] propose a framework supporting the runtime adaptation of service-based applications to unexpected or improbable context changes. Adaptation is achieved through a set of mechanisms which are meant to bring the process instance to a state where the execution can be correctly resumed. The adaptation mechanisms can be combined through adaptation strategies, thus allowing to address complex adaptation needs, which cannot be solved by applying isolated mechanisms. The adaptation mechanisms are realized by extending planning techniques originally designed for the automated composition of services [61]. The framework has been implemented and demonstrated on a complex car logistic scenario in [80].

Friedrich et al. [28] propose a self-healing approach to handle exceptions in service-based processes, such as faulty service invocations. At design-time, the approach can be used to evaluate the reparability of process definitions. At runtime, when an exception arises, the approach generates alternative repair plans by taking into account constraints posed by the process structure, the execution state of the process instance, and the available repair actions.

Dynamic adaptation can be achieved also by using a different process modeling approach. Case handling [111] is a paradigm proposed for supporting knowledge-intensive business processes. With case handling, the users work with whole cases, and the state of the case is not determined

by the control-flow status, but by the presence of data objects. An example of a case handling tool is FLOWer, which supports several deviation operations: open/skip work items not yet enabled, skip or execute enabled work items, and redo executed or skipped items.

A shortcoming of current approaches supporting runtime automated adaptation is that the adaptation occurs only in a *reactive* fashion [71]. In other words, the adaptation is performed only after a deviation from the requirements or a change in context is observed. This has several drawbacks. First, if a failure occurs, the user may have to wait until the application becomes available again. Secondly, a failure can lead to undesirable consequences such as loss of money. The alternative, also called short-term proactive adaptation, is to automatically detect that a constraint will be violated before the conflict actually occurs.

An approach going in the direction of short-term proactive adaptation is AgentWork [69]. The adaptation offered by this workflow management system consists in adding, dropping or replacing individual tasks based on Event Condition Action (ECA) rules. These rules specify the events which constitute logical failures and the corresponding workflow adaptations. The adaptation mechanism uses temporal estimates in order to predict the workflow part which would be affected by the failure and adapt it in advance. The limitations of this approach are that adaptation changes can only affect individual tasks, and that manual intervention is required in case conflicting rules generate incompatible actions.

### **Evolution of Process Models**

Evolution is the ability to modify a process model at runtime. In this subsection, we present approaches that focus mainly on handling the already running process instances when their corresponding process model is modified. Further approaches which deal with process evolution in the

sense of deriving process model changes based on previous process instance adaptations will be considered in detail in Section 2.3.

Handling the already running process instances after a process model change is a challenging issue. Letting the instances finish on the old process model may be impossible, e.g., if the change is the enforcement of a new law. Aborting their execution entirely may also be undesirable if it means losing a great amount of work. The alternative is to migrate the currently executing process instances to the new process model. This is not a straightforward task, as shown with the *dynamic-change bug* originally described in [26]. The dynamic-change bug shows that it may be impossible to put the old instance in a state of the new process model without skipping or re-executing tasks.

In response to these challenges, many approaches have been devised, e.g., [15, 95]. In [15], Casati et al. introduce a taxonomy of policies which describe how to manage running workflow instances when the corresponding workflow schema is modified. The authors introduce also a formal condition, instance *compliance*, to determine which running instances can be migrated to the new schema version: an instance is compliant with a new schema if it can be produced also on the new schema. Sadiq et al. [95] address also the problem of migrating the non-compliant instances to the new schema. If an instance is non-compliant, a compliance graph is constructed which serves to partially roll-back the instance into a compliant state, from which it can execute according to the changed schema.

The problem of migrating process instances can be further complicated by the fact that the instances themselves may have been adapted. This issue is addressed in [86], where Rinderle et al. describe different migration policies based on the degree of overlap between concurrent process model and process instance changes.

Another important issue related to process evolution and instance mi-



gration is that of version control. Without version control, a process model must either be copied manually before it is changed, or it must be overwritten. In the first case, this leads to increased maintenance efforts, as the number of different versions increases. In the second case, problems will arise if there are currently running process instances which cannot be migrated.

With version control, process instances belonging to different versions can coexist, and, if possible, can be migrated to new versions. Examples of approaches which address the problem of versioning for process models are [44] and [125]. In [44], Kradolfer and Geppert present a framework for workflow schema evolution based on workflow type versioning and workflow migration. To keep track of the evolution of a workflow schema, the versions are stored in a version tree, which encodes derived-from relations. The instances can then be migrated to versions in the same version tree. They can also be migrated to previous versions, by performing a series of inverse modification operations. Zhao and Liu [125] present a different method, in which all the versions of a process model are stored in a single directed graph by annotating the nodes and edges with version numbers. Once the graph is built, any version of the process model can be obtained from the graph by following a set of derivation rules.

## **2.2 Ensuring Process Compliance to Goals and Constraints**

Companies today need to ensure that the business practices reflected in their process models are in line with internally defined directives, such as business strategies, as well as externally imposed directives, such as legal regulations and contracts. Complying with external regulations is particularly important in industry sectors with high regulatory control, such as

financial services, gaming, or healthcare. Therefore, there is a need to design compliance rules and develop compliance checking and enforcing mechanisms. This is a challenging task, especially in the case of external directives, which cannot simply be incorporated in the design of the process models [93]. The main reason is the difference in terms of ownership, governance, and timeline, between the business strategy and the regulations.

In this section, we give an overview of approaches which focus on ensuring the compliance of process models to goals and constraints. We consider approaches belonging to the research area of business process management, but also approaches from the related area of Web service composition. These approaches differ in the time when they are applied (design-time vs. runtime), but also in purpose (e.g., verification vs. self-healing) and in the techniques used (e.g., model checking, automated reasoning).

### **2.2.1 Design-time Compliance**

Essentially, there are two ways to ensure at design-time that a process model complies with the specifications of goals and constraints. First, one can automatically generate the process model according to the specifications, starting from process fragments or from Web services. Alternatively, one can verify that the manually created or mined process model complies with the specifications. We give an overview of automatic composition approaches in Section 2.2.1, and of verification approaches in Section 2.2.1.

#### **Automated Composition**

When automatically composing process models, a common approach is to take advantage of the business (data) objects manipulated by the process model. Such business objects are commonly standardized in reference models, in order to facilitate the interoperation between process partners.

They can therefore be used, together with business rules or goals defined based on the business objects, to automatically compose compliant process models. This is the case of the approach by Kuster et al. [48], which can be used to generate business process models based on object lifecycle conformance and coverage requirements. The object lifecycles are synchronized manually, while the rest of the technique is automatic.

Automatic composition approaches which follow this idea have also been proposed in the artifact-centric community. The artifact-centric paradigm has the following dimensions [40]: (i) business artifacts (data), (ii) artifact lifecycles, which describe stages in the evolution of artifacts, (iii) services or tasks in business processes which make transactional changes to artifacts, and (iv) the associations or constraints on the manner in which the business processes can make changes to the artifacts. The work presented in this thesis can also be seen as artifact-centric, with domain objects, effects, and preconditions corresponding to (ii)-(iv). In [29], Fritz et al. devise a technique for automatically composing declarative artifact-centric workflows starting from a goal to be achieved and service descriptions. The authors consider only the restricted setting when artifacts are key-value pairs, without explicitly considering the artifact lifecycle.

With Web service composition, the problem is to generate a composite service starting from service interfaces and composition requirements. The result is an executable implementation which satisfies the requirements by suitably invoking the existing Web services. Many approaches to automated Web service composition are based on AI planning techniques (e.g., [38, 62, 76, 104]). In these approaches, automated composition is described as a planning problem: existing services are used to construct the planning domain, the planning goal is obtained from the composition requirements, and planning algorithms are used to generate a plan, which is then translated back to a composite service. In [102], we proposed an approach for the

automated composition of Web service functionalities described in terms of background ontologies. We used the most general notion of matching, partial matches, where several Web services can cooperate, each covering only a part of a requirement. Due to the background ontologies and to partial matches, finding a composition which satisfies the goal involves searching in a huge search space. To overcome the size of the search space, we developed heuristic techniques for guiding the search.

In the following, we focus on one particular Web service composition approach, ASTRO (see [76, 10, 11, 43]), which is also the starting point for the work presented in this thesis. ASTRO is a composition framework based on planning as model checking, which covers both cases of extended goals and cases of partial observability. Given an abstract BPEL description of the component services and a formal representation of composition requirements, the approach automatically synthesizes an executable BPEL process. The synthesized process orchestrates the components in such a way as to satisfy the given requirements.

The composition requirements used in ASTRO include both control flow and data flow requirements. Traditional composition requirements refer to reaching a state or outcome. However, in many scenarios it is necessary to align the behavior of the composed services. Also, it may be necessary to ensure that the composed service reacts to certain events in a predefined way. The approach in [10, 43] shows how to express such control-flow requirements in terms of object diagrams and their lifecycles. The requirement language in [10] allows to define not only reachability goals, but also recovery conditions and preferences. This approach is further enhanced in [43] to deal with problems specific to user-centric service compositions.

ASTRO has been implemented and applied to a real world case study that involves the Amazon e-commerce services and an e-payment bank service [59]. These characteristics made ASTRO an ideal starting point for

our work on corrective evolution. In particular, we have exploited the goal language and object diagrams introduced in [10] to model our application, and used the powerful planning techniques in ASTRO to implement our solution.

### **Verifying Compliance**

Another way to ensure at design-time that the process model achieves the goal and/or complies to the constraints is to apply a verification technique. Many approaches in this category annotate the tasks in the process models with semantic information, and use these annotations to evaluate formally specified constraints. Ghose and Koliadis [31] annotate the tasks in BPMN process models with effects, and devise a technique to accumulate the effects of ordered tasks. The focus in [31] is however not on verifying that the process is compliant to the constraints, but rather on how to resolve non-compliance. Non-compliance is resolved by identifying process models which are minimally different from the original, and which satisfy the constraints. The authors present also a set of process compliance patterns, which capture commonly occurring constraint violations and the actions required to resolve the non-compliance.

Hoffmann et al. [39] annotate the tasks in a process model with preconditions and effects. These are conjunctions of logical literals, formulated in terms of an ontology which axiomatizes the business domain. Using these annotations, the authors develop low-order polynomial time methods for verifying the compliance of process models with a constraint base. The methods perform exact compliance checking for restricted cases, and approximate compliance checking for more general cases, guaranteeing either only soundness or only correctness of the results.

If no semantic annotations are added, it is only possible to verify the compliance of process models with constraints referring to structure or rela-

tionships between activities. This is the case of the approach by Awad et al. [5], which checks the compliance of process models with activity ordering compliance rules. The compliance rules are expressed as queries formulated in BPMN-Q, a visual language based on BPMN. A query processor matches BPMN process graphs to the query graph. The resulting sub-graphs are reduced and eventually translated into finite state machines. The query is transformed into formulas in temporal logic, and a model checker is used to verify if the finite state machines satisfy the temporal logic formulas. A similar approach is proposed by Liu et al. in [54], which uses model checking to verify the compliance of BPEL processes with compliance rules specified in the graphical Business Property Specification Language.

The work presented by Governatori et al. in [34] addresses also the normative aspect of compliance, that is, whether the constraint refers to an obligation, a permission, or a prohibition. This is achieved by specifying business contracts in a Formal Contract Language (FCL). The compliance of a BPMN process with a contract is verified by comparing events along the process execution paths with the contract conditions.

### **2.2.2 Runtime Compliance**

When ensuring the compliance of process models at runtime, there are essentially two possibilities. The first possibility is to monitor the execution and record any constraint violation. The second possibility is to also react to constraint violations by triggering healing procedures.

Monitoring the runtime compliance has been addressed in, e.g., [4, 57, 58]. Agrawal et al. [4] present a workflow solution for addressing the internal control requirements of the Sarbanes-Oxley Act. The architecture of this solution consists of four components: workflow modeling, active enforcement, workflow auditing, and anomaly detection.

The approach presented in [57, 58] is closer to the work presented in this

thesis, as it concerns the monitoring of compliance for adaptable process models. Ly et al. formally define two types of semantic constraints which impose conditions on how certain activities can be used in the process: mutual exclusion and dependency. Based on exploring execution traces, the approach verifies that the process is semantically correct, i.e., it complies with the annotations. The authors then extend the approach to deal with adaptations of process instances and evolution of process models, as well as propagation of changes from process models to (possibly adapted) process instances. Here, optimization techniques are used, which restrict the set of relevant constraints based on the semantics of the change operations applied. The limitation of this work is that compliance is limited to constraints on relationships between activities in a process.

An approach which addresses not only the monitoring of process instances, but also their self-healing, is presented in [7, 6]. Baresi et al. propose a framework (Dynamo) for self-healing BPEL compositions. More precisely, supervision rules are added to the BPEL process, specifying a location where the property should be verified, the monitoring parameters (priority, validity, trusted providers), a monitoring expression specified in WSCoL (Web Service Constraint Language) and a recovery strategy specified in WSReL (Web Service Recovery Language). There can be different recovery strategies for the same violated constraint, and the selection is done based on context.

Monitoring the runtime compliance of processes and triggering self-healing actions in case of violations is a technique often employed when designing an automatic process instance adaptation approach. In fact, some of the approaches presented in Section 2.1.2 (e.g., [12, 28, 69, 100]) can be viewed also as self-healing approaches.

## 2.3 Related Work

Evolving a process model may be necessary in various situations, e.g., to accommodate legal or policy changes, to implement optimizations, or to improve the design of the process model. A variety of approaches exist which use the structural adaptations of process instances to evolve process models or to support evolution. We distinguish between approaches which consider the logs of the instance adaptations, also referred to as change logs (e.g., [87, 105]), and approaches which consider the process variants that result by structural adaptation (e.g., [37, 53]). We also provide an overview of process model refactoring techniques, which focus on improving the quality of the process models.

### 2.3.1 Analyzing Execution and Adaptation Logs

Process management systems typically offer the possibility to record information about the execution of process instances. This information is represented as events, where each event refers to a precise task in the process model and to a process instance, and events are totally ordered. Some process management systems also allow process instances to deviate from the prescribed process model or be adapted structurally while they are running (e.g., [83, 12]). In such systems, the instance adaptations can be recorded into adaptation logs, also referred to as change logs. The execution and adaptation logs can then be analyzed to support process diagnosis (e.g., [35, 89]), facilitate change reuse (e.g., [105, 98]), and evolve the model (e.g., [87, 117]). Also the work presented in this thesis belongs to this category, since our corrections are generated based on adaptation logs.

Guenther et al. [35] support *process diagnosis* by applying process mining techniques to change logs. The result is an abstract change process,



consisting of change operations and causal relations between them. These change processes can be used as an analysis tool to understand when and why changes were necessary. Rozinat and van der Aalst [89] analyze how data attributes influence the choices made in the process based on past process executions. They convert each decision point into a classification problem, where classes are the different decisions to be made, and then solve the problem using decision trees.

The approach by Soffer et al. [105] is designed to facilitate *change reuse*, but also to support process diagnosis. This is a learning approach for grouping the process instances based on similar contextual properties, paths, and outcomes. The groups can be used to provide criteria for path selection in a given situation, to address specific questions (e.g., "hunches" about the cause of poor performance), and to identify successful deviations from the existing process model. In the context of declarative processes, Schoenenberg et al. [98] present an approach which facilitates change reuse by generating recommendations based on similar past executions and considering optimization goals.

The approach in [87] addresses both the problem of facilitating change reuse and the actual *process model evolution*. Rinderle et al. use concepts and methods from case-based reasoning in order to log, together with the change operations, also the reasons for and context of each change. Change information is stored as cases in a case-base specific to the process model. The case-bases are used to support process actors in reusing information about similar ad-hoc changes, and are also continuously monitored to automatically derive suggestions for process model changes. The work in [87] is extended in ProCycle [117]. Retrieving similar changes was an entirely manual process in [87], due to the fact that the reasons for change were specified as a natural language annotations. In ProCycle, the retrieval process is partially automated by considering as well structured information

about the context.

A different approach also employing case-based reasoning to reuse previous changes is presented by Minor et al. in [65]. The authors propose a suspension mechanism, which allows the designer to modify suspended parts of the workflow while the remainder of the workflow can continue to be executed.

A recent approach from the process mining community which is very close to our work is presented in [27]. Fahland and van der Aalst investigate the problem of repairing a process model with respect to a log, such that the repaired model can replay the log and is as similar as possible to the original model. The non-fitting subtraces are grouped into sublogs, which are then mined for perfectly fitting subprocesses. These subprocesses can then be added to the original model at the appropriate location.

### 2.3.2 Managing Process Variants

Instead of logs of adapted instances, the result of structural adaptation can be represented as variants of a process model coexisting in the same process collection. As observed also in [23, 37], the techniques for managing process variants fall into two categories: techniques which keep the variants separate and provide a way to keep track of commonalities (e.g., [24, 25, 46, 55]), and techniques which use a reference model to represent the variants (e.g., [37, 85, 88, 53]). To obtain a reference model, the process variants may have to be merged or aggregated (e.g., [53, 49, 47]). In the following, we present several available techniques for managing and merging process variants, focusing in particular on approaches that can be used for process evolution.

If the process variants are kept as separate models, there is a need for an infrastructure that can keep track of the commonalities between variants. There are many reasons for having such an infrastructure in

place. Most notably, since process models are created by copying or merging fragments from other models, the process model repositories tend to accumulate clones over time [24]. Besides eliminating clones, other reasons are to ensure consistency between variants when updating them, or to identify reusable fragments. Many approaches for querying a process model repository have been proposed, which return either exact matches [24], or inexact matches together with a similarity measure [46, 56]. Other approaches focus on version control, e.g., [25].

Managing process variants using a reference process model can be achieved through process configuration, which we have discussed in Section 2.1.1. Among the approaches supporting process configuration, Provop [37] is particularly relevant to our work, since it allows for structural configuration [84]. In other words, Provop allows process variants to be configured from a base process model by applying structural adaptations (e.g., inserting, deleting, and moving fragments). The structural adaptations can be selected automatically based on context [36]. This idea that process models can be structurally adapted to their execution context is also one of the main assumptions in this thesis.

Another relevant approach which supports the management of process variants using a reference model is presented by Li et al. in [53]. The idea here is to learn from past changes by merging the process variants into a process model which covers the variants best. Considering that the distance between two process models is the minimal number of high-level change operations necessary for transforming one process model into the other, [53] employs a heuristic search to find a new process model such that the weighted average distance between the new model and variants is minimal.

The approaches in [49] and [47] are also techniques for merging process variants. La Rosa et al. [49] (semi-)automatically produce a configurable

process model from a pair of variant models, such that the behavior of the resulting process model subsumes that of the input variants. This is different from the approach in [53], where the new process model covers the variants only partially. The algorithm presented in [49] works by creating a copy for the common parts of the variants, and appending the differences using configurable connectors.

The technique described by Kuster et al. in [47] can be used for computing and resolving differences between variants in the absence of a change log. Differently from [49], the aim here is not to merge the variants automatically, but to assist the modelers in manually resolving the differences.

### 2.3.3 Refactoring Process Models

*Refactoring* is a term originating from software engineering, where it usually refers to techniques used for restructuring a body of code, without changing its external behavior. Refactoring is performed to improve the quality of the software in terms of readability, maintainability, reduced complexity. Similar to code refactoring, the refactoring of process models refers to improving the internal quality of process models without affecting their external behavior.

Several refactoring opportunities for the control flow of process models, also known as process model smells, have been presented in [115]. To address these smells, Weber et al. propose 11 behavior-preserving process refactoring techniques. Close to the work presented in this dissertation is the refactoring technique RF11, *Pull Up Instance Change*. In particular, our work can be used to automate this refactoring technique.

Another problem belonging to process model refactoring is that of transforming an unstructured process model into a structured one with the same behavior. A process model is structured if every split node has a corresponding join node, and split-join blocks are properly nested. Structured

process models have been shown empirically to have less errors and to be easier to understand [51]. There are many approaches which propose automated techniques for structuring process models. For example, the technique introduced by Vanhatalo et al. [114] uses the refined process structure tree (RPST) decomposition previously developed in [113]. Also using the RPST decomposition, Polyvyanyy et al. [78] provide a full characterization of the acyclic process models which are inherently unstructured, i.e., which cannot be transformed into structured models. Deriving structured models from unstructured ones is also discussed in the context of transforming graph-based process models [73], respectively BPMN models [74] to BPEL.

Golani and Gal [32] propose another approach for restructuring process models. The authors start from on the observation that exception handling design is typically performed only after the normal process of execution has been designed. They therefore propose to restructure the process model by re-ordering the activities based on exception efficiency considerations.

Refactoring process models can also refer to correcting behavioral errors in these models. Correcting behavioral errors is not trivial, for example because fixing one error may introduce new errors in other parts of the model. Gambini et al. propose a technique called Petri Nets Simulated Technique for automatically fixing unsound process models [30]. The core of this technique is a heuristic optimization algorithm. At each step, the algorithm tries to minimize the number of behavioral errors by applying controlled changes. The algorithm stops when a certain number of non-redundant solutions is found, or a timeframe elapses.

## 2.4 Discussion

The approaches in [87], [117], and [53] are closest to our work, since they generate new process models based on adaptation logs, respectively process variants. All three approaches derive process model changes from frequent instance changes. The context of the adaptations is considered in [87] and [117], but not considered in [53]. In [87], the context is represented as natural language annotations which are entered and processed manually. Structured information about the context is considered only in [117]. Further, the process trace for which changes should be applied is considered implicitly in [53], and not considered in [87] and [117].

However, a process instance adaptation is tightly coupled to the context *and* trace for which it is used, and may even be harmful if used for different contexts or traces. When deciding to evolve the process model based on instance adaptations, the contexts and traces must be considered as well. If traces are ignored, then we need to consider the overall goal of the process model, to make sure that the adaptations that we introduce in the process model are not harmful. The process goal is not considered in either of the three mentioned approaches.

The relation between context, traces, and goals has been considered in [105]. However, the aim in [105] is to provide recommendations for improving the process model, rather than actually changing the process model, and could therefore be used as an analysis technique that precedes corrective evolution.

# Chapter 3

## Evolution Framework

In this Chapter, we introduce a framework designed to support the evolution of business processes based on execution and adaptation histories. We first present an overview of the evolution framework from an architectural perspective. We describe in detail each phase in the lifecycle of a process-based application which uses the framework, with the aim to position corrective evolution in this lifecycle. We then introduce the basic elements for modeling the process-based application. Finally, we present in detail concepts related to process execution and adaptation. These concepts will be used in Chapter 4 to define the corrective evolution problem.

### 3.1 Process-Based Application Lifecycle

In this Section, we describe a framework designed to support the adaptation and evolution of process models, and explain the role of corrective evolution in this framework (see also [8, 13]). Figure 3.3 presents an architectural overview of the framework. The focus of this figure is to show the relations between the different components, and to position these components with respect to the three main phases in the evolution lifecycle, namely (1) execution phase, (2) analysis phase, and (3) evolution phase. The relations between components do not imply that there is a strict tem-

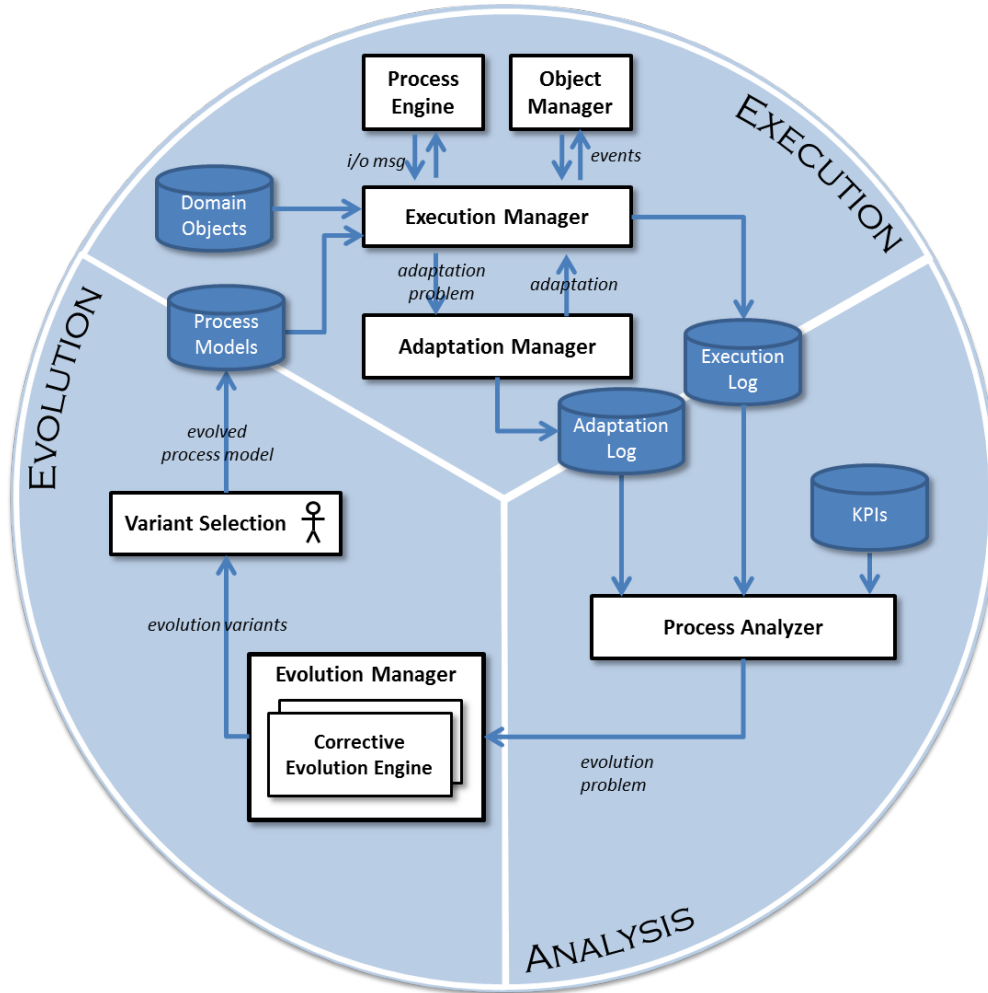


Figure 3.1: Evolution framework

poral ordering between the three phases. For example, both analysis and evolution tasks can be performed while the application is executing.

Our evolution lifecycle is not strictly different from the traditional business process lifecycle, which typically includes the following phases organized in a cyclical structure: the design and analysis phase, the configuration phase, the enactment phase, and the evaluation phase [121]. However, the focus in the traditional lifecycle is on modeling, validating, and deploying an initial process-based application, which can then be executed,



monitored, and analyzed. While a feedback loop from the evaluation phase to the design phase exists, the changes are supposed to be minimal. This is why process evolution is typically included in the design and analysis phase, and is assumed to be a manual task.

However, the ability to cope with frequent process changes has been recognized as one of the critical success factors for achieving effective process support [70]. Therefore, in our evolution lifecycle, we focus less on the initial setup of the application, corresponding in the traditional lifecycle to the design and analysis phase, as well as the system configuration phase. Instead, we consider that the process-based application is already set up and running, and focus on the application's ability to change in response to short and long-term changes in the execution environment. In our evolution lifecycle, we therefore consider explicitly both short-term process instance changes (included in the execution phase as adaptation), and long-term process model changes (the evolution phase).

In the following, we present in detail each phase in the evolution lifecycle.

### 3.1.1 Execution Phase

We model a process-based application using *domain objects* and *process models*. The domain objects are used for representing the domain knowledge, while the process models represent the business logic. In the execution phase, the process models and domain objects are instantiated. The resulting process instances are executed on the *process engine*. The *execution manager* is responsible for keeping the system configuration up to date and for consistently aligning the status of domain objects with the execution of process instances and the context events received by the *object manager*.

The system configuration is used by the execution manager to monitor constraints associated to running process instances and to trigger excep-

tions in case of constraint violation. Such exceptions can be considered at design-time, by including in the process model appropriate exception handlers, for example as in [60, 91]. If, however, the exceptions are not handled at design-time, they will require runtime structural process adaptations.

In our framework, structural process adaptations are handled by an *adaptation manager*, which can be realized using existing state of the art techniques such as [12, 21, 28]. The adaptation manager receives as input an adaptation problem, which contains information about the process instance to be adapted, the system configuration, and the constraint violation. The adaptation manager is then responsible for computing an adaptation for the process instance, which addresses the constraint violation.

There can be several adaptations which are applicable to the same exceptional situation. For example, for one constraint violation, possible adaptations can be to bring the process instance to a stable state which does not violate the constraint, to skip or replace some activities in the original process model which are no longer applicable in the current context, or to compensate and terminate the process instance. Preferences between adaptation strategies can be specified in advance, for example, we can specify that compensating and terminating the process instance should be performed only in case there is no other means to address the constraint violation.

All the information regarding execution and adaptation which may be useful in the other phases is recorded into execution and adaptation logs. The *execution log* contains events which record information such as the time when an activity was started or completed, the process instance the activity belongs to, the user who performed the activity, and the resources used for performing it. Also the *adaptation log* contains events, this time recording the adaptation needs, such as constraint violations, as well as

the actual adaptations performed.

### 3.1.2 Analysis Phase

In the analysis phase, the execution and adaptation logs are continuously monitored by a *process analyzer*, in order to determine performance problems. The process analyzer can be realized using existing approaches such as [17, 122]. An example of a performance problem is failing to meet the target of a Key Performance Indicator (KPI). State of the art approaches supporting the analysis of process instance executions use data mining techniques on the values measured for different metrics (e.g., execution time), in order to uncover patterns in the metric behavior. Such patterns show the main factors which influence the behavior of the metric. This analysis then enables system users to *react* to existing performance problems, and can also be used to *proactively* avoid such problems by predicting metric values for the currently running process instances [17, 122].

If a performance problem is identified in this phase, and the cause of the problem can be resolved by modifying the process model, this will result in the creation of evolution problems. For example, the cause of the performance problem can be that adapting at runtime is too costly in a certain situation, because the situation occurs more often than originally estimated. This can be resolved by modifying the process model to include an adaptation which has performed successfully in this situation. The actual modification of the process model can then be formulated as an evolution problem, which will include the process model, the successful adaptation, and the situation for which the adaptation should be applied.

A different example is if the cause of the performance problem is an adaptation which is unsuccessful in a particular situation. There can be different ways to address this problem. For example, the process model can be modified to include a successful adaptation for that situation, or

simply to prevent that the problematic adaptation is used. However, the use of the problematic adaptation can also be restricted without modifying the process model, for example by notifying the adaptation manager.

### 3.1.3 Evolution Phase

In the evolution phase, the evolution problems are processed by an *evolution manager*. An evolution problem will contain the process model which needs to be changed, as well as a specification of these changes. The evolution manager is then responsible for identifying, based on the evolution problem, the type of evolution required, and for delegating the task to the appropriate *evolution engines*. The evolution engine addresses the evolution problem by creating evolution variants. These are new process models which result from changing the original process model according to the change specifications. The process designer can then select from these evolution variants the new process model on which future process instances will be based.

The main focus of this dissertation is to address one particular type of evolution, *corrective evolution*. With corrective evolution, new process models are created by enhancing the original process model to handle several situations with successful adaptations, at the same time ensuring that these new process models satisfy the goal of the original process model. The evolution problems which can be solved by a *corrective evolution engine* must therefore contain a process model, a specification of the goal of this process model, and a list of situations with the corresponding successful adaptations. With corrective evolution, we can construct different evolution variants by allowing more or less freedom when deciding on which traces to plug in particular adaptations. These process models will all be extensions of the original model with the same adaptations, but will differ in terms of the behavior that they allow.

For example, in the scenario introduced in Section 1, the corrective evolution problem is to enhance the original car process model to handle two situations with successful adaptations, while at the same time ensuring that the resulting process model continues to satisfy the goal of delivering the car to the retailer in perfect condition. The first situation is when the car is damaged while moving to the storage area and other cars are waiting to be repaired. The adaptation to be plugged in for this situation is to postpone the repair. The second situation is when the car is damaged at the storage area and there are not too many cars waiting to be repaired. The successful adaptation for this second situation is to repair the car.

By applying corrective evolution, several evolution variants can be constructed, due to the fact that the two situations can be reached repeatedly. For example, with the second adaptation, the car is repaired and returns to the storage area. While moving to the storage area, the first situation can reoccur; when at the storage area, the second situation can reoccur. Since each adaptation can be applied for one or for multiple occurrences of the corresponding situation, we will obtain a different evolution variant for each combination.

Corrective evolution is only one type of process evolution. In [13] we proposed a different type of process model evolution, which we called *preventive evolution*. With preventive evolution, the idea is to modify the process model in order to prevent a situation which requires adaptation from being reached. This type of evolution is more disruptive than corrective evolution, since it requires a restructuring of the original process model such that the critical situation is avoided for all possible executions of this process model.

Another type of evolution is the inverse of corrective evolution, i.e., the task of removing from the process model the procedures for handling particular situations, at the same time ensuring that the resulting process

models satisfy the goal of the original process model. With corrective evolution, a process model will be enhanced to handle an exceptional situation with a successful adaptation. However, by changing the process model, all future process instances will handle the exceptional situation in the same way, which may cause the previously successful adaptation to perform unsatisfactorily, for example, due to a lack of resources. In this case, it would be useful to be able to remove the under-performing adaptation from the process model. This is not a trivial problem, especially if in the meanwhile several other structural changes have been performed.

After generating the evolution variants, one or more of these variants can be selected as the new process models on which future process instances will be based. For most application domains, completely automating this step of evolution variant selection is unrealistic. Modifying the process models is often a critical operation, and understanding the impact of each evolution variant on the system requires a comprehensive knowledge of the domain.

However, the task of selecting evolution variants can be simplified by first ranking these variants. This ranking can be performed according to process quality metrics, such as the metrics proposed by Mendling in [63]. The evolution variants can also be ranked based on how well they cover the behavior observed in the execution and adaptation logs, using metrics from process mining, such as the fitness and appropriateness metrics described by Rozinat and van der Aalst in [90]. Finally, they can be ranked according to the performance data recorded in the execution and adaptation logs, for example by aggregating for each evolution variant the performance values of all the fitting process instances. Ranking the evolution variants based on combinations of these metrics would allow to balance between competing concerns, for example between the quality of the process model and avoiding untested or unsuccessful behavior.

## 3.2 Process-Based Application Representation

In this section we present the modeling elements of a process-based application. We start with an overview, and then discuss in turn each modeling element.

For modeling a process-based application, we consider that there exist three layers, arising from two distinctions (Figure 3.2). First, we distinguish between domain knowledge, or knowledge about properties of entities in the domain, and process knowledge, or knowledge about business logic. Second, we distinguish between concrete, context-dependent knowledge, and knowledge that is common, abstract, independent of context. The first layer is therefore the domain knowledge, which is stable and abstract. The second layer is the stable part of the process knowledge, represented using goals, while the third is the dynamic and concrete part, represented using adaptable process models.

To encode the domain knowledge, we define *domain objects*. Each domain object corresponds to a property of an entity in the domain, and consists of a set of possible values and a set of events representing value changes. We use this domain knowledge to define our process *goals*. Using goals, we can specify the target state for our process, as well as coordination requirements. A target state is a situation we want to achieve and then maintain in every execution of the process, whereas a coordination requirement is a property we want to ensure during every execution.

Goals are then used to constrain *adaptable process models* and their execution. A process instance may be adapted at runtime, as long as it satisfies the goal associated to the process model from which the instance was created. The goal is also considered during evolution, since process models may evolve only as long as they continue to satisfy the goal of the original process model. We represent process models as directed graphs in

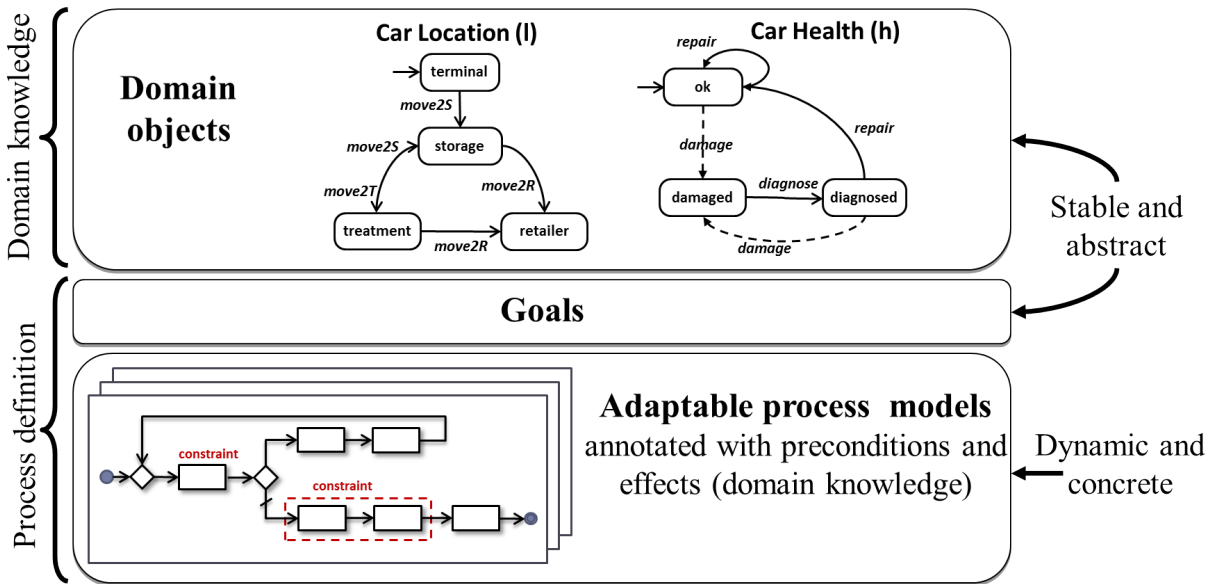


Figure 3.2: Process-based application representation

which the nodes are either steps in the process (i.e., activities) or control connectors. To capture the relation between process models and domain, we use the domain objects to specify constraints on the steps of the process model, as well as to specify how these steps contribute to the outcome of the process.

Figure 3.3 shows the usage of these modeling elements during the evolution lifecycle discussed in the previous section.

We now present in detail each modeling element. For each element, we provide an informal description, formal definitions, and examples on the car logistics scenario.

### 3.2.1 Domain Objects

A domain object is a state transition system representing a property of a physical or computational entity. States correspond to property values, and value changes are transitions between states triggered by events. We distinguish between controllable events, which can be triggered by execut-



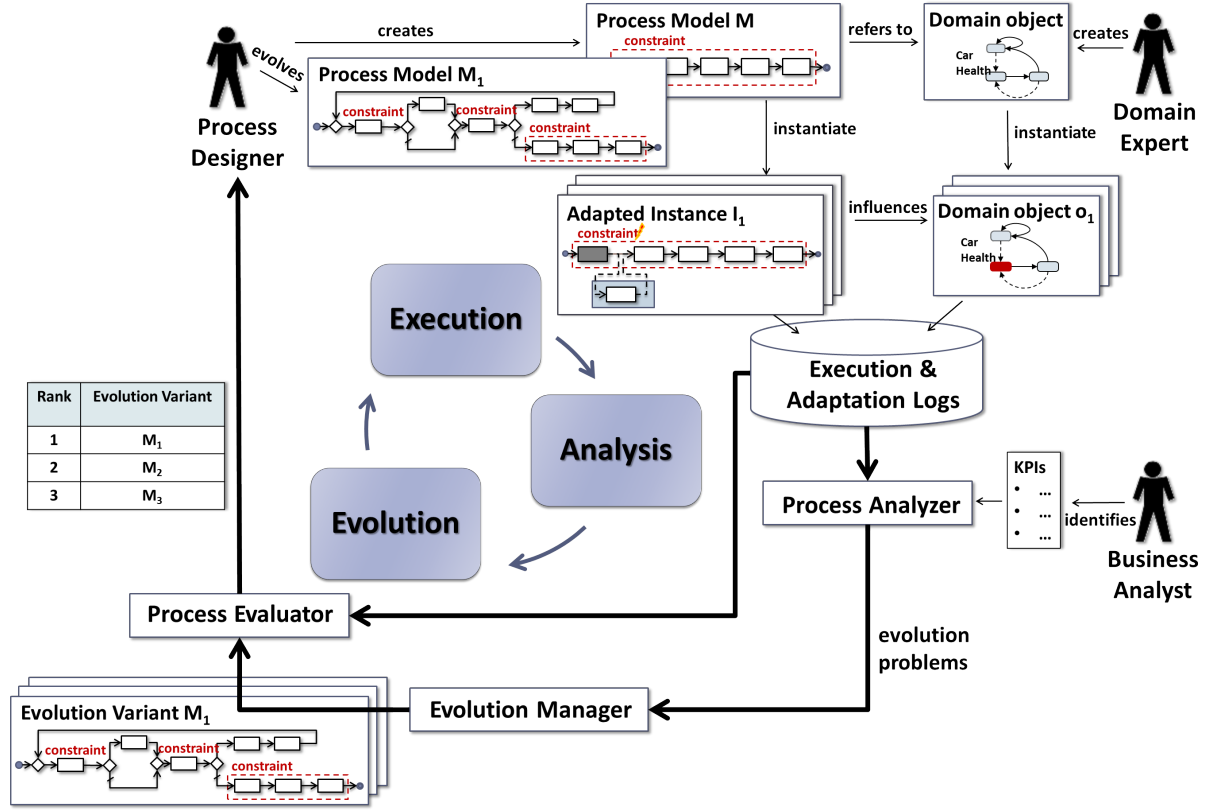


Figure 3.3: Modeling elements in the process lifecycle

ing a process, and uncontrollable events, which can happen at any time and cannot be triggered directly. Domain objects can be nondeterministic, i.e., for a state  $s$  and an event  $e$  there can be multiple choices for possible next states due to multiple transitions from state  $s$  on event  $e$ .

### Definition 1 (Domain Object)

A domain object is a tuple  $\langle L, L^0, \mathcal{E}, T \rangle$ , where

- $L$  is a finite set of states;
- $L^0 \subseteq L$  is a set of initial states;
- $\mathcal{E}$  is a set of events partitioned into sets:
  - $\mathcal{E}_C$  of controllable events;
  - $\mathcal{E}_U$  of uncontrollable events;

- $T \subseteq L \times \mathcal{E} \times L$  is a transition relation based on events.

Let  $O$  be a set of domain objects. We denote with  $P_S$  the set of state propositions of the form  $s^s(o)$ , where  $o$  is a domain object, i.e.,  $o = \langle L, L^0, \mathcal{E}, T \rangle \in O$ , and  $s$  is a state, i.e.,  $s \in L$ . Similarly, we denote with  $P_E$  the set of event propositions of the form  $e^e(o)$ , where  $e$  is an event, i.e.,  $e \in \mathcal{E}$ . Further, we denote with  $Bool(P_S)$  the set of boolean expressions over  $P_S$ .

**Example** Figure 3.4 shows the domain objects in our scenario. From the perspective of a process model, there are two types of domain objects. First, there are domain objects which can be directly manipulated by the process. In our example these are the Car Location, Health, Navigation, the Delivery Order, and the Treatment and Repair Schedules. Such domain objects refer to contextual properties of particular entities, in this case the car.

The domain objects of the second type can be read by the process, but cannot be directly manipulated. Such domain objects will contain only uncontrollable events. In our scenario, this is the case of the Service Station Queue. Such domain objects are descriptions of resources, which are shared by all process instances. Since resources can be added or removed at runtime, also the domain objects corresponding to resources can be added or removed dynamically.

### 3.2.2 Goals

Goals specify desirable states to be reached, and then continuously maintained, during the execution of a process instance. Goals can also be used to specify reaction rules, which define how the process should react when certain states are reached. We express goals in terms of domain objects.

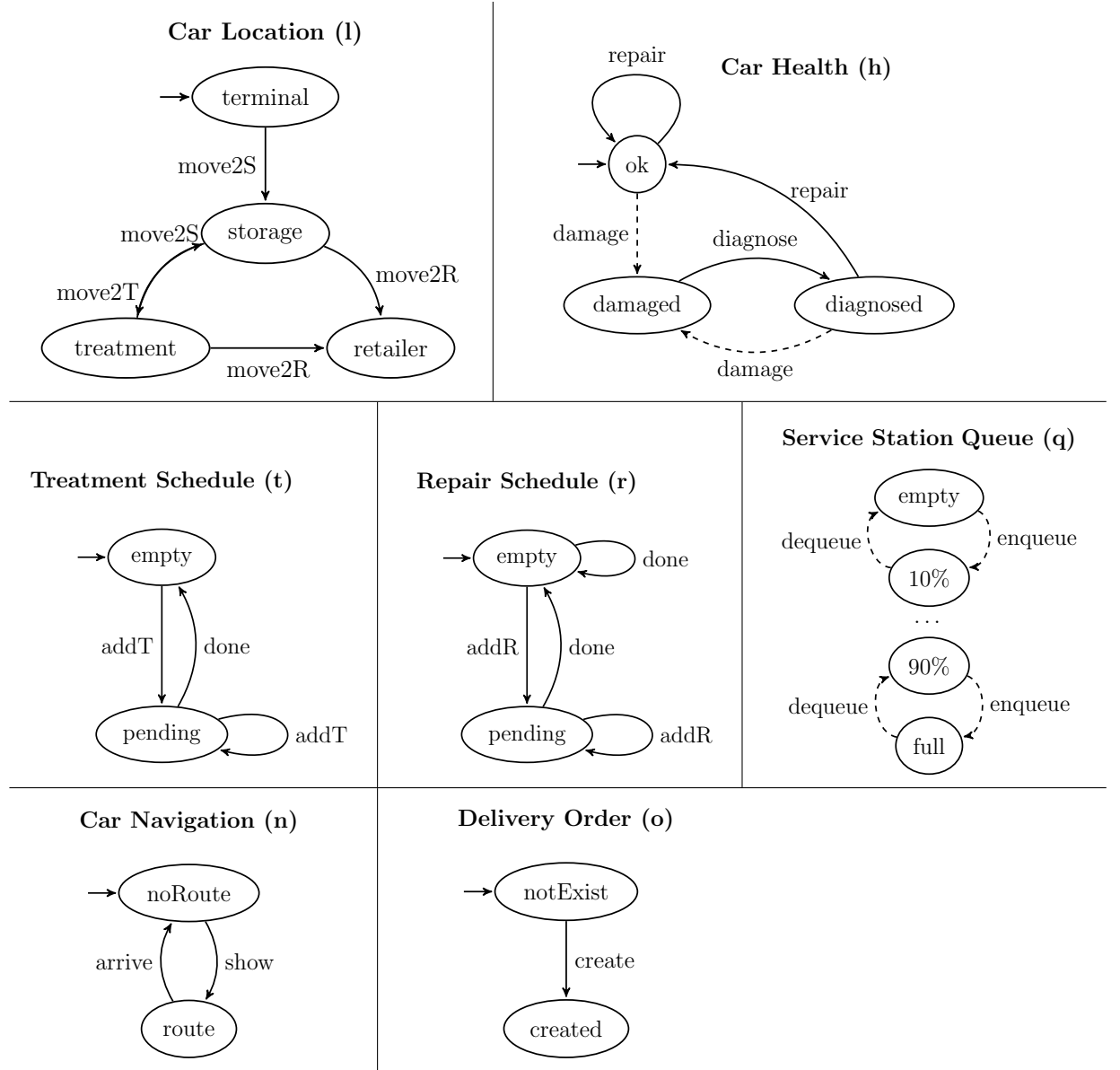


Figure 3.4: Domain objects in the car logistics scenario

### Definition 2 (Goal)

Let  $O$  be a set of domain objects. A goal defined over  $O$  is a set of goal statements, where each goal statement is defined with the generic template

$$\psi_0 \implies (\psi_1 \succ \dots \succ \psi_n)$$

where  $\psi_i \equiv \top \mid s^s(o) \mid \psi_i \vee \psi_i \mid \psi_i \wedge \psi_i$  and  $o \in O$ .

A goal statement specifies that whenever the state in the left side of the rule occurs, the process should reach the state defined by the right side. If the left side of the rule is empty ( $\top$ ), the state in the right side should be reached unconditionally. The states on the right side are ordered using a preference operator ( $\succ$ ), from the most to the least preferred.

**Example** In our scenario, the goal contains one goal statement, to unconditionally reach and maintain the state when the car is delivered to the retailer in perfect condition:

$$\top \Longrightarrow ok^s(h) \wedge retailer^s(l) \quad (G_1)$$

We also give an example of a reaction rule, to show the difference between unconditional and reaction rule goal statements. Consider the following goal with one goal statement:

$$damaged^s(h) \Longrightarrow ok^s(h) \succ diagnosed^s(h) \quad (G_2)$$

This goal specifies that, in case the car is damaged, the process should try to reach and keep the car in state ok. If this is not possible, the process should at least get the car in the state where the damage is diagnosed.

### 3.2.3 Process Models

A *process model* is a directed graph, for which the nodes are either *activity nodes* or *control connectors*. Nodes are connected by *control edges*, which represent precedence relations. Activity nodes are labeled with *activities*. Not all the activity nodes must be labeled; unlabeled activity nodes can exist for a control flow purpose. An activity can correspond to more than one activity node, i.e., duplicate nodes are allowed.

Activities are considered to be atomic. In other words, in our formalism, a long-running task cannot be represented as a single activity. Instead, such

a task will be represented using several activities which denote the different phases in the execution of the task, such as beginning and completion.

We relate activities to the domain through preconditions and effects. The *preconditions* are boolean formulas over the states of domain objects. Their semantics is that an activity can only be executed if its precondition holds. The *effects* correspond to controllable events on the domain objects. By annotating an activity with effects, we encode the fact that some domain objects may move to different states as a result of executing the activity. Because the domain objects can be nondeterministic, also the effects of an activity can be nondeterministic. We assume that the effects of an activity cannot refer to multiple events on the same domain object, i.e., an activity can trigger at most one event on a domain object.

**Definition 3 (Activity)**

An activity is a tuple  $\langle a, pre, eff \rangle$  defined over a set of domain objects  $O$ , such that:

- $a$  is the activity name;
- $pre \in Bool(P_S)$  is the activity precondition;
- $eff \subseteq P_E$  are the activity effects, and for all  $e_1^e(o) \in eff$ , if there exists  $e_2^e(o) \in eff$ , then  $e_1^e(o) = e_2^e(o)$ .

**Example** We consider an activity  $\langle a, pre, eff \rangle$  from our scenario, defined on the domain objects in Figure 3.4, and for which:

- $a = Receive\ delivery\ order$
- $pre = notExist^s(o) \wedge \neg damaged^s(h)$
- $eff = \{create^e(o), addT^e(t)\}$

To execute this activity, the precondition is that a delivery order for the car should not yet exist and the car should not be damaged. The effects

are that the delivery order is created and the required treatments (e.g., painting) are added to the treatment schedule.

For modeling the control flow of a process model, we use control edges and the following control connectors: AndSplit, AndJoin, XorSplit, XorJoin. These constructs can be used to realize the following workflow patterns [109]: sequence, parallel split, synchronization, exclusive choice, simple merge, and arbitrary loop. These patterns form the core of any process modeling language. Because of this, our representation can easily be mapped to a process modeling language such as BPMN [1], a process execution language such as WS-BPEL [2], as well as modeling languages with formal semantics such as state transition systems and Petri Nets. Moreover, these control-flow patterns are the ones most often used in practice [126], and are also directly supported by most of the contemporary workflow management systems [109]. Finally, process models which have only AND and XOR connectors are more understandable and less prone to errors [64].

In addition to activity preconditions and effects, process models can contain two other types of domain annotations: constraints and conditions. A scope with *constraint* P is a sequence of activities with the same precondition P. Control edges that connect XorSplit nodes with other nodes can also have annotations, called *conditions*. Like preconditions, the conditions are boolean formulas over domain objects states.

**Definition 4 (Process Model)**

Let  $O$  be a set of domain objects and  $\mathcal{A}$  a set of activities defined over  $O$ . A process model  $M$  defined over  $O$  and  $\mathcal{A}$  is a tuple  $\langle N, E, l, t, c \rangle$  where:

- $N$  is a finite set of nodes partitioned into sets:
  - $N_A$  of activity nodes;

- $N_C$  of control connectors;
- $E \subseteq N \times N$  is a set of directed edges;
- $l : N_A \rightarrow \mathcal{A}$  is a function mapping activity nodes to activities;
- $t : N \rightarrow \{Start, End, Normal, XorSplit, XorJoin, AndSplit, AndJoin\}$  is a total function assigning a type to each node;
- $c : E \rightarrow Bool(P_S)$  is a branch condition function such that for all  $e = (n_1, n_2) \in E$ , if  $c(e)$  is defined then  $t(n_1) = XorSplit$ .

Without loss of generality, we impose a restriction on the definition of activity preconditions. The restriction concerns the formulas included in preconditions, which can only refer to situations reachable through controllable events. The reason is that an uncontrollable situation represents in some sense an exceptional situation, which may or may not happen, and we need to make sure that the process model handles also the case when the exceptional situation does not happen. However, if we want to ensure that an activity is applied in a situation reached with uncontrollable events, we can do so using XorSplit nodes and branching conditions.

We also impose a restriction on consecutive branching conditions: for any sequence of directly connected XorSplit nodes, the conditions must be consistent. Formally, let  $M = \langle N, E, l, t, c \rangle$  be a process model defined over  $O$  and  $\mathcal{A}$ , such that  $n_1, \dots, n_k \in N_C$ ,  $n_{k+1} \in N_A$ , and  $\forall i, 1 \leq i \leq k, t(n_i) = XorSplit$ , and  $(n_i, n_{i+1}) \in E$ . Let  $\varphi_i$  be the branch condition  $c((n_i, n_{i+1}))$ , if this condition is defined, and  $\top$  otherwise. The conditions are consistent if there exists at least one configuration of  $O$  which satisfies  $\varphi_1 \wedge \dots \wedge \varphi_k$ .

**Example** Figure 3.5 shows the process model in our scenario, defined on the domain objects in Figure 3.4. Activity preconditions and effects are denoted with  $P:\dots$ , respectively  $E:\dots$ . We defined a constraint  $\neg damaged^s(h)$

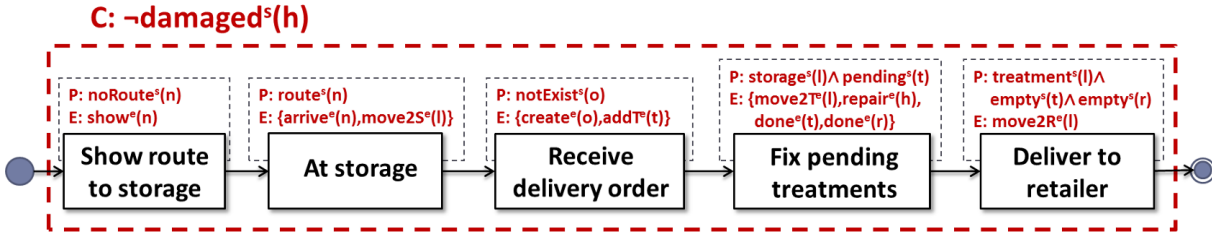


Figure 3.5: Car process model

on all the activity nodes, as a shortcut for saying that each activity includes this formula in its precondition.

Formally, our set of activities is  $\mathcal{A} = \{Show\ route\ to\ storage, At\ storage, Receive\ delivery\ order, Fix\ pending\ treatments, Deliver\ to\ retailer\}$ , where for example the activity *Show route to storage* is  $\langle Show\ route\ to\ storage, \neg damaged^s(h) \wedge noRoute^s(n), \{show^e(n)\} \rangle$ . To avoid confusions between activities in our future examples, where we refer to activities using only their id, we have used the name of the activity to denote also the activity id.

The process model  $M$  defined over  $O$  and  $\mathcal{A}$  is then a tuple  $\langle N, E, l, t, c \rangle$  where:

- $N_A = \{n_1, n_2, n_3, n_4, n_5\}$ ;
- $N_C = \emptyset$ ;
- $E = \{(n_1, n_2), (n_2, n_3), (n_3, n_4), (n_4, n_5)\}$ ;
- $l(n_i)$  is the  $i^{th}$  activity in  $\mathcal{A}$ , for all  $i, 1 \leq i \leq 5$ ;
- $t(n_1) = Start, t(n_2) = t(n_3) = t(n_4) = Normal$ , and  $t(n_5) = End$ ;
- $c$  is undefined for all  $e \in E$ .



### 3.3 Execution Concepts

In this section, we introduce several concepts related to the execution of process-based applications. Since checking that a process model satisfies a goal involves checking that every execution of the process model satisfies the goal, the concepts introduced in this section serve also to formally define the notion of goal satisfiability.

#### 3.3.1 Traces

Process instances contain information such as the point in time an activity was executed, or which activities are ready for execution. The trace of a process instance is a sequence of activities appearing in the process model, which reflects the order in which the activities were executed in the process instance. Therefore, consecutive activities in the trace must appear in the process model as labels of activity nodes which are either connected directly or through a sequence of control nodes.

#### Definition 5 (Trace)

*Let  $O$  be a set of domain objects,  $\mathcal{A}$  a set of activities defined over  $O$ , and  $M = \langle N, E, l, t, c \rangle$  a process model defined over  $O$  and  $\mathcal{A}$ . A trace  $\pi$  on  $M$  is a sequence of activities  $\langle a_1, \dots, a_k \rangle$ ,  $k \in \mathbb{N}$ , such that  $\forall i, 1 \leq i \leq k, a_i \in \mathcal{A}$ , and  $\exists n_i \in N, l(n_i) = a_i$ , with  $t(n_1) = \text{Start}$ . For  $i < k$ ,  $n_i$  and  $n_{i+1}$  are such that either  $(n_i, n_{i+1}) \in E$ , or  $\exists n'_1, \dots, n'_j \in N_C, j \geq 1$ , and  $(n_i, n'_1), (n'_1, n'_2), \dots, (n'_j, n_{i+1}) \in E$ . Activities can occur multiple times due to loops and duplicate nodes. A trace is complete if  $t(n_k) = \text{End}$ , and partial otherwise.*

We denote with  $\text{Traces}(M)$  the set of complete traces that can be produced by a process model  $M$ .  $\text{Traces}(M)$  can be an infinite set if  $M$  contains loops.

**Example** For our car process model in Figure 3.5, the set of complete traces contains only one trace  $\pi = \langle \textit{Show route to storage}, \textit{At storage}, \textit{Receive delivery order}, \textit{Fix pending treatments}, \textit{Deliver to retailer} \rangle$ .

### 3.3.2 Configurations

A global state of the domain objects at a certain point during the execution of a process instance is defined by the state of each domain object. We call such a global state a *configuration*.

#### Definition 6 (Configuration)

Let  $O$  be a set of domain objects. A configuration  $\gamma$  of  $O$  is a total function which maps each domain object  $o \in O, o = \langle L, L^0, \mathcal{E}, T \rangle$  to a state in  $L$ . If  $\gamma$  maps every domain object  $o$  to an initial state in  $L^0$  then  $\gamma$  is an initial configuration.

Since every domain object state is described by a proposition in  $P_S$ , a configuration corresponds to an interpretation of the propositions in  $P_S$ , which assigns the value true or false depending on whether the domain object is in that particular state or not in the given configuration. For every configuration  $\gamma$  there will be exactly one corresponding interpretation  $I_\gamma$  over  $P_S$ . Slightly abusing the notation, we say that a configuration  $\gamma$  satisfies a boolean expression  $b \in \textit{Bool}(P_S)$ ,  $\gamma \models b$ , if the corresponding interpretation  $I_\gamma$  satisfies  $b$ , i.e., if  $I_\gamma \models b$ .

If  $\gamma$  and  $\gamma'$  are configurations of  $O$ , we say that  $\gamma'$  is *directly reachable* from  $\gamma$  if for every domain object  $o \in O$  for which  $\gamma(o)$  is different from  $\gamma'(o)$ , there exists a sequence of transitions in  $o$  from  $\gamma(o)$  to  $\gamma'(o)$  only on uncontrollable events. Formally,  $\gamma'$  is directly reachable from  $\gamma$  if for all  $o \in O, o = \langle L, L^0, \mathcal{E}, T \rangle$  such that  $\gamma(o) = s$ ,  $\gamma'(o) = s'$ , and  $s \neq s'$ , there exist  $e_1, \dots, e_k \in \mathcal{E}_U$  and  $s_1, \dots, s_{k-1} \in L$  such that  $(s, e_1, s_1), \dots, (s_{k-1}, e_k, s') \in T$ . Note that this relation between configura-

tions is reflexive, since every configuration is directly reachable from itself. This relation is also transitive, i.e., if  $\gamma_1$  is directly reachable from  $\gamma_2$ , and  $\gamma_2$  is directly reachable from  $\gamma_3$ , then also  $\gamma_1$  is directly reachable from  $\gamma_3$ .

If  $\langle a, pre, eff \rangle$  is an activity defined over  $O$ , and  $\gamma$  is a configuration of  $O$ , we say that the activity  $\langle a, pre, eff \rangle$  is *applicable* in configuration  $\gamma$  if there exists a configuration of  $O$   $\gamma'$  such that  $\gamma' \models pre$ , and  $\gamma'$  is directly reachable from  $\gamma$ .

Moreover, we say that a configuration  $\gamma_{eff}$  is *reachable by applying* the activity  $\langle a, pre, eff \rangle$  in  $\gamma$  if:

- $\langle a, pre, eff \rangle$  is applicable in  $\gamma$ . Let  $\gamma_{pre}$  be the actual configuration satisfying *pre*.
- by applying the events in *eff* to  $\gamma_{pre}$  we obtain  $\gamma_{eff}$ . In other words, for every domain object  $o \in O$ ,  $o = \langle L, L^0, \mathcal{E}, T \rangle$ , if there exists an event  $e \in eff$  such that  $e \in \mathcal{E}$ , then there exists a transition from  $\gamma_{pre}(o)$  to  $\gamma_{eff}(o)$  on  $e$  in  $T$ . Otherwise,  $\gamma_{eff}(o) = \gamma_{pre}(o)$ .

### 3.3.3 Executions

Checking whether a process model  $M$  satisfies a goal  $G$  involves checking if the configurations of the domain objects which are reached during each complete execution satisfy the properties specified by the goal. To formally define this goal satisfaction criterion, we first introduce the notion of execution. Informally, an execution of a process model over a set of domain objects will be a complete trace interleaved after each activity with information about the configuration of the domain objects.

#### Definition 7 (Execution)

Let  $O$  be a set of domain objects,  $\mathcal{A}$  a set of activities defined over  $O$ , and  $M = \langle N, E, l, t, c \rangle$  a process model defined over  $O$  and  $\mathcal{A}$ . An execution of

$M$  is an alternating sequence of configurations and activities represented as  $\gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k$ , where:

- $a_1, \dots, a_k \in \mathcal{A}$ , and  $\langle a_1, \dots, a_k \rangle$  is a trace on  $M$ ,
- $\gamma_0, \dots, \gamma_k$  are configurations of  $O$ , with  $\gamma_0$  an initial configuration,
- $\forall i, 1 < i \leq k$ , if  $n_{i-1}, n_i \in N$  are the nodes corresponding to  $a_{i-1}$  and respectively  $a_i$ , and  $a_i = \langle \text{name}_i, \text{pre}_i, \text{eff}_i \rangle$ , then:
  - if  $(n_{i-1}, n_i) \in E$ , then  $\gamma_i$  is reachable from  $\gamma_{i-1}$  by applying  $a_i$ ,
  - otherwise,  $\gamma_i$  is reachable from  $\gamma_{i-1}$  by applying  $a'_i = \langle \text{name}_i, \text{pre}'_i, \text{eff}_i \rangle$ , where  $\text{pre}'_i = \text{pre}_i \wedge \text{cond}$ , and  $\text{cond}$  is the conjunction of conditions on the edges between  $n_{i-1}$  and  $n_i$ .

An execution is complete if the corresponding trace  $\langle a_1, \dots, a_k \rangle$  is complete, and partial otherwise.

Note that Definition 12 covers also executions for which at some step  $i$ , the edges connecting nodes  $n_{i-1}$  and  $n_i$  have conditions, and at the same time these conditions are not satisfied in  $\gamma_i$ , and are satisfied in a configuration  $\gamma'_i$  which is directly reachable from  $\gamma_i$ . This is the case when a condition holds because one or more uncontrollable events have been triggered in the corresponding domain objects.

**Example** In our scenario, a possible complete execution allowed by the process model in Figure 3.5 is:  $\gamma_0 \xrightarrow{\text{Show route to storage}} \gamma_1 \xrightarrow{\text{At storage}} \gamma_2 \xrightarrow{\text{Receive delivery order}} \gamma_3 \xrightarrow{\text{Fix pending treatments}} \gamma_4 \xrightarrow{\text{Deliver to retailer}} \gamma_5$  where the configurations  $\gamma_0, \dots, \gamma_5$  are given in Table 3.1. The executions allowed by our process model will differ only in terms of the value that the configurations assign to the domain object Service Station Queue.

For every configuration  $\gamma_i$  there is a corresponding configuration  $\gamma'_i$  for which  $\gamma'_i(h) = \text{damaged}$ , which is directly reachable from  $\gamma_i$ . However,

$o$	Car Loc. (l)	Car Health (h)	Treatment Sched. (t)	Repair Sched. (r)	Service St. Queue (q)	Car Navi. (n)	Delivery Order (o)
$\gamma_0(o)$	terminal	ok	empty	empty	30%	noRoute	notExist
$\gamma_1(o)$	terminal	ok	empty	empty	30%	route	notExist
$\gamma_2(o)$	storage	ok	empty	empty	30%	noRoute	notExist
$\gamma_3(o)$	storage	ok	pending	empty	30%	noRoute	created
$\gamma_4(o)$	treatment	ok	empty	empty	30%	noRoute	created
$\gamma_5(o)$	retailer	ok	empty	empty	30%	noRoute	created

Table 3.1: Configurations along one execution in the car logistics scenario

the activity at step  $i$  is not applicable to configuration  $\gamma'_i$ . This is ensured by the fact that the precondition of each activity includes the formula  $\neg\text{damaged}^S(h)$ . Therefore, no execution allowed by our process model can go through such a configuration.

We denote with  $Exec(M)$  the set of complete executions that can be produced by a process model  $M$ . Similar to  $Traces(M)$ ,  $Exec(M)$  can be an infinite set if  $M$  contains loops.

Two process models  $M_1$  and  $M_2$  are considered to be the same  $M_1 \equiv M_2$  if they have the same complete executions, i.e., if  $Exec(M_1) = Exec(M_2)$ .

### 3.3.4 Goal Satisfaction

A process model  $M$  satisfies a goal statement  $\psi_0 \implies (\psi_1 \succ \dots \succ \psi_n)$  if every complete execution of  $M$  is such that if a configuration satisfying  $\psi_0$  is reached at some point during the execution, then the last configuration reached satisfies at least one of the formulas  $\psi_1, \dots, \psi_n$ .  $M$  satisfies a goal  $G$  if it satisfies all the goal statements in  $G$ .

#### Definition 8 (Goal Satisfaction)

Let  $O$  be a set of domain objects,  $\mathcal{A}$  a set of activities, and  $G$  a goal defined over  $O$ . Let  $M = \langle N, E, l, t, c \rangle$  be a process model defined over  $O$  and  $\mathcal{A}$ .

Let  $g : \psi_0 \implies (\psi_1 \succ \dots \succ \psi_n)$  be a goal statement in  $G$ , and

$\omega = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k$  a complete execution of  $M$ .  $\omega$  satisfies  $g$  iff it is such that if  $\psi_0$  is satisfied in any configuration traversed up to step  $k$ , then  $\gamma_k$  satisfies at least one of  $\psi_1, \dots, \psi_n$ .

$M$  satisfies  $g$  if every complete execution of  $M$  satisfies  $g$ .  $M$  satisfies  $G$  if it satisfies every goal statement  $g \in G$ .

In case a complete execution is such that at some step  $i < k$ , the activity  $a_i$  was not applied to configuration  $\gamma_{i-1}$ , but to a directly reachable configuration  $\gamma'_{i-1}$ , then not only  $\gamma_{i-1}$ , but also  $\gamma'_{i-1}$  counts as a traversed configuration. Therefore, if  $\gamma'_{i-1}$  satisfies  $\psi_0$ , then the last configuration  $\gamma_k$  in the execution must satisfy at least one of  $\psi_1, \dots, \psi_n$ .

**Example** Our process model  $M$  from Figure 3.5 satisfies the goal  $G_1$  introduced in Section 3.2.2. We recall that  $G_1$  contains one goal statement  $\top \implies ok^s(h) \wedge retailer^s(l)$ . For every complete execution of  $M$ , all of the configurations reached during the execution satisfy  $\top$ . Then, since configurations along different executions differ only in terms of the value associated to the object Service Station Queue (q), the last configuration in every complete execution will have the same values as  $\gamma_5$  in Table 3.1, except for the value for q. Therefore, every last configuration satisfies  $ok^s(h) \wedge retailer^s(l)$ .

### 3.4 Adaptation Concepts

Process instances can be adapted structurally while they are running: activities can be added or removed, or they can be re-executed. Such adaptations occur ad-hoc, in order to deal with unexpected situations or exceptions. Moreover, the adaptation of one process instance does not affect other (currently running or new) process instances.

We start from the premise that adaptation is triggered by the violation

of a design-time assumption, e.g., a constraint is violated, or a goal cannot be satisfied. Based on this premise, plugging an adaptation into the process model in the evolution phase results in an enhancement of the process model. This in turn allows us to plug into the process model several adaptations at the same time. If plugging an adaptation could result in a contraction of the model, other adaptations referring to the deleted parts would no longer be applicable. Note that adaptations can also contain design-time assumptions, which, if violated, will trigger re-adaptations.

A second premise is that adaptation is performed in order to reach the goal. Based on this second premise, in the evolution phase we know, without having to reason on the domain, that there is at least one situation for which an instance adaptation can be plugged into the process model, such that the resulting process model satisfies the goal. This is important in case each adaptation must be plugged-in only for a particular situation, since it allows us to obtain the updated model more efficiently, without any verification.

These two premises are more restrictive than that of existing process evolution approaches (e.g.,[47, 53, 87]), where adaptation can occur without the violation of a constraint, as a result of human intervention, and there are no domain-level restrictions on the adaptations that can be performed.

### 3.4.1 Adaptation Operation

Given a process model  $M$ , we consider a single generic adaptation operation tailored for instance adaptations:

$$\mathit{adapt}(M^a, \mathit{from}, \mathit{to})$$

Here,  $M^a$  is a process model, and  $\mathit{from}, \mathit{to}$  are nodes in  $M$ . This operation allows to interrupt the execution of  $M$  after having completed  $\mathit{from}$ , and

execute a different model  $M^a$ . After executing  $M^a$ , the control is given back to  $M$  and execution is resumed from  $to$ . If  $to$  has already been executed, the operation is a jump back. In case it has not been executed, if  $from$  and  $to$  are connected directly, the operation is a simple insertion, otherwise a jump forward.

**Definition 9 (Adaptation)**

Let  $O$  be a set of domain objects,  $\mathcal{A}$  a set of activities defined over  $O$ , and  $M = \langle N, E, l, t, c \rangle$  a process model defined over  $O$  and  $\mathcal{A}$ . An adaptation of  $M$  is an operation  $\Delta = \text{adapt}(M^a, from, to)$ , such that:

- $M^a = \langle N^a, E^a, l^a, t^a, c^a \rangle$  is a process model defined over  $O$  and  $\mathcal{A}$ , for which  $N \cap N^a = \emptyset$ ;
- $from, to \in N$ ;
- $to$  is not part of an AND-block.

Without loss of generality, we consider that adaptations cannot contain nodes from the main model, and, to prevent deadlocks, that jumping in a parallel branch is not possible. An equivalent adaptation satisfying these conditions can always be constructed by duplicating relevant nodes.

An adaptation operation cannot be applied arbitrarily to the process model  $M$ . In particular, an adaptation operation is applicable to a particular partial execution  $\omega$  of  $M$  only if  $from$  is the last completed node in the execution  $\omega$ , there is an execution of  $M^a$  starting from the last configuration in  $\omega$ , and any resulting complete execution satisfies the goal of process model  $M$ .

**Definition 10 (Adaptation Applicability)**

Let  $M = \langle N, E, l, t, c \rangle$  a process model and  $G$  a goal such that  $M$  satisfies  $G$ . Let  $\omega = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{i-1} \xrightarrow{a_i} \gamma_i$  be a partial execution of  $M$ , and  $\Delta =$



$\text{adapt}(M^a, \text{from}, \text{to})$  an adaptation of  $M$ . We say that  $\Delta$  is applicable to  $M$  on  $\omega$  iff:

- $l(\text{from}) = a_i$ ;
- there exists at least one execution of  $M^a$  starting from  $\gamma_i$ ,  
 $\gamma_i \xrightarrow{a_{i+1}} \gamma_{i+1} \dots \gamma_{j-1} \xrightarrow{a_j} \gamma_j, j \geq i$ ;
- any complete execution  $\gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_j \xrightarrow{a_{j+1}} \gamma_{j+1} \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k, k > j$ ,  
 where  $a_{j+1} = l(\text{to})$  and  $a_{j+1}, \dots, a_k$  are activities in  $M$ , satisfies  $G$ .

We represent adaptation as a single operation rather than using change patterns as proposed by Weber et al. in [116], to emphasize that it is a solution to an exceptional situation encountered during the execution of a process instance. If it were represented as multiple change operations, the meaning of an adaptation as an indivisible solution would be lost. As a solution to an exceptional situation, the adaptation is always performed at the point of failure and corresponds to a one-time change, i.e., if  $\text{from}$  is reached again,  $M^a$  is not re-executed (in [116], this is modeled as a temporary instance change design choice). Finally, an adaptation can be applicable to multiple executions of the same process model, in different configurations, and therefore does not include the cause.

### 3.4.2 Adapted Traces and Executions

We can now generalize our definitions of traces and executions to cover adapted process instances. Traces of adapted instances are traces which reflect not only the order in which activities from a process model  $M$  were executed, but also the order in which adaptation operations were applied to  $M$  and the execution order of the activities in the process models corresponding to these adaptations.

**Definition 11 (Adapted Trace)**

Let  $O$  be a set of domain objects,  $\mathcal{A}$  a set of activities defined over  $O$ , and  $M = \langle N, E, l, t, c \rangle$  a process model defined over  $O$  and  $\mathcal{A}$ . Let  $\sigma = \Delta_1, \dots, \Delta_n$  be a sequence of adaptations of  $M$  such that  $\forall i, 1 \leq i \leq n, \Delta_i = \text{adapt}(M_i^a, \text{from}_i, \text{to}_i)$ , and  $M_i^a = \langle N_i^a, E_i^a, l_i^a, t_i^a, c_i^a \rangle$  is a process model defined over  $O$  and  $\mathcal{A}$ .

A trace  $\pi$  on  $M$  adapted by  $\sigma$  is a sequence  $\langle a_1, \dots, a_k \rangle$ , such that  $\forall j, 1 \leq j \leq k, a_j \in \mathcal{A}$ , and  $\exists n_j \in N \cup \bigcup_{1 \leq i \leq n} N_i^a$ , such that  $l(n_j) = a_j$ , with  $t(n_1) = \text{Start}$ . The order of  $a_j$  in  $\pi$  reflects the temporal order in which adaptations from  $\sigma$  were applied, and activities from  $M, M_1^a, \dots, M_n^a$  were completed. Activities can occur multiple times due to loops and duplicate nodes. A trace is complete if  $t(n_k) = \text{End}$ , and is partial otherwise.

Similarly, the execution of an adapted process instance reflects not only the order in which activities from  $M$  and configurations were traversed, but also the order in which adaptations were applied and executed.

**Definition 12 (Adapted Execution)**

Let  $M = \langle N, E, l, t, c \rangle$  be a process model and  $\sigma = \Delta_1, \dots, \Delta_n$  a sequence of adaptations of  $M$ .

An execution  $\omega$  of  $M$  adapted by  $\sigma$  is an alternating sequence of configurations and activities represented as  $\gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k$ , where:

- $a_1, \dots, a_k \in \mathcal{A}$ , and  $\langle a_1, \dots, a_k \rangle$  is a trace on  $M$  adapted by  $\sigma$ ,
- $\gamma_0, \dots, \gamma_k$  are configurations of  $O$ , with  $\gamma_0$  an initial configuration,
- there exist  $0 \leq j_1 < \dots < j_n \leq k$ , such that if  $\omega_i$  is the prefix of  $\omega$  up to step  $j_i$ ,  $1 \leq i \leq n$ , then:
  - $\omega_1$  is an execution of  $M$ , and
  - $\forall i, 1 \leq i \leq n, \Delta_i$  is applicable to  $M$  on  $\omega_i$ .

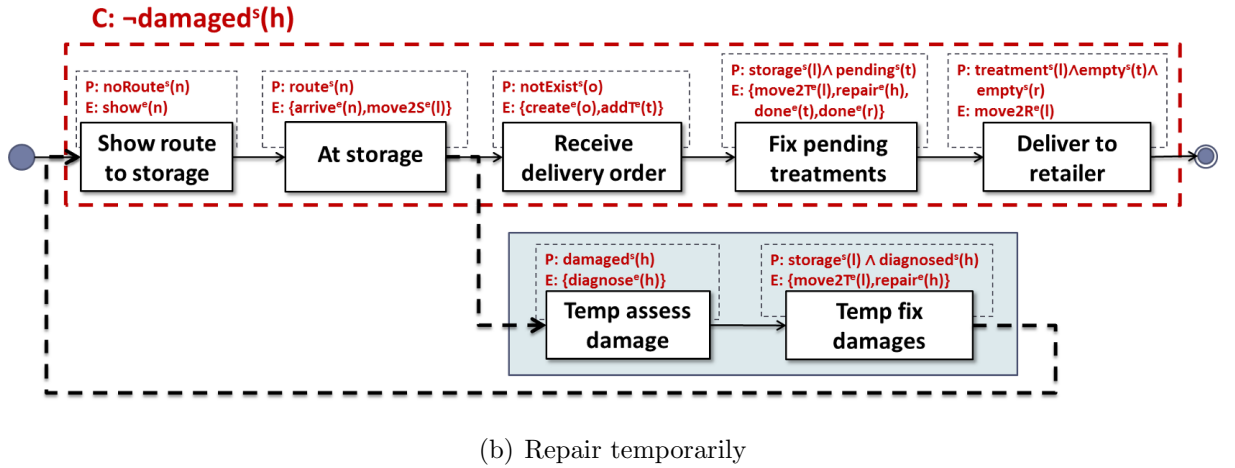
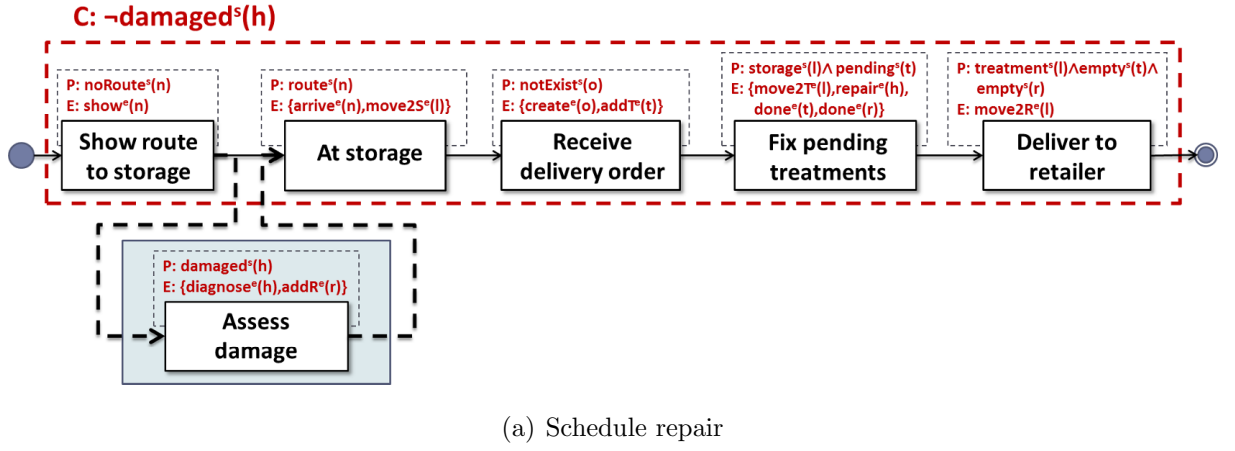


Figure 3.6: Adaptation operations

An execution is complete if the trace  $\langle a_1, \dots, a_k \rangle$  is complete, and partial otherwise.

The definition of adapted executions only specifies that adaptations were applied at some point during the execution, and does not require adaptations to be completely executed once they are applied. In fact, the execution of an adaptation model can be interrupted by the application of the next adaptation. This is due to the fact that adaptations models can also contain constraints, which, if violated, trigger re-adaptation.

**Example** Let  $M$  be the process model from Figure 3.5, and consider the adaptations in Figure 3.6. The first adaptation, *Schedule repair*, is applicable to  $M$  on execution  $\omega_1 = \gamma_0 \xrightarrow{\text{Show route to storage}} \gamma_1$ .

Similarly, *Repair temporarily* is applicable to  $M$  on execution  $\omega_2 = \gamma_0 \xrightarrow{\text{Show route to storage}} \gamma_1 \xrightarrow{\text{At storage}} \gamma_2$ .

Consider now the execution of  $M$  adapted by *Schedule repair*,  $\omega_3 = \gamma_0 \xrightarrow{\text{Show route to storage}} \gamma_1 \xrightarrow{\text{Assess damage}} \gamma_2 \xrightarrow{\text{At storage}} \gamma_3$ . *Repair temporarily* is applicable to  $M$  also on  $\omega_3$ , which corresponds to the case when the constraint that the car should not be damaged is violated a second time.

### 3.5 Discussion

In this Chapter, we presented a framework designed to support the execution, adaptation, and evolution of process-based applications. We also introduced a formal model suitable for representing a process-based application. This model includes a formal representation of the business logic and the domain knowledge, as well as the relation between the business logic and the domain knowledge. In [103], we used a variation of this formal model for composing overlapping process fragments described in APFL (Adaptable Pervasive Flow Language), an extension of WS-BPEL [2]. In [13], we used this formal model to outline a method for analysing adapted process instances, for which we proposed different evolution strategies.

Our formal model was inspired by several approaches dealing with service composition presented in [10, 43, 12]. In particular, for domain objects we used the same representation as the one used for object diagrams in [43], and context property diagrams in [12]. For annotating processes, our approach is based on the formalisms used in [12] and [103], which allow (without enforcing) each activity to be annotated with precondition and effects, where the precondition is a set of allowed configurations (which we

expressed as boolean formulas) and the effects are sets of events.

Our definition of process goals is based on the composition requirements defined in [10, 43]. The composition requirements in [10] allow formulas to contain also propositions referring to events in the object diagrams; in [43] these composition requirements are further enhanced with a try-catch construct, which allows to define additional reactions in case of failure.

In our formal model, we used a restricted version of these composition requirements, in which goals to refer only to states in the domain objects. The reason is that while the requirements in [10, 43] are used for composing services which interact with one another and with the user, we used the goals to verify properties of process models. Our setting is therefore much more static, where we would rather leave it up to the process designer or adaptation mechanism to specify how to react to exceptions or failure, and specify instead only what properties the process tries to achieve or maintain. However, since we use the same representation for our domain objects as the object diagrams in [43], our goal language and the goal satisfaction criteria could in principle be extended to cover the full composition requirements in [43].

For modeling processes, we used a graph-based representation which allows to capture basic control-flow patterns supported by most of the process management systems [109]. Since our notions of goal satisfaction, adaptation, and evolutionary correction (discussed in Chapter 4) are based on the notions of trace and execution, the set of control-flow patterns allowed for modeling processes can easily be extended. This will affect the definitions of trace and execution, but not the goal satisfaction, adaptation, and correction.



# Chapter 4

## Corrective Evolution

In this Chapter, we define the corrective evolution problem and discuss the challenges involved in solving this problem automatically.

With corrective evolution, the idea is to integrate several adaptations into a process model at the same time, and ensure that the resulting process models continue to satisfy the goal of the original process model. Since each adaptation must be plugged in at a certain point in the process model and for a certain condition, we first group the adaptation, plug-in point, and condition into a new concept called correction. We then define the corrective evolution problem as the problem of iteratively correcting a process model with  $n$  corrections, such that the final process model satisfies the original goal.

### 4.1 Corrections

To integrate a process instance adaptation into a process model, we need to specify the point in the process model where the adaptation must be plugged in, and the condition under which it must be plugged in. While there is some freedom in choosing these plug-in points and conditions, they cannot be arbitrary. In this Section, we first discuss the restrictions on the possible plug-in points and conditions for an adaptation. We then

introduce the concept of a correction, which gives us a convenient way to group the adaptation to be integrated in the process model, together with the plug-in point and condition. Finally, we discuss the effects of applying one correction to a process model in terms of resulting executions.

### 4.1.1 Adaptation Plug-in Point and Condition

When integrating an instance adaptation into a process model, the point where the adaptation must be plugged in and the condition under which it must be plugged in do not need to correspond exactly to the process instance or instances that were adapted. The basic idea is that we want to allow as much freedom as possible for selecting the plug-in point and condition. However, neither the plug-in point nor the condition can be arbitrary, they must satisfy some minimal restrictions for the insertion of the adaptation in the process model to be possible. These minimal restrictions are that the adaptation must be applicable on any partial execution of the process model which leads to the plug-in point, and which is such that the last configuration (or a configuration directly reachable from the last) satisfies the condition.

We consider that the plug-in point is specified as a set of traces, and that the plug-in condition selects from the executions possible along these traces only executions for which the instance adaptation is applicable. For a process model  $M$  and an adaptation of  $M$ ,  $\Delta = \text{adapt}(M^a, \text{from}, \text{to})$ , the minimum restriction for the plug-in point is that for each trace to the plug-in point, the last activity in the trace corresponds to the *from* node in the adaptation.

**Example** Consider the process model  $M$  from Section 3.2.3, and the *Repair temporarily* adaptation introduced in Section 3.4. The *from* node for *Repair temporarily* is the node in  $M$  labeled with activity *At storage*.



Therefore, the plug-in points for *Repair temporarily* are given by traces or combination of traces which end with activity *At storage*. The simplest plug-in point is given by the trace  $\pi_1 = \langle \textit{Show route to storage}, \textit{At storage} \rangle$ .

However, if applied to  $M$  on  $\pi_1$ , *Repair temporarily* introduces a new trace which also finishes with *At storage*. This trace is:  $\pi_2 = \langle \textit{Show route to storage}, \textit{At storage}, \textit{Temp assess damage}, \textit{Temp fix damages}, \textit{Show route to storage}, \textit{At storage} \rangle$ . Therefore, a different point where *Repair temporarily* can be plugged in  $M$  is after both traces  $\pi_1$  and  $\pi_2$ . Because of the jump back realized by *Repair temporarily*, there are in fact an infinity of points where the adaptation could be plugged in the process model  $M$ .

Given a plug-in point, there are several restrictions for selecting the condition:

1. the condition must be possible at the selected plug-in point. Therefore, at least one configuration reachable at the plug-in point must satisfy the condition.
2. if the condition holds, then the next steps in the original process model cannot be executed. In other words, the next activities in the original model are not applicable in any plug-in point configuration which satisfies the condition.
3. if the condition holds, the adaptation can be applied. From every plug-in point configuration which satisfies the condition, there must be at least one execution of the process model included in the adaptation.

Given a plug-in point, we refer to conditions which satisfy the first two restrictions as *deviating conditions*. In other word, given a plug-in point in the process model, a deviating condition is a condition which is not accounted for in the process model, for which there is no specification in the model.

**Definition 13 (Deviating Condition)**

Let  $M$  be a process model and  $\pi = \langle a_1, \dots, a_k \rangle$  a partial trace of  $M$ . Let  $\varphi$  be a boolean expression from  $Bool(P_S)$ . Then  $\varphi$  is a deviating condition for  $M$  and  $\pi$  if:

- there exists an execution of  $M$   $\omega = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k$  such that  $\gamma_\varphi \models \varphi$  and  $\gamma_\varphi$  is directly reachable from  $\gamma_k$ . Let  $\Omega$  be the set of all such executions.
- for every execution  $\omega \in \Omega$ , no next activity in  $M$  is applicable to  $\gamma_\varphi$ .

$\varphi$  is a deviating condition for  $M$  and an activity node  $n$  of  $M$  if it is a deviating condition for  $M$  and every trace of  $M$  leading to  $n$ .

If a deviating condition satisfies also the third restriction, this is a *plug-in condition* for applying an adaptation to the process model at the plug-in point.

**Definition 14 (Plug-in Condition)**

Let  $M$  be a process model and  $\Delta = \text{adapt}(M^a, \text{from}, \text{to})$  an adaptation of  $M$ . Let  $\pi = \langle a_1, \dots, a_k \rangle$  be a partial trace of  $M$ , with  $a_k = l(\text{from})$ . Let  $\varphi$  be a boolean expression from  $Bool(P_S)$ . Then  $\varphi$  is a plug-in condition for applying  $\Delta$  to  $M$  on  $\pi$  if:

- $\varphi$  is a deviating condition for  $M$  and  $\pi$ ;
- for every execution  $\omega = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k$  of  $M$  such that  $\gamma_\varphi \models \varphi$  and  $\gamma_\varphi$  is directly reachable from  $\gamma_k$ ,  $\Delta$  is applicable to  $M$  on  $\omega$ .

**Example** Consider again the process model  $M$  from Section 3.2.3, and the *Repair temporarily* adaptation from Section 3.4. Our plug-in point is given by the trace  $\pi_1 = \langle \text{Show route to storage}, \text{At storage} \rangle$ . The configurations  $\gamma$  reached after executing the activities in  $\pi_1$  are such that  $\gamma(l) =$

storage,  $\gamma(h) = ok, \gamma(t) = empty, \gamma(r) = empty, \gamma(r) = noRoute, \gamma(o) = notExist$ , and  $\gamma(q) \in \{empty, 10\%, \dots, 90\%, full\}$ .

Assume that for all the process instances for which *Repair temporarily* has been applied after executing  $\pi_1$ , the configurations  $\gamma$  in which the adaptation was applied are such that  $\gamma(h) = damaged$  and  $\gamma(q) \in \{empty, 10\%\}$ . The condition  $\varphi_1 = damaged^s(h) \wedge (empty^s(q) \vee 10\%^s(q))$  which corresponds to these configurations is a plug-in condition for applying *Repair temporarily* to  $M$  on  $\pi_1$ . The restrictions are satisfied:

- $\varphi_1$  is possible after  $\pi_1$ , since there exist executions of  $M$  which correspond to  $\pi_1$ ,  $\omega = \gamma_0 \xrightarrow{\text{Show route to storage}} \gamma_1 \xrightarrow{\text{At storage}} \gamma_2$ , and for which a configuration  $\gamma_{\varphi_1}$  directly reachable from  $\gamma_2$  satisfies  $\varphi_1$ . Let  $\Omega$  be the set of such executions.
- for every execution  $\omega \in \Omega$ , the next activity *Receive delivery order* is not applicable to  $\gamma_{\varphi_1}$ , since its precondition includes  $\neg damaged^s(h)$ .
- *Repair temporarily* is applicable to every execution  $\omega \in \Omega$ :
  - there is at least one possible execution  $\gamma_{\varphi_1} \xrightarrow{\text{Temp assess damage}} \gamma' \xrightarrow{\text{Temp repair damage}} \gamma''$ , where:
    - \*  $\gamma'(h) = diagnosed$  and  $\forall o \in O, o \neq h, \gamma'(o) = \gamma_{\varphi_1}(o)$ ,
    - \*  $\gamma''(h) = ok, \gamma''(l) = treatment$ , and  $\forall o \in O, o \notin \{h, l\}, \gamma''(o) = \gamma_{\varphi_1}(o)$ .
  - every complete execution  $\omega' = \gamma_0 \xrightarrow{\text{Show route to storage}} \gamma_1 \xrightarrow{\text{At storage}} \gamma_2 \xrightarrow{\text{Temp assess damage}} \gamma_3 \xrightarrow{\text{Temp repair damage}} \gamma_4 \xrightarrow{\text{Show route to storage}} \gamma_5 \dots$  satisfies  $G_1$ .

The plug-in condition can also be more general or more specific than  $\varphi_1$ . Some more general conditions for which the restrictions are satisfied are for example  $\varphi_2 = damaged^s(h) \wedge (empty^s(q) \vee 10\%^s(q) \vee 20\%^s(q))$  or

$\varphi_3 = \text{damaged}^s(h)$ . An example of a more specific plug-in condition is  $\varphi_4 = \text{damaged}^s(h) \wedge \text{empty}^s(q)$ .

Note that the condition  $\varphi_5 = \text{empty}^s(q) \vee 10\%^s(q)$ , although more general than  $\varphi_1$ , is not a plug-in condition for applying *Repair temporarily* to  $M$  on  $\pi_1$ . This is because the second restriction is not satisfied: the next activity in  $M$ , *Receive delivery order*, is applicable to configurations reached after applying  $\pi_1$  which satisfy  $\varphi_5$ .

Plug-in conditions are not supposed to be created manually. This is related to the fact that evolution is invoked only after an adaptation operation has been applied multiple times at a certain point during the execution of process instances. After analysing the adapted process instances using an approach such as the one proposed by Wetzstein et al. in [122], we may decide to evolve the process model to include the adaptation at a particular execution point, for some of the configurations which required adaptation. For example, we may want to plug-in the adaptation only for the configurations for which the adaptation has been successful (e.g., the process instances did not need to be re-adapted).

These selected configurations which required adaptation can be automatically grouped into a formula which constitutes a valid plug-in condition. To simplify this formula, we can compare the configurations for which adaptation was necessary with the configurations for which the execution could be completed regularly. The domain objects which are in the same states in the configurations of the adapted and regular executions can then be excluded from the plug-in condition.

### 4.1.2 Correction Applicability

For each instance adaptation that must be inserted in the process model, we can now select a plug-in point and a condition according to the restrictions

described in the previous Section. The combination of adaptation, plug-in point, and condition is called a correction.

**Definition 15 (Correction)**

Let  $M$  be a process model defined over  $O$  and  $\mathcal{A}$ . A correction  $C$  is a tuple  $\langle ct, \pi, \varphi, \Delta \rangle$  such that:

- $ct \in \{\text{strict}, \text{relaxed}, \text{with-conditions}\}$  is the correction type;
- $\pi$  is a partial trace on  $M$ ;
- $\varphi$  is a boolean expression from  $\text{Bool}(P_S)$ ;
- $\Delta = \text{adapt}(M^a, \text{from}, \text{to})$  is an adaptation of  $M$ .

We say that  $C$  is applicable to  $M$  if:

- $\varphi$  is a plug-in condition for applying  $\Delta$  to  $M$  on  $\pi$ ,
- if  $ct \neq \text{strict}$ , then  $\varphi$  is a deviating condition for  $M$  and node  $\text{from}$ .

The point in the process model where the adaptation  $\Delta$  must be plugged in is determined by the correction type  $ct$ , the partial trace  $\pi$ , and the node  $\text{from}$ . For all correction types, the condition  $\varphi$  can occur after the activity node  $\text{from}$ , and  $\Delta$  should be plugged into the process model after  $\text{from}$ , under condition  $\varphi$ .

However, the partial traces for which the condition should be evaluated and the adaptation should be applied are different depending on the correction type  $ct$ . If the correction is strict, both condition and adaptation should be applied only on  $\pi$ . If the correction is relaxed, they should be applied on all the traces leading to  $\text{from}$ . Finally, if relaxed with conditions, they should be applied on one or more traces leading to  $\text{from}$ , but at least on  $\pi$ .

**Definition 16 (Plug-in Point)**

Let  $M$  be a process model and  $C = \langle ct, \pi, \varphi, \Delta \rangle$  a correction applicable to  $M$ . Then the plug-in point for  $C$  is a set of traces  $point_C$  such that:

- if  $ct = strict$ , then  $point_C = \{\pi\}$ ;
- if  $ct = relaxed$ , then  $point_C$  is the set of all traces  $\pi' = \langle a_1, \dots, a_k \rangle$  for which  $l(from) = a_k$ ;
- if  $ct = with-conditions$ , then  $point_C$  is a subset of all the traces  $\pi' = \langle a_1, \dots, a_k \rangle$  for which  $l(from) = a_k$ , and  $\pi \in point_C$ .

**4.1.3 Correction Effects**

Let  $M$  be a process model and  $C = \langle ct, \pi, \varphi, \Delta \rangle$  a correction applicable to  $M$ , with  $\Delta = adapt(M^a, from, to)$ . We denote with  $Exec(M, C)$  the set of complete executions that can be produced by  $M$  corrected by  $C$ .

In  $Exec(M, C)$ , we add new complete executions to  $Exec(M)$ . A new execution  $\omega$  in  $Exec(M, C)$  corresponds to a process instance for which a configuration  $\gamma$  satisfying condition  $\varphi$  is reached at the plug-in point for correction  $C$ . At this point, the process instance is adapted with  $\Delta$ , and our new execution  $\omega$  proceeds with an execution of the adaptation model  $M^a$ . When the execution of  $M^a$  is completed,  $\omega$  continues from the activity corresponding to the node  $to$  in  $M$  and until an end node is reached.

By correcting the process model, the new complete executions do not replace any existing complete executions of  $M$ , and therefore  $Exec(M) \subseteq Exec(M, C)$ . We prove this property formally.

**Lemma 1**

Let  $M$  be a process model and  $C$  a correction applicable to  $M$ . Then  $Exec(M) \subseteq Exec(M, C)$ .

*Proof.* We assume there exists a complete execution  $\omega \in Exec(M)$  such that  $\omega \notin Exec(M, C)$ . Since in  $Exec(M, C)$  we are adding executions to  $Exec(M)$ , this can happen only if  $\omega$  is overwritten by an execution  $\omega'$ .

Assume  $C = \langle ct, \pi, \varphi, \Delta \rangle$ . For  $\omega$  to be overwritten, the plug-in point of  $C$  must be reached in  $\omega$ . Therefore, there exists a partial trace  $\pi' = \langle a_1, \dots, a_k \rangle$  in  $point_C$  such that:

- $\omega = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k \xrightarrow{a_{k+1}} \dots$
- and there exists a configuration  $\gamma_\varphi$  directly reachable from  $\gamma_k$  such that  $\gamma_\varphi \models \varphi$  and  $a_{k+1}$  is applicable in  $\gamma_\varphi$ .

If  $ct = strict$ , then  $\pi' = \pi$  and  $\varphi$  is a plug-in condition for applying  $\Delta$  to  $M$  on  $\pi$ . With Definition 14, this means that for our execution  $\omega$  no next activities in  $M$  are applicable to  $\gamma_\varphi$ .

If  $ct \neq strict$ , then  $\varphi$  is a deviating condition for  $M$  and  $\pi'$ . With Definition 13, also in this case  $\omega$  is such that no next activities in  $M$  are applicable to  $\gamma_\varphi$ .

Since there cannot be an activity  $a_{k+1}$  which is applicable in  $\gamma_\varphi$ , this means that  $\omega$  cannot be overwritten.  $\square$

With corrective evolution, we are interested in retrieving the process model which corresponds to  $Exec(M, C)$ , i.e., a process model  $M'$  such that  $Exec(M') = Exec(M, C)$ . It can be the case that a process model  $M'$  which also satisfies the goal of process model  $M$  does not exist. However, there is always a process model  $M'$  such that  $Exec(M') = Exec(M, C)$ .

**Lemma 2 (Existence of a corrected process model)**

*Given a process model  $M$  and a correction  $C$  applicable to  $M$ , there exists at least one process model  $M'$  such that  $Exec(M') = Exec(M, C)$ .*

*Proof.* By construction.

Assume  $C = \langle ct, \pi, \varphi, \Delta \rangle$ , with  $\Delta = \text{adapt}(M^a, \text{from}, \text{to})$ . If  $ct = \text{strict}$ , then we can obtain a process model  $M'$  by first unfolding  $M$  up to and including the node  $\text{from}$ . We then insert an XorSplit node after the sequence corresponding to trace  $\pi$ . We connect the XorSplit to the initial node in  $M^a$ , and add  $\varphi$  as condition on the control edge. We also connect the XorSplit to the node which was following in the unfolded  $M$  the sequence corresponding to  $\pi$ . We duplicate the fragment in  $M$  starting with node  $\text{to}$ , and connect this duplicated fragment to the end node(s) of  $M^a$ .

If  $ct = \text{relaxed}$ , we can obtain a process model  $M'$  by first inserting an XorSplit node in  $M$  after the node  $\text{from}$ . We connect the XorSplit to the initial node in  $M^a$ , and add  $\varphi$  as condition on the control edge. We also connect the XorSplit to the node which followed  $\text{from}$  in  $M$  (since  $\text{from}$  is an activity node, there is exactly one following node in  $M$ ). We then connect the end node(s) of  $M^a$  to the node  $\text{from}$ .

If  $ct = \text{with-conditions}$ , then there exists at least one process model which corresponds to  $\text{Exec}(M, C)$ , and this is the process model obtained with  $C$  as a strict correction.

Therefore, a process model  $M'$  which corresponds to  $\text{Exec}(M, C)$  can always be constructed from  $M$ , by duplicating nodes.  $\square$

## 4.2 Corrective Evolution Problem

In this Section, we consider the problem of correcting a process model with  $n$  corrections. We first discuss what is a valid ordering of a set of corrections, and then define the corrective evolution problem as the problem of iteratively correcting a process model with a sequence of  $n$  corrections. We examine general properties of a solution to the problem, and explain why the corrective evolution problem is more general than the problem of correcting a process model with only one correction. Finally, we discuss



the challenges involved in solving the problem.

### 4.2.1 Ordering Corrections

Given a process model  $M$  and two corrections  $C_1 = \langle ct_1, \pi_1, \varphi_1, \Delta_1 \rangle$  and  $C_2 = \langle ct_2, \pi_2, \varphi_2, \Delta_2 \rangle$ , we consider that  $C_2$  is dependent on  $C_1$  in two cases. The first case is if the  $from_2$  node in the adaptation  $\Delta_2$  is not a node of the original process model  $M$ , and is a node of the adaptation process model  $M_1^a$  in  $\Delta_1$ . In this case,  $\Delta_2$  must be applied if  $\Delta_1$  fails at some point. The second case is if the trace  $\pi_2$  is not a trace possible on the original model  $M$ , and is instead a trace of  $M$  corrected by  $C_1$ . In this case,  $\Delta_2$  must be applied only if  $\Delta_1$  has been applied first.

With corrective evolution, we want to be able to apply several corrections to a process model at the same time. Since there can be dependencies between corrections, it is important to establish a valid ordering. A valid ordering of a set of corrections must respect the dependencies between corrections, i.e., if  $C_2$  is dependent on  $C_1$ , then  $C_1$  must appear before  $C_2$  in the sequence. Given a set of corrections  $C_1, \dots, C_n$  to be applied to a process model  $M$ , such that for every  $C_i, 1 \leq i \leq n$ , if  $C_i$  is dependent on  $C_j$ , then  $C_j \in \{C_1, \dots, C_n\} \setminus \{C_i\}$ , there exists at least one valid ordering. The reason is that there cannot be circular dependencies between corrections. In the following, we prove this property formally.

#### **Lemma 3 (No circular dependencies)**

*Let  $M$  be a process model and  $C_1, \dots, C_n$  a set of corrections such that  $C_1$  is applicable to  $M$  and, for all  $1 \leq i < n$ ,  $C_{i+1}$  is dependent on  $C_i$ . Then for all  $i, j, 1 \leq i < j \leq n$ ,  $C_i$  is not dependent on  $C_j$ .*

*Proof.* We assume that  $C_i$  is dependent on  $C_j$ . There are three cases. The first case is if  $C_i$  depends on  $C_j$  in that  $\pi_i$  is possible only on the process

model corrected by  $C_1, \dots, C_j$ . In this case,  $\pi_i$  would have to involve also correction  $C_i$  and pass  $M_i^a$ , which is not possible.

The second case is if  $C_i$  is such that node  $from_i$  is from the adaptation model  $M_j^a$  in  $C_j$ , and  $from_j$  in  $C_j$  is from  $M_i^a$ . While there can be a chain of dependent corrections such that the  $from$  node in each correction is a node in a previous correction, to be applied to  $M$ , there must be one correction in this chain which is the start of the chain and which is directly applicable to  $M$ . This is not possible for  $C_i$  and  $C_j$ , since they link to one another.

The last case is if the node  $from_i$  in  $C_i$  is from  $M_j^a$  in  $C_j$ , and in  $C_j$ , the trace  $\pi_j$  is possible only on the process model corrected by  $C_1, \dots, C_i$ . This case is symmetrical to the first. In this case,  $C_j$  would have to involve itself, which is not possible.

Since we derived a contradiction in each of the three cases, it is impossible that  $C_i$  is dependent on  $C_j$ .  $\square$

## 4.2.2 Problem Definition

Starting from a process model which satisfies a goal, the corrective evolution problem is to apply a sequence of corrections to this process model, such that the resulting process model satisfies the original goal.

### Definition 17 (Corrective evolution problem)

Let  $M_0$  be a process model and  $G$  a goal such that  $M_0$  satisfies  $G$ . Let  $C_1, \dots, C_n$  be a sequence of corrections, such that  $\forall i, 1 \leq i \leq n$ :

- $C_i = \langle ct_i, \pi_i, \varphi_i, \Delta_i \rangle$ ,  $\Delta_i = \text{adapt}(M_i^a, from_i, to_i)$ , and  $to_i$  is a node from  $M_0$ ;
- $C_i$  is applicable to  $M_{i-1}$ , and  $M_i$  is a process model such that  $\text{Exec}(M_i) = \text{Exec}(M_{i-1}, C_i)$ .

The corrective evolution problem is to find a process model  $M_n$  such that  $M_n$  satisfies  $G$ .

Without loss of generality, we assume that every adaptation returns the control to  $M_0$  (i.e.,  $to_i$  is a node from  $M_0$ ). If an adaptation  $\Delta_2$  were to return the control to a previous adaptation  $\Delta_1$ , we can obtain an equivalent adaptation  $\Delta'_2$  by duplicating the relevant nodes from  $\Delta_1$ .

**Example** We can now formalize the inputs to the corrective evolution problem in our car logistics scenario:

- process model  $M$  from Section 3.2.3 satisfying the goal  $G_1$  from Section 3.2.2;
- correction  $C_1 = \langle \textit{strict}, \pi_1, \varphi_1, \Delta_1 \rangle$  where:
  - $\pi_1 = \langle \textit{Show route to storage} \rangle$ ,
  - $\varphi_1 = \textit{damaged}^s(h) \wedge (40\%^s(q) \vee \dots \vee \textit{full}^s(q))$ ,
  - $\Delta_1$  is the *Schedule repair* adaptation in Section 3.4;
- correction  $C_2 = \langle \textit{strict}, \pi_2, \varphi_2, \Delta_2 \rangle$  where:
  - $\pi_2 = \langle \textit{Show route to storage}, \textit{Assess damage}, \textit{At storage} \rangle$ ,
  - $\varphi_2 = \textit{damaged}^s(h) \wedge (\textit{empty}^s(q) \vee \dots \vee 70\%^s(q))$ ,
  - $\Delta_2$  is the *Repair temporarily* adaptation in Section 3.4.

### 4.2.3 Solution Properties

The solution to a corrective evolution problem  $M_n$  is an enhancement of the process model  $M_0$ , i.e., every complete execution of  $M_0$  is also a complete execution of  $M_n$ . This property follows directly from Lemma 1.

Then, for every correction  $C_i = \langle \textit{ct}_i, \pi_i, \varphi_i, \Delta_i \rangle$ , there are two possibilities:

- if  $C_i$  is applicable to  $M_0$ , then  $M_n$  can replay all the executions that would result by applying  $C_i$  to  $M_0$ . Note that if  $C_i$  is applicable to  $M_0$ , then it is also applicable to  $M_0$  corrected by  $C_1, \dots, C_{i-1}$ , i.e., to  $M_{i-1}$ .
- $C_i$  is not applicable to  $M_0$ , but it is applicable to  $M_0$  corrected by  $C_{j_0}, \dots, C_{j_k}$ , where  $1 \leq j_0 < \dots < j_k < i$ . This is the case when  $C_i$  is dependent on  $C_{j_0}, \dots, C_{j_k}$ . Then  $M_n$  can replay all the executions that would result by applying  $C_i$  to  $M_0$  corrected by  $C_{j_0}, \dots, C_{j_k}$ .

If we apply only strict corrections, any resulting process model must satisfy the goal  $G$ . This property follows directly from the restrictions on the plug-in point and condition for each adaptation. We now prove this property formally.

**Lemma 4 (Strict corrections do not affect goal satisfaction)**

*Assume a corrective evolution problem defined by a process model  $M_0$ , a goal  $G$ , and a sequence of corrections  $C_1, \dots, C_n$ , such that for all  $i, 1 \leq i \leq n$ ,  $C_i = \langle \text{strict}, \pi_i, \varphi_i, \Delta_i \rangle$ . If for all  $i, 1 \leq i \leq n$ ,  $M_i$  is any process model such that  $\text{Exec}(M_i) = \text{Exec}(M_{i-1}, C_i)$ , then  $M_n$  satisfies  $G$ .*

*Proof.* The proof goes by induction.

**base case.**  $M_0$  satisfies  $G$ .

**inductive step.** We assume  $M_{n-1}$  satisfies  $G$ . From Lemma 1, we know that  $\text{Exec}(M_{n-1}) \subseteq \text{Exec}(M_n)$ . Therefore, every complete execution of  $M_n$  which is also a complete execution of  $M_{n-1}$  satisfies  $G$ . (1)

Let  $\omega$  be a complete execution of  $M_n$  which is not a complete execution of  $M_{n-1}$ . This can only be the case if the adaptation  $\Delta_n$  has been applied. From Definition 15,  $\varphi_n$  is a plug-in condition for applying  $\Delta_n$  to  $M_{n-1}$  on  $\pi_n$ . With Definition 14, this means that  $\Delta_n$  is applicable to every partial

execution of  $M_{n-1}$  which corresponds to  $\pi_n$  and reaches a configuration satisfying  $\varphi_n$ .

Assume  $\pi_n = \langle a_1, \dots, a_i \rangle$ . Then,  $\omega = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{i-1} \xrightarrow{a_i} \gamma_i \xrightarrow{a_{i+1}} \gamma_{i+1} \dots \gamma_{j-1} \xrightarrow{a_j} \gamma_j \xrightarrow{a_{j+1}} \gamma_{j+1} \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k$ , where:

- the prefix of  $\omega$ ,  $\omega' = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{i-1} \xrightarrow{a_i} \gamma_i$ , is a partial execution of  $M_{n-1}$  and either  $\gamma_i \models \varphi_n$  or there exists a configuration  $\gamma$  directly reachable from  $\gamma_i$ , and  $\gamma \models \varphi_n$ ;
- $\gamma_i \xrightarrow{a_{i+1}} \gamma_{i+1} \dots \gamma_{j-1} \xrightarrow{a_j} \gamma_j$  is an execution of  $M_n^a$ ;
- $a_{j+1}, \dots, a_k$  are activities in  $M_{n-1}$  and  $a_{j+1} = l(to_n)$ .

$\Delta_n$  is applicable to  $M_{n-1}$  on  $\omega'$ . From Definition 10, any complete execution which results from applying  $\Delta_n$  to  $M_{n-1}$  on  $\omega'$  satisfies  $G$ . Since  $\omega$  is such an execution,  $\omega$  satisfies  $G$ . (2)

From (1) and (2), we have that every execution of  $M_n$  satisfies  $G$ . Therefore,  $M_n$  satisfies  $G$ .  $\square$

#### 4.2.4 Multiple Corrections at Once vs. One at a Time

The corrective evolution problem involves applying a sequence of  $n$  corrections to a process model. By formulating the problem as the application of  $n$  corrections, rather than only one correction, we are addressing a more general problem.

If all corrections are strict, there is no difference between solving the corrective evolution problem and applying one correction at a time. However, if at least one of the corrections is relaxed or relaxed with conditions, the corrective evolution problem becomes more general, in the sense that it can be the case that more solutions are found when solving the corrective evolution problem than when applying one correction at a time.

The main reason is that we check that the evolved process model satisfies the goal only after applying all  $n$  corrections. If applying one correction at a time, we would have to ensure that each intermediary process model satisfies the goal. Moreover, by applying  $n$  corrections, the set of traces on which a relaxed correction *should* be applied changes depending on the other corrections in the sequence. Similarly, the set of traces on which a relaxed correction with conditions *can* be applied depends on the other corrections.

To clarify why the problem of applying  $n$  corrections is more general, we present a high-level example in which more solutions are found when solving the corrective evolution problem. The reason for finding more solutions will be that when solving a corrective evolution problem, the traces on which a relaxed correction with conditions can be applied depend on other corrections.

**Example** We consider the corrective evolution problem defined by a process model  $M_0$ , an empty goal, and two corrections, a relaxed correction with conditions  $C_1 = \langle \textit{with-conditions}, \pi_1, \varphi_1, \Delta_1 \rangle$ , and a relaxed correction  $C_2 = \langle \textit{relaxed}, \pi_2, \varphi_2, \Delta_2 \rangle$ . The plug-in point for  $C_1$  consists of only one partial trace of  $M_0$ ,  $\pi_1$ .  $\varphi_1$  is a plug-in condition for applying  $\Delta_1$  on  $\pi_1$ , and we obtain the process model  $M_1$ .  $C_2$  is such that the adaptation  $\Delta_2 = \textit{adapt}(M_2^a, \textit{from}_2, \textit{to}_2)$  performs a backward jump, with the node  $\textit{to}_2$  appearing in  $M_0$  before the plug-in point for  $C_1$ . By applying  $C_2$  to  $M_1$ , we introduce a loop in the process model, and there will be an infinity of traces leading to the plug-in point of  $C_2$ .

If the corrections are applied one at a time, then we will have one solution, the process model  $M_2$  obtained by applying  $C_2$  to  $M_1$ . If corrections are applied as a corrective evolution problem,  $M_2$  is one solution. However, since  $C_2$  introduces traces which lead to the plug-in point of  $C_1$ ,  $C_1$  can

be applied also to these traces. Therefore, there will be at least one other process model  $M'_2$  which is a solution to the problem, and in which  $C_1$  has been applied also on the traces introduced by  $C_2$  with the loop.

### 4.3 Discussion

In this Chapter, we have defined the corrective evolution problem as the application of a sequence of corrections to a process model. Solving a corrective evolution problem automatically poses several challenges. The challenges depend on whether all corrections are strict, they are all either strict or relaxed, or there is at least one correction which is relaxed with conditions. Since each type of problem introduces new challenges with respect to the previous (less general) type, we consider each type of problem separately.

If all corrections are strict, a solution to the problem can be constructed naively, by unfolding the original process model up to the plug-in point, adding the adaptation, and duplicating the fragment in the original process model starting from the node  $to$  in the adaptation and until the end node. This naive method has been used in the proof of Lemma 2. However, such a naively constructed solution will have many duplicated nodes, which are created both when unfolding the process model and when duplicating the fragment. Therefore, the challenge here is to automatically find a solution which contains as few duplicated nodes as possible.

If the corrections are all either strict or relaxed, finding a solution to the problem is no longer trivial. While constructing a corrected process model is relatively simple (as discussed in the proof of Lemma 2), this process model does not necessarily satisfy the goal of the original process model. Therefore, this case adds a new challenge, which is to automatically verify that every execution of the corrected process model satisfies the goal.

Finally, if any correction in the sequence of corrections given as input is relaxed with conditions, not only finding a solution is not trivial, but also constructing a corrected process model becomes not trivial. The reason is that the corrective evolution problem becomes a search problem. For each relaxed correction with conditions, we must search for the partial traces on which the corresponding adaptation can be applied, such that, by applying the other corrections, we can obtain a process model which satisfies the goal. Testing each partial trace individually is not an option, since there may be an infinite set of such partial traces.

Another issue which makes designing a search strategy difficult is the fact that corrections themselves introduce traces, and therefore the set of traces on which a correction which is relaxed or relaxed with conditions can be applied is not fixed, and depends on the other corrections. In fact, the problem gets significantly more complex if there is more than one relaxed correction with conditions, due to the combinatorial explosion. Therefore, a first new challenge in this third case is to understand how to group the partial traces, such that the number of tests to be performed is finite. The second challenge is to design search techniques which can deal efficiently with the combinatorial explosion due to multiple relaxed corrections with conditions.



## Chapter 5

# Encoding into State Transition Systems

Solving a corrective evolution problem involves constructing a synthesis of a process model with a sequence of adaptations, such that the synthesis complies with a set of domain object specifications and satisfies a goal. This synthesis is further constrained by the restrictions associated with each adaptation, which are specified as a plug-in point and a condition.

In order to generate this synthesis automatically, we encode each element of a corrective evolution problem as a *labeled state transition system* (STS). Each STS will be a compact representation of the behaviors of the corresponding element. We combine these STSs into a parallel STS which represents all the possible behaviors of the component STSs. This parallel STS can then be used to obtain the required synthesis. In particular, we may need to first restrict the behaviors of the parallel STS in order to comply to the given specifications. The result will be also an STS, which can afterwards be translated back to a process model. As we will show in Chapter 6 and Chapter 7, different techniques should be applied to the parallel STS to obtain the synthesis, depending on how restrictive the corrective evolution problem is.

In the following, we start by introducing some basic notions relative to

STSs, followed by the encoding of each element in the corrective evolution problem: process models, adaptations, partial traces, domain objects, conditions, and goals.

## 5.1 Basic STS Notions

An STS contains a set of states, some of which are marked as initial and/or accepting. Each state is labeled with a set of properties that hold in that state. The STS can move to new states as a result of performing actions. Actions are either *input* (controllable) or *output* (not controllable). The conditions under which an action can be performed and the effects of performing the action are defined by the transition relation.

### Definition 18 (STS)

Let  $\mathcal{P}$  be a set of propositions and  $Bool(\mathcal{P})$  the set of boolean expressions over  $\mathcal{P}$ . A state transition system  $\Sigma$  is a tuple  $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{S}^F, \mathcal{F} \rangle$ , where

- $\mathcal{S}$  is the set of states,
- $\mathcal{S}^0 \subseteq \mathcal{S}$  is the set of initial states,
- $\mathcal{I}$  and  $\mathcal{O}$  are the input and respectively output actions,
- $\mathcal{R} \subseteq \mathcal{S} \times Bool(\mathcal{P}) \times (\mathcal{I} \cup \mathcal{O}) \times \mathcal{S}$  is the transition relation,
- $\mathcal{S}^F \subseteq \mathcal{S}$  is the set of accepting states,
- $\mathcal{F} : \mathcal{S} \rightarrow 2^{\mathcal{P}}$  is the labeling function.

The labeling function  $\mathcal{F}$  determines if a boolean expression  $b \in Bool(\mathcal{P})$  holds in a particular state  $s$ . We write  $s, \mathcal{F} \models b$  to denote that boolean expression  $b$  is satisfied at state  $s$  given  $\mathcal{F}$ . Satisfiability of a formula is determined according to the standard inductive rules:

- $s, \mathcal{F} \models \top$ ;
- $s, \mathcal{F} \models p$ , iff  $p \in \mathcal{F}(s)$ ;
- $s, \mathcal{F} \models \neg b$ , iff  $s, \mathcal{F} \not\models b$ ;
- $s, \mathcal{F} \models b_1 \vee b_2$ , iff  $s, \mathcal{F} \models b_1$  or  $s, \mathcal{F} \models b_2$ .

The transitions in the STS are guarded: a transition  $(s, b, a, s')$  is possible in state  $s$  only if the guard expression  $b$  is satisfied in that state, i.e., if  $s, \mathcal{F} \models b$ . We say that an action  $a \in \mathcal{I} \cup \mathcal{O}$  is applicable in a state  $s \in \mathcal{S}$  if there exists a state  $s' \in \mathcal{S}$  such that  $(s, b, a, s') \in \mathcal{R}$  and  $s, \mathcal{F} \models b$ . A state  $s$  is final if no action is applicable in  $s$ , i.e., there is no transition leaving  $s$ .

The behavior of an STS is represented by its set of possible runs. A *run* is an alternating sequence of states and actions  $s_0, a_0, s_1, a_1, \dots$  such that  $s_0 \in \mathcal{S}^0$  and  $(s_i, b_i, a_i, s_{i+1}) \in \mathcal{R}$ . In general, runs may be finite or infinite. A run is said to be complete if it is finite and its last state is accepting.

If  $\sigma = s_0, a_0, s_1, a_1, \dots$  is a run of the STS, the sequence of actions  $a_0, a_1, \dots$  is called a *trace*. A trace is complete if it corresponds to a complete run. The projection of a run  $\sigma = s_0, a_0, s_1, a_1, \dots$  on a set of actions  $A \subseteq \mathcal{I} \cup \mathcal{O}$  is an ordered sequence  $a'_0, \dots, a'_m$ , representing the actions in  $\sigma$  which are also in  $A$ . We denote the projection of  $\sigma$  on  $A$  with  $\prod_A(\sigma)$ .

The parallel product of two STSs specifies that the two STSs move concurrently on common actions, and independently if there are no common actions.

**Definition 19 (Parallel Product)**

Let  $\Sigma_1 = \langle \mathcal{S}_1, \mathcal{S}_1^0, \mathcal{I}_1, \mathcal{O}_1, \mathcal{R}_1, \mathcal{S}_1^F, \mathcal{F}_1 \rangle$  and  $\Sigma_2 = \langle \mathcal{S}_2, \mathcal{S}_2^0, \mathcal{I}_2, \mathcal{O}_2, \mathcal{R}_2, \mathcal{S}_2^F, \mathcal{F}_2 \rangle$  be two STSs. The parallel product  $\Sigma_1 \parallel \Sigma_2$  is defined as

$$\langle \mathcal{S}_1 \times \mathcal{S}_2, \mathcal{S}_1^0 \times \mathcal{S}_2^0, \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{R}_1 \parallel \mathcal{R}_2, \mathcal{S}_1^F \times \mathcal{S}_2^F, \mathcal{F}_1 \parallel \mathcal{F}_2 \rangle$$

where  $(\mathcal{F}_1 \parallel \mathcal{F}_2)(s_1, s_2) = \mathcal{F}_1(s_1) \cup \mathcal{F}_2(s_2)$  and

- $\langle (s_1, s_2), b_1 \wedge b_2, a, (s'_1, s'_2) \rangle \in (\mathcal{R}_1 \parallel \mathcal{R}_2)$  if  $\langle s_1, b_1, a, s'_1 \rangle \in \mathcal{R}_1$  and  $\langle s_2, b_2, a, s'_2 \rangle \in \mathcal{R}_2$ ;
- $\langle (s_1, s_2), b_1, a_1, (s'_1, s'_2) \rangle \in (\mathcal{R}_1 \parallel \mathcal{R}_2)$  if  $\langle s_1, b_1, a_1, s'_1 \rangle \in \mathcal{R}_1$  and  $\forall \langle s_2, b_2, a_2, s'_2 \rangle \in \mathcal{R}_2, a_1 \neq a_2$ ;
- $\langle (s_1, s_2), b_2, a_2, (s_1, s'_2) \rangle \in (\mathcal{R}_1 \parallel \mathcal{R}_2)$  if  $\langle s_2, b_2, a_2, s'_2 \rangle \in \mathcal{R}_2$  and  $\forall \langle s_1, b_1, a_1, s'_1 \rangle \in \mathcal{R}_1, a_1 \neq a_2$ .

## 5.2 Encoding the Process Models

To transform a process model into an STS, we recursively translate its basic constructs. We encode the constructs using input and output actions, depending on whether the inclusion of an element can be controlled while performing the synthesis of the process model with the sequence of adaptations. Input actions correspond to elements that can be controlled in the synthesis, such as activity nodes and control connectors. Output actions correspond to elements that cannot be controlled by the synthesis, such as the branching conditions which follow an XorSplit. The idea is that once the XorSplit is included in the synthesis, all its branches must be included. In this sense, it is not under the control of the synthesis whether to include the branch or not.

Note that this encoding as input/output actions is different from the encoding proposed in service composition approaches such as [10, 76]. With service composition, the purpose is not to integrate the services, but to orchestrate them using an external controller. In that setting, actions are considered to be input (output) if they are controllable (non-controllable) by the external orchestrator.

Table 5.1 contains the process model elements and their recursive translation to STS. We denote with  $\alpha$  the generic process model element. Fur-

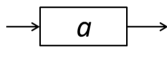
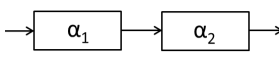
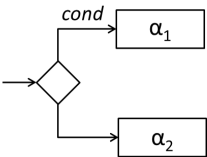
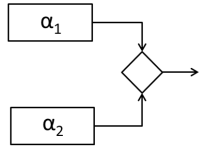
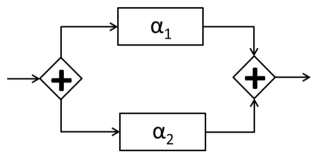
Process model element		STS transitions
activity node $n \in N_A, l(n) = \langle a, pre, eff \rangle$		$(s_b, pre, ?a, s_e)$
sequence		$s_b \xrightarrow{\alpha_1} s', s' \xrightarrow{\alpha_2} s_e$
XorSplit		$(s_b, \top, ?xor, s_0)$ $(s_0, cond, !case, s_1)$ $(s_0, \neg cond, !otherwise, s_2)$ $s_1 \xrightarrow{\alpha_1} s_e, s_2 \xrightarrow{\alpha_2} s'_e$
XorJoin		$s_b \xrightarrow{\alpha_1} s_e, s'_b \xrightarrow{\alpha_2} s_e$
And-block		$(s_b, \top, ?and, s_0)$ $(s_0, \top, !order_1, s_1)$ $(s_0, \top, !order_2, s_2)$ $s_1 \xrightarrow{\alpha_1} s'_1, s'_1 \xrightarrow{\alpha_2} s_e$ $s_2 \xrightarrow{\alpha_2} s'_2, s'_2 \xrightarrow{\alpha_1} s_e$

Table 5.1: Encoding process model elements as STSs

ther, we denote the start state of the element with  $s_b$ , and the end states with  $s_e$  and  $s'_e$ . With  $s_b \xrightarrow{\alpha} s_e$  we denote the recursive translation of  $\alpha$ . To differentiate between input and output actions, we prepend the names with  $?$ , respectively  $!$ . Note that for the And-block, Table 5.1 shows the transformation rules for the case when each branch contains only one activity node; a generic And-block will result in an STS allowing every possible interleaving combination of the activities in its branches.

Given a process model  $M = \langle N, E, l, t, c \rangle$ , we first perform a pre-

processing step on  $M$ . In this step, XorSplit nodes which are connected directly are merged into a single XorSplit node with multiple branches. This is just to ensure that there is at most one condition between any two consecutive activity nodes in  $M$ . We then construct an STS  $\Sigma_M = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{S}^F, \mathcal{F} \rangle$  by applying the translation rules in Table 5.1.

Every activity and control connector will have a corresponding input action in  $\mathcal{I}$ . Further, each node in  $M$ , whether activity node or control connector, will have at least one corresponding transition in  $\Sigma_M$ . Output actions do not have corresponding activities or nodes, see for example the encoding of the branching conditions.

Activity preconditions are copied as transition guards in  $\Sigma_M$ . In contrast to preconditions, the effects of activities are not captured in  $\Sigma_M$ . The information encoded in the effects will be used later on, when transforming the domain objects to STSs.

We label each state in  $\Sigma_M$  with a new proposition corresponding to this state, i.e.,  $\forall s_i \in \mathcal{S}, \mathcal{F}(s_i) = \{s_i(M)\}$ . However, we do not label the states with propositions referring to domain object states. The reason is that  $\Sigma_M$  is not supposed to be run in isolation. In fact, due to the guarded transitions, this STS will not move if it is run in isolation. However, the STSs corresponding to the domain objects will have appropriately labeled states. This way, the guarded transitions in  $\Sigma_M$  will become enabled in the parallel product of  $\Sigma_M$  with the domain objects STSs.

Table 5.1 shows also the encoding of an XorSplit node connected to other nodes through control edges annotated with conditions. In the special case when the condition holds only after one or more uncontrollable events are triggered in some domain objects, this encoding is not sufficient. The reason is that the triggering of a such an uncontrollable event must be simulated in the STS encoding. Therefore, for each such condition, we will create a separate STS in Section 5.5. The label  $s_b(M)$  which corresponds

to the start state of the encoding  $s_b$  is sufficient for marking the point where the condition must be triggered.

Finally, we mark all initial and final states of  $\Sigma_M$  as accepting to encode the fact that, once started, the process must be completed.

### 5.3 Encoding the Adaptations

For each adaptation  $\Delta_i = \text{adapt}(M_i^a, \text{from}_i, \text{to}_i)$ , we define an STS  $\Sigma_{\Delta_i}$  as follows. We first translate the process model included in the adaptation, i.e.,  $\Sigma_{\Delta_i} = \Sigma_{M_i^a}$ . To use the adaptation multiple times, we introduce an input action  $?resume_i$ , and add a transition on  $?resume_i$  from each final state to the initial state.

We recall that  $\text{from}_i$  can be any node from the original process model  $M_0$  or the model of a previous adaptation  $\Delta_j$ ,  $1 \leq j < i$ . Further,  $\text{to}_i$  is a node from  $M_0$  (see Definition 17). Therefore, each adaptation realizes a jump in  $M_0$ . If applied on a previous adaptation  $\Delta_j$ ,  $j < i$ , it also realizes a reset jump in  $\Delta_j$ . This reset jump is necessary in order to be able to reuse the adaptation  $\Delta_j$ .

To encode these jumps, we use the  $?resume_i$  action. Let  $\langle s_{\text{from}}, b_{\text{from}}, a_{\text{from}}, s'_{\text{from}} \rangle$  be the transitions corresponding to  $\text{from}_i$ . Since  $\text{to}_i$  cannot be part of an And-block, there will be only one transition  $\langle s_{\text{to}}, b_{\text{to}}, a_{\text{to}}, s'_{\text{to}} \rangle$  corresponding to  $\text{to}_i$ . We can then be in one of the following two situations:

- the transitions are all in  $\Sigma_{M_0}$ . We then add to  $\Sigma_{M_0}$  the action  $?resume_i$  and transitions on  $?resume_i$  from every  $s'_{\text{from}}$  to  $s_{\text{to}}$ .
- $\langle s_{\text{from}}, b_{\text{from}}, a_{\text{from}}, s'_{\text{from}} \rangle$  are transitions in  $\Sigma_{\Delta_j}$ , and  $\langle s_{\text{to}}, b_{\text{to}}, a_{\text{to}}, s'_{\text{to}} \rangle$  is a transition in  $\Sigma_{M_0}$ . We add  $?resume_i$  to both  $\Sigma_{\Delta_j}$  and  $\Sigma_{M_0}$ . In  $\Sigma_{\Delta_j}$ , we add transitions on  $?resume_i$  from all  $s'_{\text{from}}$  to the initial state. In

$\Sigma_{M_0}$ , the start of the jump is the same as for the jump corresponding to  $\Delta_j$ . For every state  $s$  such that there is a transition from  $s$  on  $?resume_j$ , we add a transition on  $?resume_i$  to  $s_{to}$ .

The start of the jump are the states  $s'_{from}$  in  $\Sigma_{M_0}$ , or, if there is a reset jump, in  $\Sigma_{\Delta_j}$ . We label these states with a new proposition  $point_i$ , to mark the point where  $\Delta_i$  must be applied.

The adaptation can be used only if  $\varphi_i$  holds, at a particular execution point, and, if the correction is strict, only for  $\pi_i$ . We therefore add  $\varphi_i \wedge point_i \wedge trace_i$  to the guard of every initial transition in  $\Sigma_{\Delta_i}$ . Here,  $trace_i$  is a proposition which will be used in the STS of trace  $\pi_i$  to label the state corresponding to the completion of the activities in  $\pi_i$  (if the correction is relaxed, since  $\pi_i$  is ignored, this will be the initial state).

### The Semaphore STS

Adaptations may be executed only partially, if there is re-adaptation. However, if an adaptation is started, one adaptation (not necessarily the same) has to finish for the control to be given back to the main process. Moreover, if there is re-adaptation, the control is never given back to the failed adaptation. To encode these properties, we use a semaphore STS,  $\Sigma_{semaphore}$ .

$\Sigma_{semaphore}$  has an initial, accepting state  $s_0$  corresponding to  $M_0$ , and a state  $s_i$  for every adaptation  $\Delta_i$ ,  $1 \leq i \leq n$ . If  $\Sigma_{\Delta_i}$  moves from its initial state,  $\Sigma_{semaphore}$  will move from any state  $s_j$ ,  $0 \leq j < i$ , to state  $s_i$ . In other words, for every initial transition  $(s, b, a, s')$  in  $\Sigma_{\Delta_i}$ , we add to  $\Sigma_{semaphore}$  a transition from  $s_j$  to  $s_i$  on  $a$  guarded by  $point_i \wedge trace_i$ . From state  $s_i$  it will move back to the initial state  $s_0$  on  $?resume_i$ , to encode that the control is given back to  $M_0$  if  $\Delta_i$  is completed. This transition is guarded, such that it becomes enabled only when  $\Delta_i$  is also in a state from which it can execute  $?resume_i$ .



Each state  $s_i$  in  $\Sigma_{semaphore}$  is labeled with a proposition  $flag_i$ ; these flags are added as guards to transitions in  $\Sigma_{M_0}, \Sigma_{\Delta_1}, \dots, \Sigma_{\Delta_n}$  which are not on *?resume* actions. In particular, in  $\Sigma_{M_0}$  we add  $flag_0$  to the guard of every transition. In  $\Sigma_{\Delta_i}$  we add  $flag_0 \vee \dots \vee flag_{i-1}$  to the guard of transitions from initial states, and  $flag_i$  to the guard of every other transition. This way, if  $\Delta_i$  fails and another adaptation  $\Delta_k, k > i$ , is started,  $\Sigma_{semaphore}$  moves to state  $s_k$  and the control is never given back to  $\Delta_i$ .

**Example** Figure 5.1 shows the encoding of the process model, adaptations, and semaphore in our scenario. In particular, Figure 5.1(a) is the encoding of the car process model introduced in Figure 3.5, while the Figures 5.1(b) and 5.1(c) show the encoding of the adaptations in Figure 3.6.

## 5.4 Encoding the Partial Traces

We construct an STS  $\Sigma_{\pi_i}$  for each trace  $\pi_i = \langle a_1, \dots, a_k \rangle$ . If the correction is relaxed,  $\pi_i$  is ignored and  $\Sigma_{\pi_i}$  will contain only an initial state  $s_0$ .

If the correction is strict,  $\Sigma_{\pi_i}$  needs to capture the situation when the trace  $\pi_i$  has been followed, that is, the activities in  $\pi_i$  have been completed in the order specified by  $\pi_i$ . For every  $j, 1 \leq j \leq k$ ,  $a_j$  is an activity in the original process model  $M_0$  and/or in the adaptation models  $M_1^a, \dots, M_{i-1}^a$ . The idea is that  $\Sigma_{\pi_i}$  should move from state  $s_{j-1}$  to  $s_j$  on any action corresponding to  $a_j$ .

Let  $?a_j$  be the input actions corresponding to  $a_j$ . Suppose that  $a_j$  appears in the process model  $M_l$ , and that  $?a_j$  can be executed in  $\Sigma_{M_l}$  from one or more states  $s$ . We therefore add to  $\Sigma_{\pi_i}$  a new state  $s_j$ , the actions  $?a_j$ , and for every  $?a_j$  and state  $s$  a transition  $\langle s_{j-1}, s(M_l) \wedge (flag_0 \vee \dots \vee flag_{l-1}), ?a_j, s_j \rangle$  if  $s$  is an initial state in  $M_l$  and  $l \geq 1$ , and  $\langle s_{j-1}, s(M_l) \wedge flag_l, ?a_j, s_j \rangle$  otherwise. This way, in the parallel prod-

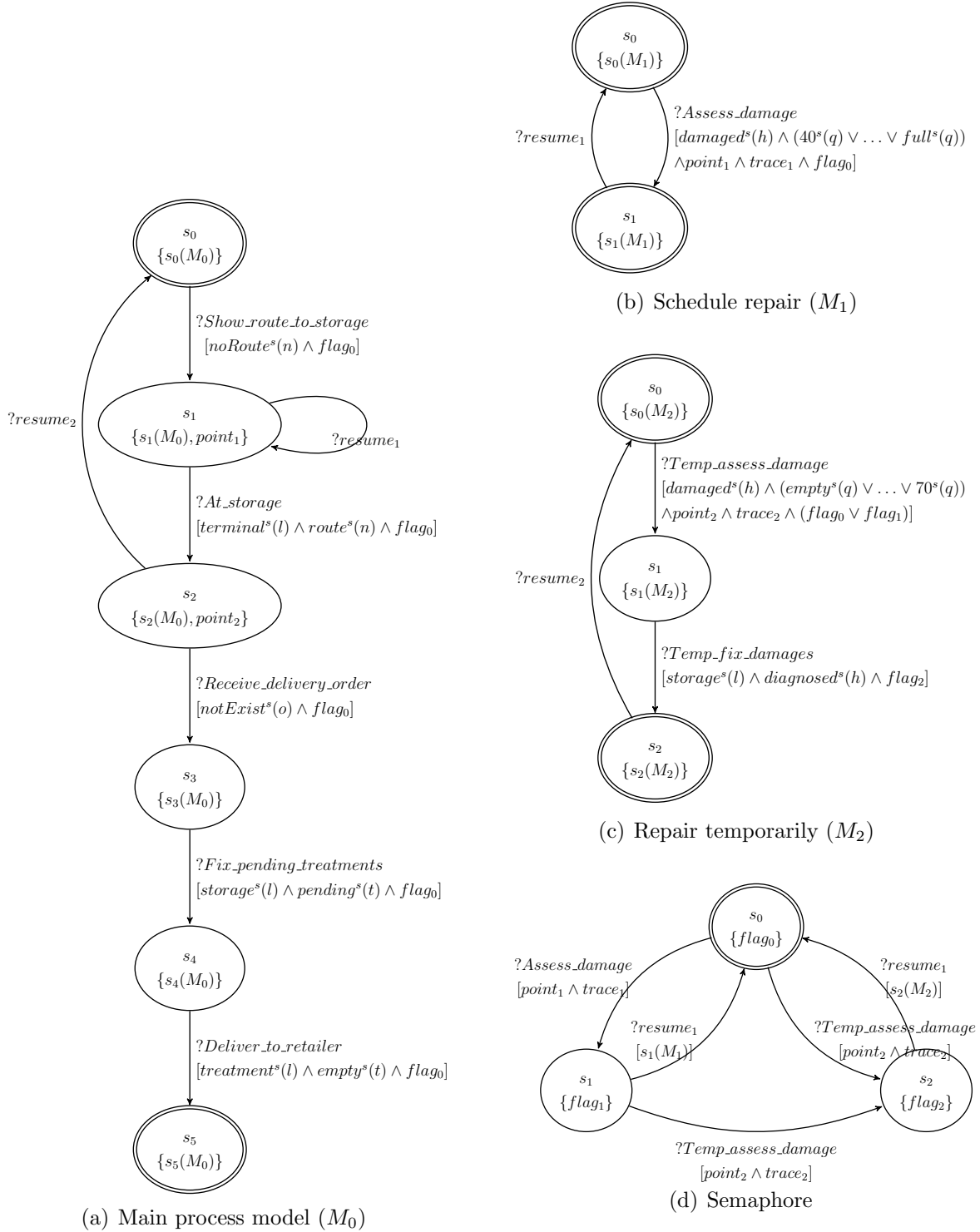


Figure 5.1: STS encoding of the process model and adaptations

uct of  $\Sigma_{\pi_i}$  with  $\Sigma_{M_0}, \Sigma_{\Delta_1}, \dots, \Sigma_{\Delta_n}$ , the transitions on  $?a_j$  will be triggered only when  $\Sigma_{M_l}$  is ready to execute  $?a_j$ . We mark the end of the trace  $\pi_i$  by labeling the last added state  $s_k$  with a new proposition  $trace_i$ , i.e.,  $\mathcal{F}(s_k) = \{trace_i\}$ .

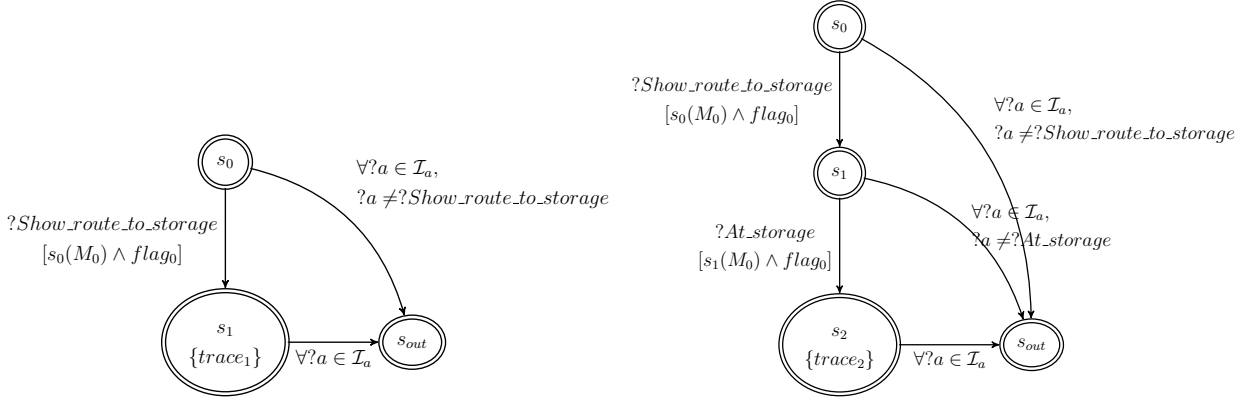
Further, we add to  $\Sigma_{\pi_i}$  a state  $s_{out}$ , which signals that the trace  $\pi_i$  has not been followed, that at some point we executed an activity not on  $\pi_i$ . We then add transitions to this state.  $\Sigma_{\pi_i}$  should move from any state  $s_{j-1}$ ,  $1 \leq j \leq k$ , to  $s_{out}$  on any action corresponding to an activity different from  $a_j$ , and from  $s_k$  to  $s_{out}$  on any action corresponding to an activity.

Let  $\mathcal{I}_a$  be the set of input actions from  $\Sigma_{M_0}, \Sigma_{\Delta_1}, \dots, \Sigma_{\Delta_n}$  which correspond to activities. In other words,  $\mathcal{I}_a$  does not include the actions  $?resume_1, \dots, ?resume_n$ , or the actions corresponding to control connectors. We add  $\mathcal{I}_a$  to the actions of  $\Sigma_{\pi_i}$ . For every  $j, 0 \leq j \leq k$ , we add to  $\Sigma_{\pi_i}$  a transition  $(s_j, b, ?a, s_{out})$  for every transition  $(s, b, ?a, s')$  in  $\Sigma_{M_0}, \Sigma_{\Delta_1}, \dots, \Sigma_{\Delta_n}$  which is such that  $?a \in \mathcal{I}_a$  and, if  $j < k$ , such that  $?a \neq ?a_{j+1}$ .

**Example** As discussed in Section 4.2.2, the two partial traces in our scenario are  $\pi_1 = \langle Show\ route\ to\ storage \rangle$  and  $\pi_2 = \langle Show\ route\ to\ storage, Assess\ damage, At\ storage \rangle$ . The encoding of these two partial traces is shown in Figure 5.2.

## 5.5 Encoding the Conditions

We now consider all the conditions that appear in the corrective evolution problem. This includes first the conditions  $\varphi_1, \dots, \varphi_n$  which appear in the corrections. However, it also includes the conditions which appear in the original process model  $M_0$  or in the adaptation process models  $M_1^a, \dots, M_n^a$ . With this second type of conditions, we refer to the annotations to control edges connecting XorSplit nodes to other nodes.


 Figure 5.2: STS encoding of traces: (a)  $\pi_1$ ; (b)  $\pi_2$ 

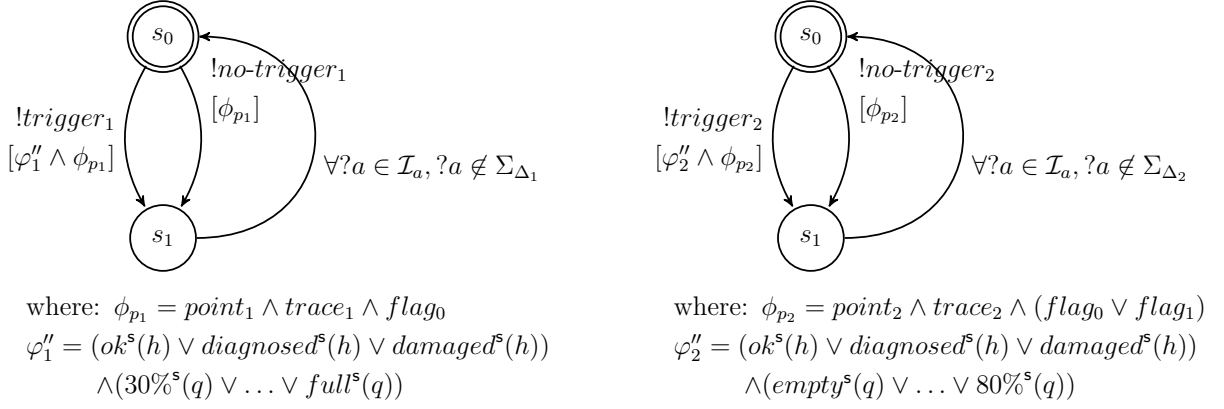
For some of these conditions, we construct STSs which are responsible for triggering any uncontrollable events which are necessary for satisfying the conditions. The main difficulty here is that we should not consider all possible situations, but only the situations that either appeared in the process models  $M_0, M_1^a, \dots, M_n^a$ , or are described by the conditions  $\varphi_1, \dots, \varphi_n$ .

Let  $\varphi_i$  be the formula corresponding to such a condition. We perform two transformations on  $\varphi_i$ . First, we replace in  $\varphi_i$  every negative literal  $\neg s^s(o)$  with the equivalent disjunction  $\bigvee_{s_j \in L, s_j \neq s} s_j^s(o)$ . We then check if the updated condition  $\varphi'_i$  contains any literal  $s_u^s(o)$  such that in the domain object  $o = \langle L, L^0, \mathcal{E}, T \rangle$  the state  $s_u$  can be reached through one or more uncontrollable events, i.e.,  $\exists e \in \mathcal{E}_U, (s, e, s_u) \in T$ .

If such a literal exists, we construct an STS  $\Sigma_{\varphi_i}$ , which triggers any uncontrollable events necessary for satisfying the condition. Since  $\varphi_i$  is an arbitrary formula from  $Bool(P_S)$ , more than one uncontrollable event might need to be triggered at the same time for the condition to be satisfied.

We first generate a new condition  $\varphi''_i$  by replacing in  $\varphi'_i$  all uncontrollable literals  $s_u^s(o)$  with the formula that identifies the states before and after the uncontrollable events, i.e.,  $s_u^s(o) \vee \bigvee_{s \in L, (s, e, s_u) \in T, e \in \mathcal{E}_U} s^s(o)$ .

We add to  $\Sigma_{\varphi_i}$  two output actions:  $!trigger_i$  and  $!no-trigger_i$ , and two transitions:  $(s_0, \varphi''_i \wedge \phi_{p_i}, !trigger_i, s_1)$ , respectively  $(s_0, \phi_{p_i}, !no-trigger_i, s_1)$ .

Figure 5.3: STS encoding of conditions: (a)  $\varphi_1$ ; (b)  $\varphi_2$ 

The two actions simulate the uncontrollability of the events, in that both cases (when  $\varphi_i$  holds, as well as when it does not hold) will be considered. The expression  $\phi_{p_i}$  included in the guards will have a different value, depending on whether  $\varphi_i$  is a condition in a correction or a condition appearing in a process model:

- if  $\varphi_i$  is the condition for the correction at step  $j$ , then  $\phi_{p_i} = point_j \wedge trace_j \wedge (flag_0 \vee \dots \vee flag_{j-1})$ . The reason is that  $\varphi_i$  should only be triggered at the same point where  $\Delta_j$  should be plugged in.
- if  $\varphi_i$  is a branch condition appearing in a process model  $M_j$ , then  $\phi_{p_i} = s_b(M_j) \wedge flag_j$ , where  $s_b(M_j)$  is the label associated to the start state in the encoding of the corresponding XorSplit node. Also in this case,  $\varphi_i$  should only be triggered before the corresponding branch condition is evaluated.

Since  $\varphi_i$  can be triggered repeatedly, we add reset transitions  $(s_1, b, ?a, s_0)$  for every transition  $(s, b, ?a, s')$  on an input action  $?a \in \mathcal{I}_a, ?a \notin \Sigma_{\Delta_j}$  (respectively every  $?a \in \mathcal{I}_a, ?a \notin \Sigma_{M_0}$ , if  $\varphi_i$  is a condition in  $M_0$ ).

**Example** Figure 5.3 shows the encoding of the two conditions in our scenario. The conditions appear in the two corrections discussed in Section

4.2.2. For both conditions, all the literals included in the condition are such that the corresponding state is reachable through an uncontrollable event. This is the *damage* event in the *Car Health* ( $h$ ) domain object, respectively the *enqueue/dequeue* events in the *Service Station Queue* ( $q$ ).

Note that for condition  $\varphi_1 = \text{damaged}^s(h) \wedge (40\%{}^s(q) \vee \dots \vee \text{full}^s(q))$ , the formula which identifies the states before and after the uncontrollable events is therefore  $\varphi_1'' = (\text{ok}^s(h) \vee \text{diagnosed}^s(h) \vee \text{damaged}^s(h)) \wedge (30\%{}^s(q) \vee \dots \vee \text{full}^s(q))$ . The trigger actions are added also to the domain objects STSs, as will be described in the next Section. This way, if either  $h$  or  $q$  is not already in a state satisfying  $\varphi_1$ , but from which such a state can be reached ( $h$  is in state *ok* or *diagnosed*, or  $q$  is in state 30%), by executing the trigger action, they will move simultaneously to a state satisfying  $\varphi_1$ .

## 5.6 Encoding the Domain Objects

For each domain object  $o = \langle L, L^0, \mathcal{E}, T \rangle$ , we define an STS  $\Sigma_o = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{S}^F, \mathcal{F} \rangle$ , which has the same states ( $\mathcal{S} = L$ ) and initial states ( $\mathcal{S}^0 = L^0$ ), and for which all states are accepting ( $\mathcal{S}^F = \mathcal{S}$ ). We label states with the corresponding propositions (i.e.,  $\forall s \in \mathcal{S} : \mathcal{F}(s) = \{s^s(o)\}$ ).

To create the transitions in the new STS, we use the original process model  $M_0$  and the adaptation process models  $M_1^a, \dots, M_n^a$ . In particular, we add transitions to reflect how the state of the STS is affected by the execution of the actions corresponding to process model activities. We guard these transitions, such that they are triggered only when the corresponding actions in  $\Sigma_{M_0}, \Sigma_{\Delta_1}, \dots, \Sigma_{\Delta_n}$  are executed.

For each transition  $(s, e, s') \in T$ , we consider all the activities  $\langle a, \text{pre}, \text{eff} \rangle$  appearing in  $M_0$  and  $M_1^a, \dots, M_n^a$  which could trigger the event  $e$ , i.e., for which  $e^e(o) \in \text{eff}$ . Let  $?a$  be an input action corresponding to such an activity. Suppose that  $a$  appears in the process model  $M_i$ , and that  $?a$  can be

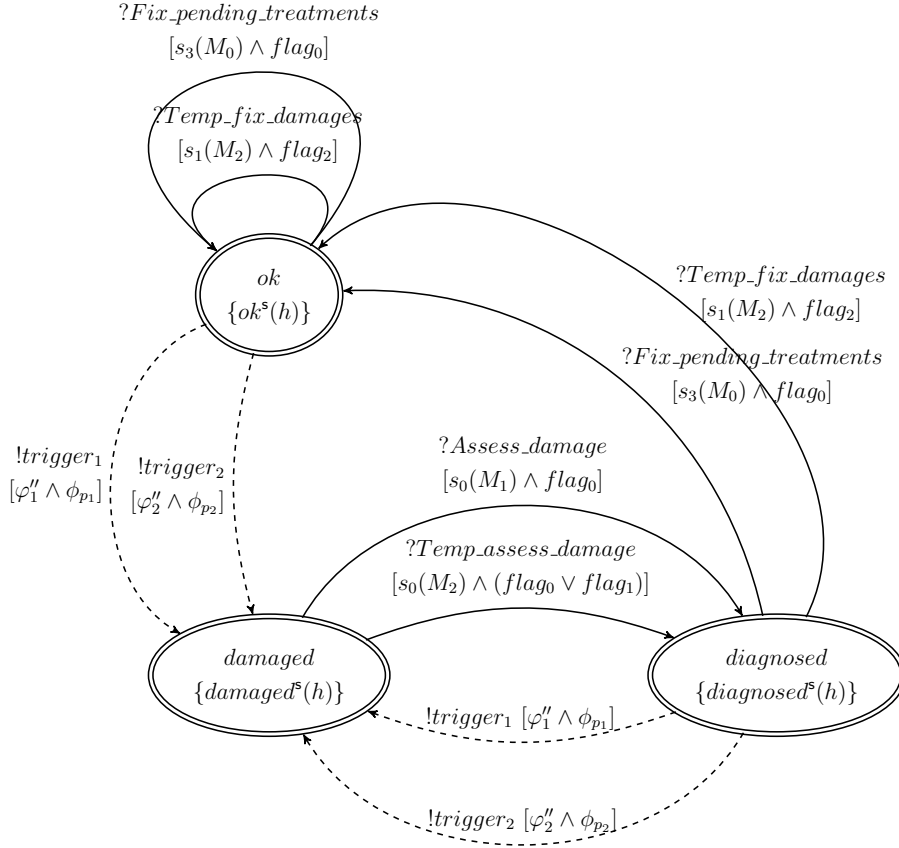
executed in  $\Sigma_{M_i}$  from one or more states  $s_j, j \geq 0$ . In this case, we add  $?a$  to  $\mathcal{I}$ . For each state  $s_j$ , we add to  $\mathcal{R}$  a transition  $(s, s_j(M_i) \wedge (flag_0 \vee \dots \vee flag_{i-1}), ?a, s')$  if  $s_j$  is an initial state in  $M_i$ , and  $(s, s_j(M_i) \wedge flag_i, ?a, s')$  otherwise. This way, in the parallel product of  $\Sigma_o$  and  $\Sigma_{M_i}$ , the transition on  $?a$  will be triggered only when  $\Sigma_{M_i}$  is ready to execute  $?a$ . This ensures that it is never the domain object STS that initiates the transitions, but rather that these transitions reflect the execution on actions corresponding to activities in the process model STSs.

If the domain object includes transitions on uncontrollable events, we add new transitions using the STSs of conditions created in Section 5.5. For each transition  $(s, e, s_u) \in T, e \in \mathcal{E}_U$ , we consider all the conditions containing the uncontrollable literal  $s_u^s(o)$ . Let  $\varphi_i$  be such a condition. We then add to  $\Sigma_o$  the action  $!trigger_i$  and the transition  $(s, \varphi_i'' \wedge \phi_{p_i}, !trigger_i, s_u)$ .

**Example** Figure 5.4 shows the encoding of the *Car Health (h)* domain object from Figure 3.4. Note that the transition on the event *diagnose* has been replaced with two transitions on the actions corresponding to the activities *Assess damage* and *Temp assess damage*. Both activities contained the event *diagnose* in their effects. Similarly, the transitions on the event *repair* have been replaced with two transitions on the actions corresponding to the activities *Fix pending treatments* and *Temp fix damages*. Finally, the transitions on the event *damage* have been replaced by transitions on the  $!trigger_1$  and  $!trigger_2$  actions.

## 5.7 Encoding the Goals

To represent the satisfaction of our goals, we follow the approach proposed by Bertoli et al. in [9, 10] for tracking the satisfaction of composition requirements when composing service descriptions. There, the idea is to



where:

$$\varphi_1'' = (ok^s(h) \vee diagnosed^s(h) \vee damaged^s(h)) \wedge (30\%^s(q) \vee \dots \vee full^s(q))$$

$$\phi_{p1} = point_1 \wedge trace_1 \wedge flag_0$$

and

$$\varphi_2'' = (ok^s(h) \vee diagnosed^s(h) \vee damaged^s(h)) \wedge (empty^s(q) \vee \dots \vee 80\%^s(q))$$

$$\phi_{p2} = point_2 \wedge trace_2 \wedge (flag_0 \vee flag_1)$$

Figure 5.4: STS encoding of the *Car Health* domain object

create a set of STSs for each composition requirement, and then define a propositional formula on the states of these STSs, which holds when the requirement is satisfied.

Following this approach, we construct STSs which correspond to the satisfaction of each goal statement  $\psi_0 \implies (\psi_1 \succ \dots \succ \psi_k)$  included in the goal. Since the formulas  $\psi_0, \dots, \psi_n$  contain only state propositions, they can be used directly as transition guards in our STS. For every formula  $\psi$ , we introduce an output action  $!a_\psi$  which is triggered when the formula is



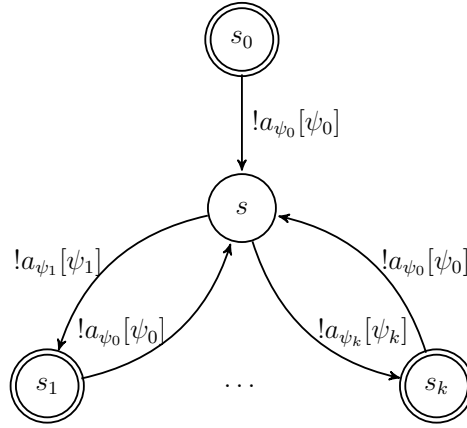


Figure 5.5: STS encoding of a goal statement

satisfied. Transitions on  $!a_{\psi}$  will always be guarded by  $\psi$ .

The STS for a goal statement  $\psi_0 \implies (\psi_1 \succ \dots \succ \psi_k)$  is shown in Figure 5.5. The STS is initially in an accepting state  $s_0$ . If the premise  $\psi_0$  is satisfied ( $!a_{\psi_0}$  is triggered), it moves to a non-accepting state and waits for any of the formulas  $\psi_1, \dots, \psi_k$  to be completed (one of the actions  $!a_{\psi_1}, \dots, !a_{\psi_k}$  is triggered). When one of  $!a_{\psi_1}, \dots, !a_{\psi_k}$  is triggered, the STS moves to the corresponding accepting state from  $s_1, \dots, s_k$ . From each of  $s_1, \dots, s_k$ , the STS moves back to the non-accepting state in case  $!a_{\psi_0}$  is triggered again.

To encode the preference order, for each goal statement we define a requirement  $\rho = (s_0, \dots, s_k)$ , where  $s_0$  is the initial state, and  $s_1, \dots, s_k$  are the states reached with transitions on  $!a_{\psi_1}, \dots, !a_{\psi_k}$ . These requirements will be used in Chapter 7 for constructing the planning goal.



# Chapter 6

## Strict Corrective Evolution

In this Chapter, we design an automated technique for solving a special case of corrective evolution problems, the case when all corrections are strict. First, we encode the process model, domain objects, partial traces, conditions, and adaptations into state transition systems (STSs). We then compute the parallel product of these STSs and obtain an STS which encodes all the executions of the corrected process model. We minimize this STS in order to remove redundant transitions. Finally, we use the correspondences created when encoding the inputs to translate the resulting STS to a new process model.

In the following, we first present an overview of our automated technique in Section 6.1. We then prove that the approach is correct and complete in Section 6.2. We conclude with a discussion on the advantages and limitations of the approach in Section 6.3.

### 6.1 Description of the Approach

A strict corrective evolution problem is defined by a process model  $M_0$  which satisfies a goal  $G$ , and a sequence of corrections  $C_1, \dots, C_n$  such that for all  $i$ ,  $1 \leq i \leq n$ ,  $C_i = \langle \textit{strict}, \pi_i, \varphi_i, \Delta_i \rangle$ .

An overview of the solution for strict corrective evolution is shown in

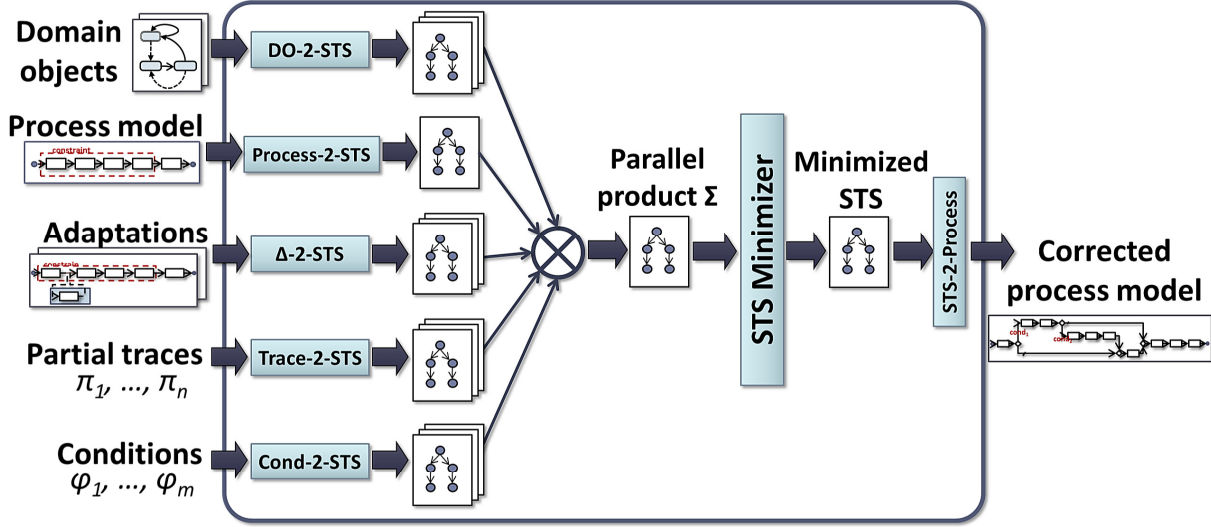


Figure 6.1: Strict corrective evolution: solution overview

Figure 6.1. We encode the process model into an STS  $\Sigma_{M_0}$ , adaptations into  $\Sigma_{\Delta_1}, \dots, \Sigma_{\Delta_n}, \Sigma_{semaphore}$ , partial traces into  $\Sigma_{\pi_1}, \dots, \Sigma_{\pi_n}$ , conditions into  $\Sigma_{\varphi_1}, \dots, \Sigma_{\varphi_m}$ , and domain objects into  $\Sigma_{o_1}, \dots, \Sigma_{o_p}$ , as described in the previous chapter. We then compute their parallel product:

$$\Sigma = \Sigma_{M_0} \parallel \Sigma_{\Delta_1} \parallel \dots \parallel \Sigma_{\Delta_n} \parallel \Sigma_{semaphore} \parallel \Sigma_{\pi_1} \parallel \dots \parallel \Sigma_{\pi_n} \parallel \Sigma_{\varphi_1} \parallel \dots \parallel \Sigma_{\varphi_m} \parallel \Sigma_{o_1} \parallel \dots \parallel \Sigma_{o_p}$$

We simplify  $\Sigma$  by removing the transitions for which the guard condition evaluates to false and which are therefore never enabled, i.e., the transitions  $(s, b, a, s')$  for which  $s, \mathcal{F} \neq b$ . After this simplification step, since for all remaining transitions the guard condition evaluates to true, we can remove from  $\Sigma$  the guards and the labeling function  $\mathcal{F}$ .

$\Sigma$  is nondeterministic, due to the fact that the domain objects STSs can have multiple initial states and nondeterministic transitions. Moreover, if at a certain point in the process multiple configurations of the domain objects are possible, and these configurations are treated in the same way by the process, then  $\Sigma$  will contain many similar transitions. For example, in

our scenario, at any point during the execution of the process model multiple configurations are possible, differing among each other in the value associated to the service station queue domain object. Each of these possible configurations will be treated separately in  $\Sigma$ , leading to an exponential blowup of the number of states.

To be able to transform  $\Sigma$  back to a process model, we must first convert it to a deterministic STS. For this, as well as to remove the redundant transitions, we minimize  $\Sigma$ . As criteria for STS equivalence we use completed trace equivalence, one of the weakest notions of behavioral equivalence [112]. Modulo completed trace equivalence, every STS is deterministic, that is, we can always find a deterministic STS which is completed trace equivalent to  $\Sigma$ . The minimal, deterministic STS  $\Sigma_{strict}$  which results can then be transformed back into a process model.

**Definition 20** ( $\Sigma_{strict}$ )

*Assume a corrective evolution problem defined by a process model  $M_0$ , a goal  $G$ , and a sequence of corrections  $C_1, \dots, C_n$ , such that  $\forall i, 1 \leq i \leq n$ ,  $C_i = \langle strict, \pi_i, \varphi_i, \Delta_i \rangle$ . Let  $\Sigma$  be the parallel product of the encoding STSs, and  $\Sigma_{no-labels}$  the simplified  $\Sigma$ . Then  $\Sigma_{strict}$  is the minimization of  $\Sigma_{no-labels}$  according to completed trace equivalence.*

The transformation from an STS back to a process model is done by first removing from  $\Sigma_{strict}$  the actions which were introduced for controlling the parallel product, such as the *?resume* actions. If by removing the corresponding transition we do not remove or introduce new traces, then also the transition can be removed. We can then use the encoding correspondences (between actions and activities, branching conditions, etc.) to map the resulting STS to a process model.

**Example** The solution for the strict corrective evolution problem in our

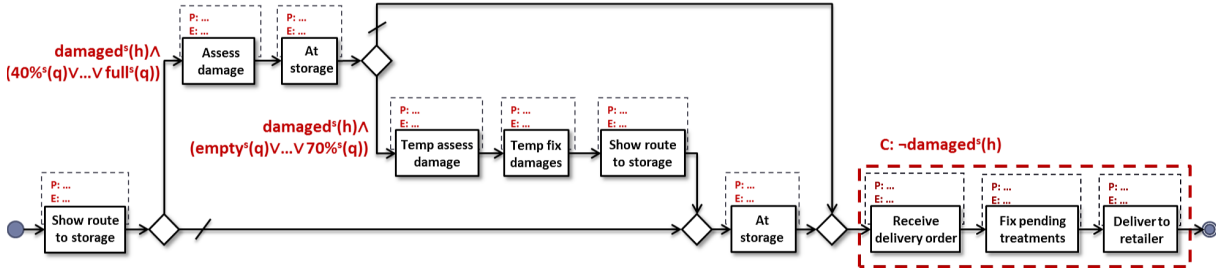


Figure 6.2: Corrected process model: both corrections are strict

scenario is shown in Figure 6.2.

The corrected process model introduces exactly two new traces: one corresponding to the application of the adaptation operation *Schedule repair*, and one corresponding to the application of both *Schedule repair* and *Repair temporarily*.

## 6.2 Correctness of the Approach

In this Section, we prove that the strict corrective evolution approach is correct. In other words, we prove that for a corrective evolution problem defined by a process model  $M_0$ , a goal  $G$ , and a sequence of strict corrections  $C_1, \dots, C_n$ , the process model  $M_{strict}$ , which results by following the approach presented in the previous Section, is a solution to the problem.

The correctness proof is based the correspondence between executions of corrected process models and runs of the encoding STS. A correspondence between an execution and a run is such that each configuration in the execution matches a certain state in the run, and each activity in the execution matches a certain action in the run.

We say that there is a correspondence between a configuration  $\gamma$  of the domain objects  $O$  and a state  $s$  in the encoding STS  $\Sigma$  if for every domain object  $o \in O$ , if  $\gamma(o) = s_o$  then  $s$  is such that  $\Sigma_o$  is in state  $s_o$ .

Let  $a$  be an activity appearing in one of the process models  $M_0, M_1^a, \dots$ ,

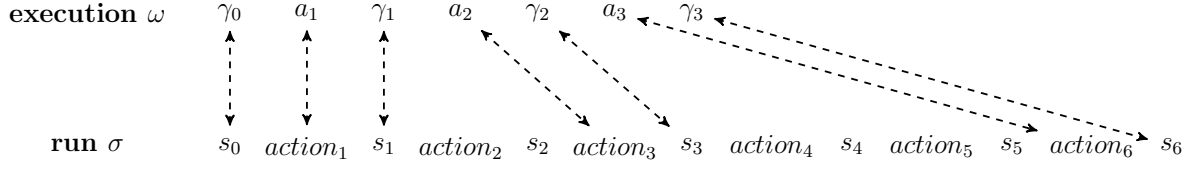


Figure 6.3: Example of a correspondence between an execution and a run

$M_n^a$ . We say that there is a correspondence between  $a$  and an action of  $\Sigma$  if the action was created when translating an activity node in  $M_0, M_1^a, \dots, M_n^a$  labeled with  $a$ , during the translation described in Section 5.2. Note that an activity can have multiple corresponding actions, while an action can have at most one corresponding activity.

We can now define formally the correspondence between executions and runs.

**Definition 21 (Correspondence)**

Let  $M$  be a process model and  $\omega = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k$  an execution of  $M$ . Let  $\Sigma$  be an STS and  $\sigma = s_0, action_1, s_1, \dots, s_n$  a run of  $\Sigma$ . We say that there is a correspondence between  $\omega$  and  $\sigma$  iff:

- there is a correspondence between  $\gamma_0$  and  $s_0$ ;
- for every  $i, 1 \leq i \leq k$ , there exists  $j, i \leq j \leq n$ , such that there is a correspondence between  $a_i$  and  $action_j$ , and between  $\gamma_{i+1}$  and  $s_{j+1}$ ;
- for every  $action_j, 1 \leq j \leq n$  in  $\sigma$ , if  $action_j$  does not correspond to an activity in  $\omega$ , then  $action_j$  does not correspond to any activity in  $M$ .

Note that the correspondence relation between executions and runs does not imply a one-to-one correspondence between configurations and states, or between activities and actions. To better visualize this relation, in Figure 6.3 we show a high-level example of such a correspondence.

In the following, we first prove that there exists a correspondence between the executions of  $M_n$  and the runs of  $\Sigma_n$ , for every  $n \in \mathbb{N}$ . Here,

$M_n$  is the corrected process model and  $\Sigma_n$  the parallel product STS, both obtained at step  $n$ . In the correctness proof, we then use this correspondence to establish the equivalence between  $M_n$  and  $M_{strict}$ , where  $M_{strict}$  is the translation of the simplified and minimized  $\Sigma_n$ .

### 6.2.1 Correspondence Between Executions of a Corrected Model and Runs of the Parallel Product STS

#### Lemma 5

*Assume a corrective evolution problem defined by a process model  $M_0$ , a goal  $G$ , and an empty sequence of corrections. Let  $\Sigma_0$  be the STS encoding. Then for every execution of  $M_0$  there is a corresponding run of  $\Sigma_0$ , and for every run of  $\Sigma_0$  there is a corresponding execution of  $M_0$ .*

*Proof.*  $M_0$  is defined over a set of domain objects  $O = \{o_1, \dots, o_p\}$ . Since there are no corrections, the corresponding STSs are  $\Sigma_{M_0}^0, \Sigma_{semaphore}^0, \Sigma_{\varphi_0}, \dots, \Sigma_{\varphi_m}, \Sigma_{o_1}^0, \dots, \Sigma_{o_p}^0$ . With  $\Sigma_{M_0}^0$  we denoted the translation to an STS of  $M_0$ , according to the rules in Table 5.1. Note that  $\Sigma_{M_0}^0$  does not include ?*resume* actions, which are added only if there are corrections. With  $\Sigma_{\varphi_0}, \dots, \Sigma_{\varphi_m}$  we denoted the encoding of the conditions in  $M_0$ .  $\Sigma_{o_1}^0, \dots, \Sigma_{o_p}^0$  denote the encoding of the domain objects, with transitions on actions corresponding to activities in  $M_0$ , and trigger actions from  $\varphi_0, \dots, \varphi_m$ . Finally,  $\Sigma_{semaphore}^0$  is the semaphore STS when there are no corrections, i.e., it contains only one state  $s_0$  labeled with  $flag_0$ . We need to prove that there is a correspondence between executions of  $M_0$  and runs of the parallel product  $\Sigma_0 = \Sigma_{M_0}^0 \parallel \Sigma_{semaphore}^0 \parallel \Sigma_{\varphi_0} \parallel \dots \parallel \Sigma_{\varphi_m} \parallel \Sigma_{o_1}^0 \parallel \dots \parallel \Sigma_{o_m}^0$ .

**( $\Rightarrow$ ) For every execution of  $M_0$  there is a corresponding run of  $\Sigma_0$ .** Let  $\omega = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k$  be an execution of  $M_0$ . We need to prove that there exists a corresponding run of  $\Sigma_0$ , having the form  $\sigma = s_0, action_1, s_1, \dots$



**base case.** By definition,  $\gamma_0$  is an initial configuration. Let  $s_0$  be a state in  $\Sigma_0$  corresponding to  $\gamma_0$ .  $s_0$  is such that for each domain object  $o$ ,  $\Sigma_o^0$  is in the state specified by  $\gamma_0(o)$ , and  $\Sigma_{M_0}^0$  and  $\Sigma_{\varphi_0}, \dots, \Sigma_{\varphi_m}$  are each in their only initial state. Then, for the execution  $\omega = \gamma_0$  there is a corresponding run of  $\Sigma_0$ ,  $\sigma = s_0$ .

**induction step.** We assume that for execution  $\omega = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{i-2} \xrightarrow{a_{i-1}} \gamma_{i-1}$  there is a corresponding run of  $\Sigma_0$ ,  $\sigma = s_0, action_1, \dots, action_j, s_j, j \geq i-1$ . According to Definition 21, this means that for every activity  $a_1, \dots, a_{i-1}$  there is a corresponding action in  $\sigma$ , and for every configuration  $\gamma_1, \dots, \gamma_{i-1}$  the corresponding state in  $\sigma$  is the state directly following the action which encodes the activity.

We need to prove that for execution  $\omega' = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{i-1} \xrightarrow{a_i} \gamma_i$  there is a corresponding run of  $\Sigma_0$   $\sigma' = s_0, action_1, \dots, action_n, s_n, n > j$ , such that  $action_n$  corresponds to  $a_i$ ,  $action_{j+1}, \dots, action_{n-1}$  do not correspond to activities, and  $s_n$  corresponds to  $\gamma_i$ .

**(a)  $\sigma'$  is a run of  $\Sigma_0$ ,  $action_n$  corresponds to  $a_i$ , and  $action_{j+1}, \dots, action_{n-1}$  do not correspond to activities.**

For the activity  $a_i = \langle name_i, pre_i, eff_i \rangle$  to be part of an execution on  $M_0$ , the activity  $a'_i = \langle name_i, pre'_i, eff_i \rangle$  must be applicable to  $\gamma_{i-1}$ , where  $pre'_i = pre_i \wedge cond$ , and  $cond$  is the conjunction of branch conditions on the path from  $a_{i-1}$  to  $a_i$ .

Assume  $a'_i$  is applicable to  $\gamma_{i-1}$ , i.e.,  $\gamma_{j-1} \models pre'_i$ . Since  $\gamma_{i-1}$  corresponds to  $s_j$ , also  $s_j, \mathcal{F} \models pre'_i$ . This means that also the state  $s'_j$  reached after executing the actions corresponding to the branching conditions is such that  $s'_j, \mathcal{F} \models pre'_i$ . The reason is that the actions corresponding to branching conditions appear either only in  $\Sigma_{M_0}$ , or as *!no-trigger* actions in the condition STSs (if the other branch has a condition which requires uncontrollable events). In both cases, these actions do not trigger state changes in the domain objects STSs. State  $s'_j$  is such that  $\Sigma_{M_0}$  is in a

state from which there is a transition on action  $?name_i$  which corresponds to  $a_i$ . This transition is guarded by  $pre_i$ . Since  $s'_j, \mathcal{F} \models pre'_i$ , the transition on  $?name_i$  in  $\Sigma_0$  is enabled. Therefore, there exists a run of  $\Sigma_0$  such that  $action_n = ?name_i$ , with  $action_{j+1}, \dots, action_{n-1}$  corresponding to XorSplit/AndSplit nodes and branching conditions on the path between the node labeled with  $a_{i-1}$  and the node labeled with  $a_i$ .

Otherwise, if  $a'_i$  is not applicable to  $\gamma_{i-1}$ , there exists a configuration of  $O$   $\gamma'_{i-1}$  such that  $\gamma'_{i-1} \models pre'_i$  and  $\gamma'_{i-1}$  is directly reachable from  $\gamma_{i-1}$ . This is the case when for  $pre'_i$  to hold, one or more uncontrollable events must be triggered. This is accomplished by firing the *!trigger* action in a condition STS. If in  $M_0$  the nodes labeled with  $a_{i-1}$  and  $a_i$  are connected through XorSplit nodes, and the control edges are annotated with conditions which require uncontrollable events, then the action fired is the *!trigger* action in the STS corresponding to the conjunction of these conditions. The state reached after firing the trigger action in  $\Sigma_0$  is a state  $s'_j$  such that  $\Sigma_{M_0}$  has not moved, for each domain object  $o$ ,  $\Sigma_o^0$  is in the state corresponding to  $\gamma'_{i-1}(o)$ , and the condition STSs are in the initial/final state (depending on whether they moved on the trigger action). Then  $s'_j, \mathcal{F} \models pre'_i$  and the transition on  $?name_i$  in  $\Sigma_0$  is enabled. The corresponding run of  $\Sigma_0$  is then such that  $action_n = ?name_i$ , and  $action_{j+1}, \dots, action_{n-1}$  correspond not only to branching nodes and conditions, but can also be *!trigger* actions.

**(b) State  $s_n$  corresponds to  $\gamma_i$ .** We now prove that the state reached in  $\Sigma_0$  after firing  $?name_i$  corresponds to  $\gamma_i$ . Let  $eff_i$  be the effects of  $a_i$ . Further, let  $\gamma_{pre}$  be the configuration satisfying  $pre'_i$  (either  $\gamma_{i-1}$  or  $\gamma'_{i-1}$ ). Then  $\gamma_i$  is as follows. For every domain object  $o = \langle L, L^0, \mathcal{E}, T \rangle$ , if there exists  $e \in eff_i$  such that  $e \in \mathcal{E}$  and there is a transition from  $\gamma_{pre}(o)$  on  $e$  to a state  $s$ , then  $\gamma_i(o) = s$ . Otherwise  $\gamma_i(o) = \gamma_{pre}(o)$ .

In  $\Sigma_0$  this is realized as follows. For every event  $e \in eff_i$  from a domain object  $o$ , there is in  $\Sigma_o^0$  a transition on  $?name_i$  from the state specified

by  $\gamma_{pre}(o)$  to the state specified by  $\gamma_i(o)$ . This transition is enabled, and based on the parallel product definition, the action is fired in  $\Sigma_{M_0}$  and in every matching  $\Sigma_o^0$ . Therefore, the state reached in  $\Sigma_0$  after firing  $?name_i$  corresponds to  $\gamma_i$ .

**(<=) For every run of  $\Sigma_0$ , there is a corresponding execution of  $M_0$ .** Let  $\sigma = s_0, action_1, s_1, \dots$  be a run of  $\Sigma_0$ . Then there exists an execution of  $M_0$   $\omega = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k$  which corresponds to  $\sigma$ .

**base case.**  $s_0$  is an initial state in  $\Sigma_0$ , which means that for every domain object  $o$ ,  $\Sigma_o$  is in an initial state. Therefore, there is an initial configuration  $\gamma_0$  which corresponds to  $s_0$ , and for the run  $\sigma = s_0$  there is a corresponding execution  $\omega = \gamma_0$ .

**induction step.** We assume that the run  $\sigma = s_0, action_1, \dots, action_{i-1}, s_{i-1}$  corresponds to an execution  $\omega = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{j-2} \xrightarrow{a_{j-1}} \gamma_{j-1}$ ,  $1 \leq j \leq i$ .

Let  $\sigma' = s_0, action_1, \dots, action_n, s_n$ ,  $n \geq i$ , be a run of  $\Sigma_0$  such that  $action_n$  is the first action after  $s_{i-1}$  which corresponds to an activity  $a_j$ . For all the intermediate runs between  $\sigma$  and  $\sigma'$  there is a corresponding execution of  $M_0$ , and this is  $\omega$ . We need to prove that there exists an execution  $\omega'$  of  $M_0$  which corresponds to  $\sigma'$ . In other words, we need to prove that  $s_n$  corresponds to a configuration  $\gamma_j$ , and that  $\omega' = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{j-1} \xrightarrow{a_j} \gamma_j$  is an execution of  $M_0$ .

**(a)**  $\gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{j-1} \xrightarrow{a_j} \gamma_j$  **is an execution of  $M_0$ .** Each of the actions  $action_i, \dots, action_{n-1}$  is part of the translation of a control connector from  $M_0$ , or is a *!trigger*/*!no-trigger* action in a condition STS. If any of  $action_i, \dots, action_{n-1}$  is part of the translation of a control connector or is a *!no-trigger* action, then it will only affect the state of  $\Sigma_{M_0}$  or the state of the condition STSs, without affecting the relation between states in  $\sigma$  and configurations. If any of  $action_i, \dots, action_{n-1}$  is a *!trigger* action, then it will affect the state of the condition STS, and the state of domain

object STSs. However, there can be at most one transition on a trigger action, and this is possible only if one or more branching conditions will be tested, and the conjunction of these conditions requires uncontrollable events. Since the other actions do not affect the state of the domain object STSs,  $s_{n-1}$  corresponds either to  $\gamma_{j-1}$ , or, if branching conditions are tested, to a configuration  $\gamma'$  directly reachable from  $\gamma_{j-1}$ .

For the activity  $a_j = \langle name_j, pre_j, eff_j \rangle$  to be part of an execution of  $M_0$ , the activity  $a'_j = \langle name_j, pre'_j, eff_j \rangle$  must be applicable to  $\gamma_{j-1}$ , where  $pre'_j = pre_j \wedge cond$ , and  $cond$  is the conjunction of branch conditions on the path from  $a_{j-1}$  to  $a_j$ . Since  $action_n$  corresponds to  $a_j$ , and to execute  $action_n$  any conditions on the path from  $a_{j-1}$  to  $a_j$ , as well as the precondition of  $a_j$ , must hold in  $s_{n-1}$ , we have that  $s_{n-1}, \mathcal{F} \models pre'_j$ . Since  $s_{n-1}$  corresponds either to  $\gamma_{j-1}$  or to a configuration  $\gamma'$  directly reachable from  $\gamma_{j-1}$ , this means that  $a'_j$  is applicable to  $\gamma_{j-1}$  and there exists a corresponding execution of  $M_0$   $\omega' = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{j-1} \xrightarrow{a_j} \gamma_j$ .

**(b) Configuration  $\gamma_j$  corresponds to state  $s_n$ .** We first prove that there is a transition in  $\Sigma_{M_0}$  from the current state  $s_c$  (the state which is part of the global state  $s_n$ ) on  $action_n$ , and that  $action_n$  is the only action which can be fired from  $s_c$  in  $\Sigma_{M_0}$ .

$\Sigma_0$  moves on  $action_n$  from  $s_{n-1}$ , which means that at least one of the STSs in the parallel product moves on  $action_n$ .  $\Sigma_{semaphore}^0$  has only one state and cannot move. The condition STSs do not move, since  $action_n$  is not a trigger action. In the domain object STS, due to the guards, the only transitions which are enabled are transitions on actions which can be fired in the current state of  $\Sigma_{M_0}$ . Therefore, for  $\Sigma_0$  to move on  $action_n$  from  $s_{n-1}$ , there must be a transition in  $\Sigma_{M_0}$  from the current state on  $action_n$ . Moreover, according to the translation rules in Table 5.1, in  $\Sigma_{M_0}$  from one state there is either only one transition (and the action on this transition corresponds to an activity or to a control connector), or there are

two or more possible transitions, and these correspond to a branching in the model. Since  $action_n$  corresponds to an activity, in  $\Sigma_{M_0}$  the transition on  $action_n$  is the only transition from the current state.

State  $s_n$  is reached in  $\Sigma_0$  after triggering  $action_n$  in  $\Sigma_{M_0}$  and in all domain object STSs for which an event  $e$  was included in the effects of activity  $a_j$ . Therefore,  $s_n$  corresponds to the configuration  $\gamma_j$  reached from  $\gamma_{j-1}$  by applying  $a_j$ . If the domain objects contain nondeterministic events, there can be multiple configurations which are reached from  $\gamma_{j-1}$  by applying  $a_j$ . However, exactly one such configuration corresponds to  $s_n$ .  $\square$

### Lemma 6

*Assume a corrective evolution problem defined by a process model  $M_0$ , a goal  $G$ , and a sequence of corrections  $C_1, \dots, C_n$ , such that  $\forall i, 1 \leq i \leq n$ ,  $C_i = \langle strict, \pi_i, \varphi_i, \Delta_i \rangle$ . For all  $i, 1 \leq i \leq n$ ,  $M_i$  is computed such that  $Exec(M_i) = Exec(M_{i-1}, C_i)$ . Let  $\Sigma_n$  be the STS encoding. Then for every execution of  $M_n$  there is a corresponding run of  $\Sigma_n$ , and for every run of  $\Sigma_n$  there is a corresponding execution of  $M_n$ .*

*Proof.* The proof goes by induction.

**base case,  $n = 0$ .** Follows directly from Lemma 5.

**induction step,  $n-1 \rightarrow n$ .**

$\Sigma_n$  is the parallel product  $\Sigma_{M_0}^n \parallel \Sigma_{\Delta_1} \parallel \dots \parallel \Sigma_{\Delta_n} \parallel \Sigma_{semaphore}^n \parallel \Sigma_{\pi_1} \parallel \dots \parallel \Sigma_{\pi_n} \parallel \Sigma_{\varphi_1} \parallel \dots \parallel \Sigma_{\varphi_m} \parallel \Sigma_{o_1}^n \parallel \dots \parallel \Sigma_{o_p}^n$ .

Here,  $\Sigma_{M_0}^n$  is the extension of  $\Sigma_{M_0}^{n-1}$  with a transition on the  $?resume_n$  action.  $\Sigma_{semaphore}^n$  is the extension of  $\Sigma_{semaphore}^{n-1}$  with a state  $s_n$  and transitions to and from this state. For every domain object  $o$ ,  $\Sigma_o^n$  is the extension of  $\Sigma_o^{n-1}$  with new transitions: on actions corresponding to activities in  $M_n^a$  and on trigger actions introduced by  $\varphi_n$  and conditions in  $M_n^a$ . Since STSs are added or extended in step  $n$ , every run of  $\Sigma_{n-1}$  is also a run of  $\Sigma_n$ .

( $\Rightarrow$ ) for every execution of  $M_n$  there is a corresponding run of  $\Sigma_n$ . We assume that for every execution of  $M_{n-1}$  there is a corresponding run of  $\Sigma_{n-1}$ .

$M_n$  is computed such that  $Exec(M_n) = Exec(M_{n-1}, C_n)$ , where  $C_n = \langle strict, \pi_n, \varphi_n, \Delta_n \rangle$ . From Lemma 1, we know that  $Exec(M_{n-1}) \subseteq Exec(M_{n-1}, C_n)$ . We also know that every run of  $\Sigma_{n-1}$  is also a run of  $\Sigma_n$ . Therefore, for every execution of  $M_n$  which is also an execution of  $M_{n-1}$  there is a corresponding run of  $\Sigma_n$ . (1)

Assume  $\omega$  is an execution of  $M_n$  which is not an execution of  $M_{n-1}$ . Let  $\pi_n = \langle a_1, \dots, a_k \rangle$ . Then the execution  $\omega$  has the form:  $\omega = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k \xrightarrow{a_{k+1}} \gamma_{k+1} \dots \gamma_{l-1} \xrightarrow{a_l} \gamma_l \xrightarrow{a_{l+1}} \gamma_{l+1} \dots \gamma_{m-1} \xrightarrow{a_m} \gamma_m$ , such that:

- there exists  $\gamma'_k$  directly reachable from  $\gamma_k$ , such that  $\gamma'_k \models \varphi_n$ ;
- $\gamma'_k \xrightarrow{a_{k+1}} \gamma_{k+1} \dots \gamma_{l-1} \xrightarrow{a_l} \gamma_l$  is an execution of  $M_n^a$ ;
- $a_{l+1}$  is the activity label of  $to_n$ , and  $\gamma_l \xrightarrow{a_{l+1}} \gamma_{l+1} \dots \gamma_{m-1} \xrightarrow{a_m} \gamma_m$  is part of an execution on  $M_0$ .

The prefix of  $\omega$  up to step  $k$ ,  $\omega' = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k$ , is an execution of  $M_{n-1}$ . Therefore, there exists a run of  $\Sigma_n$   $\sigma = s_0, action_1, s_1, \dots, s_{i-1}, action_i, s_i$  which corresponds to  $\omega'$ . State  $s_i$  corresponds to  $\gamma_k$ . We need to prove that there exists a run  $\sigma'$  having  $\sigma$  as prefix, and which corresponds to  $\omega$ .

If  $\gamma_k \models \varphi_n$ , then also  $s_i, \mathcal{F} \models \varphi_n$ . Otherwise, here exists  $\gamma'_k$ , such that  $\gamma'_k \models \varphi_n$  and  $\gamma'_k$  directly reachable from  $\gamma_k$ . This means that  $\varphi_n$  contains literals which correspond to states reachable through uncontrollable events. In this case, there will be an STS corresponding to  $\varphi_n$ ,  $\Sigma_{\varphi_n}$ , and the next action  $a_{i+1}$  in the run  $\sigma'$  is the *!trigger* action in  $\Sigma_{\varphi_n}$ . The guard of the transition holds since up to this point we followed the run corresponding to trace  $\pi_n$ . The state reached  $s_{i+1}$  is such that  $s_{i+1}, \mathcal{F} \models \varphi_n$ .

Since  $\varphi_n$  holds in the current state, the initial transition in  $\Sigma_{\Delta_n}$  becomes enabled.  $\Sigma_{\Delta_n}$  will move simultaneously with  $\Sigma_{semaphore}$ , which moves to state  $s_n$ . The same mechanism used for generating  $\Sigma_0$  from  $M_0$  has been used for generating  $\Sigma_{\Delta_n}$  from  $M_n^a$ . The difference is that in  $\Sigma_{\Delta_n}$  the initial transitions have extra guards, and there are transitions from the final states on the  $?resume_n$  action. Therefore, similar to the correspondence between executions of  $M_0$  and runs of  $\Sigma_0$ , there is a correspondence between executions of  $M_n^a$  which start from  $\gamma_k$ , respectively  $\gamma'_k$ , and runs of  $\Sigma_n$  which start from state  $s_i$ , respectively  $s_{i+1}$ .

If  $\gamma_k \xrightarrow{a_{k+1}} \gamma_{k+1} \dots \gamma_{l-1} \xrightarrow{a_l} \gamma_l$  is the execution of  $M_n^a$ , then our run  $\sigma'$  is  $s_0, action_1, \dots, action_j, s_j, \dots, j \geq i$ , where for every activity  $a_{k+1}, \dots, a_l$  there is one corresponding action in  $action_{i+1}, \dots, action_j$ , and for every configuration  $\gamma_{k+1}, \dots, \gamma_l$  there is one corresponding state in  $s_{i+1}, \dots, s_j$ , with  $\gamma_l$  corresponding to  $s_j$ . Since  $a_l$  is the label of an End node in  $M_n^a$ , there will be an enabled transition from  $s_j$  on  $?resume_n$ . When this transition fires,  $\Sigma_{\Delta_n}$  moves to its initial state,  $\Sigma_{semaphore}$  moves to  $s_0$ , and  $\Sigma_{M_0}$  moves to a state  $s_{to}$ . In  $\Sigma_{M_0}$  there is a transition from  $s_{to}$  on the  $action_{j+1}$  corresponding to the activity label  $a_{l+1}$  of  $to_n$ . The precondition  $pre$  of  $a_{l+1}$  is satisfied in configuration  $\gamma_l$ , and since  $s_j$  corresponds to  $\gamma_l$  and the domain object STSs do not move on  $?resume_n$ , then also  $s_{j+1}, \mathcal{F} \models pre$ . Due to this and to the fact that  $\Sigma_{semaphore}$  is in state  $s_0$ , the transition on  $action_{j+1}$  corresponding to  $a_{l+1}$  is enabled.

From  $a_{l+1}$  onwards, the activities in execution  $\omega$  are activities from  $M_0$ . In the corresponding run  $\sigma'$ , since all trace STSs are in state  $s_{out}$ , from  $action_{j+1}$   $\Sigma_n$  moves only on transitions corresponding to  $M_0$ . These are either transitions in  $\Sigma_{M_0}$  which trigger also transitions in the domain object STSs, or transitions in the STSs of conditions in  $M_0$ . The proof that for the sequence  $\gamma_l \xrightarrow{a_{l+1}} \gamma_{l+1} \dots \gamma_{m-1} \xrightarrow{a_m} \gamma_m$  in execution  $\omega$  there is a corresponding sequence  $s_j, action_{j+1}, \dots, action_p, s_p$  in the run  $\sigma'$ , is then the same as the

proof of a correspondence between an execution of  $M_0$  and a run of  $\Sigma_0$  in Lemma 5. Therefore, also for every execution  $\omega$  of  $M_n$  that is not an execution of  $M_{n-1}$  there exists a corresponding run  $\sigma'$  of  $\Sigma_n$ . (2)

From (1) and (2), we have that for every execution of  $M_n$  there is a corresponding run of  $\Sigma_n$ .

**( $\Rightarrow$ ) for every run of  $\Sigma_n$  there is a corresponding execution of  $M_n$ .**

We assume that for every run of  $\Sigma_{n-1}$  there is a corresponding execution of  $M_{n-1}$ . We know that every run of  $\Sigma_{n-1}$  is a run of  $\Sigma_n$ . Moreover, from Lemma 1 we know that  $Exec(M_{n-1}) \subseteq Exec(M_n)$ , and therefore every execution of  $M_{n-1}$  is also an execution of  $M_n$ . Then, for every run of  $\Sigma_n$  which is also a run of  $\Sigma_{n-1}$  there is a corresponding execution of  $M_n$ . (1)

Let  $\sigma$  be a run of  $\Sigma_n$  which is not a run of  $\Sigma_{n-1}$ . Then  $\sigma$  must be such that at some point transitions in  $\Sigma_{\Delta_n}$  and (if it exists) in  $\Sigma_{\varphi_n}$  were triggered. For transitions in  $\Sigma_{\Delta_n}$  and  $\Sigma_{\varphi_n}$  to be possible, the trace STS  $\Sigma_{\pi_n}$  must reach the state labeled with  $trace_n$ , and at the same time one of  $\Sigma_{M_0}, \Sigma_{\Delta_1}, \dots, \Sigma_{\Delta_{n-1}}$  must be in a state labeled with  $point_n$ . Then  $\sigma = s_0, action_1, \dots, action_i, s_i, \dots$ , where  $s_i$  is such that  $s_i, \mathcal{F} \models trace_n \wedge point_n$ . The prefix of  $\sigma$  up to step  $i$  is a run  $\sigma'$  which is also a run of  $\Sigma_{n-1}$ . Therefore there exists an execution of  $M_n$  which corresponds to  $\sigma'$ , and since  $s_i, \mathcal{F} \models trace_n$ , this execution is  $\omega' = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k$ .

The run  $\sigma$  is such that transitions in  $\Sigma_{\Delta_n}$  were triggered at some point, and this point can only be state  $s_i$ , or a state  $s'_i$  such that the transitions between  $s_i$  and  $s'_i$  are triggered on actions from condition STSs.

(a) If  $s_i, \mathcal{F} \models \varphi_n$ , there are two possibilities. If  $\varphi_n$  does not include uncontrollable literals, then  $\Sigma_{\varphi_n}$  has not been created, and the initial transition in  $\Sigma_{\Delta_n}$  is enabled from  $s_i$ . Otherwise,  $\varphi_n$  does include uncontrollable literals, and  $\Sigma_{\varphi_n}$  has been created. The action in  $\sigma$  which follows  $s_i$  is the *!no-trigger<sub>n</sub>* action in  $\Sigma_{\varphi_n}$ . The next state  $s'_i$  will have the same labels



as  $s_i$ , since only  $\Sigma_{\varphi_n}$  moves on  $!no-trigger_n$ . Configuration  $\gamma_k$  corresponds to  $s_i$  and in the second case also to  $s'_i$ . Therefore,  $\gamma_k \models \varphi_n$ , and the first activity in  $M_n^a$  is applicable to  $\gamma_k$ .

(b) If  $s_i, \mathcal{F} \not\models \varphi_n$ , since transitions in  $\Sigma_{\Delta_n}$  are triggered in  $\sigma$ , this can happen only if  $\varphi_n$  contains uncontrollable literals. The next action in  $\sigma$  is then the  $!trigger_n$  from  $\Sigma_{\varphi_n}$ . The relevant domain object STSs move on  $!trigger_n$ , and the next state  $s'_i$  is such that  $s'_i, \mathcal{F} \models \varphi_n$ . In this case, there exists a configuration  $\gamma'_k$  directly reachable from  $\gamma_k$  such that  $\gamma'_k$  corresponds to  $s'_i$ , the first activity in  $M_n^a$  is applicable to  $\gamma'_k$ .

In the run  $\sigma$ , the next action  $action_{i+1}$  corresponds to the initial transition in  $\Sigma_{\Delta_n}$ . Since both  $\Sigma_{\Delta_n}$  and  $\Sigma_{semaphore}$  move, the next state  $s_{i+1}$  is such that  $s_{i+1}, \mathcal{F} \models flag_n$ . Therefore, until a transition on  $?resume_n$  is reached in  $\Sigma_{\Delta_n}$ , the next transitions in  $\sigma$  are controlled by  $\Sigma_{\Delta_n}$ . Although transitions in the domain objects STSs and conditions STSs may be triggered as well, these are controlled by the state of  $\Sigma_{\Delta_n}$  through guards. Therefore, for the run  $\sigma'' = s_0, action_1, \dots, action_j, s_j, \dots, j > i$ , where  $action_j = ?resume_n$ , there is a corresponding execution of  $M_n$ ,  $\omega'' = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k \dots \gamma_{l-1} \xrightarrow{a_l} \gamma_l$ , where  $\gamma_k \xrightarrow{a_{k+1}} \gamma_{k+1} \dots \gamma_{l-1} \xrightarrow{a_l} \gamma_l$  is an execution of  $M_n^a$ .

Since  $\Sigma_{\Delta_n}$ ,  $\Sigma_{M_0}$ , and  $\Sigma_{semaphore}$  all move on  $?resume_n$ , state  $s_j$  is such that  $s_j, \mathcal{F} \models flag_0$  and  $\Sigma_{M_0}$  is in a state  $s_{to}$  from which there is a transition on the  $action_{j+1}$  corresponding to the activity label of  $to_n$ . All trace STSs are in state  $s_{out}$ , with the last  $\Sigma_{\pi_n}$  having moved to  $s_{out}$  on the first action in  $\Sigma_{\Delta_n}$  corresponding to an activity. Therefore, from  $s_j$  and until the end of the run  $\sigma$ , the only STSs that can move are  $\Sigma_{M_0}$  and, controlled by  $\Sigma_{M_0}$ , the domain object STSs and condition STSs. The proof that for the sequence in the run  $\sigma$  from  $s_j$  until the last state  $s_p$ ,  $s_j, action_{j+1}, \dots, action_p, s_p$ , there is a corresponding sequence  $\gamma_l \xrightarrow{a_{l+1}} \gamma_{l+1} \dots \gamma_{m-1} \xrightarrow{a_m} \gamma_m$  in the execution  $\omega$  is then the same as the proof of a correspondence between a run of  $\Sigma_0$

and an execution of  $M_0$  in Lemma 5. Therefore, also for a run of  $\Sigma_n$  which is not a run of  $\Sigma_{n-1}$  there is a corresponding execution of  $M_n$ . (2)

From (1) and (2), we have that for every run of  $\Sigma_n$  there is a corresponding execution of  $M_n$ .  $\square$

### 6.2.2 Correctness

#### Theorem 1 (Correctness)

*Assume a corrective evolution problem defined by a process model  $M_0$ , a goal  $G$ , and a sequence of corrections  $C_1, \dots, C_n$ , such that  $\forall i, 1 \leq i \leq n$ ,  $C_i = \langle \text{strict}, \pi_i, \varphi_i, \Delta_i \rangle$ . Let  $M_{\text{strict}}$  be the translation of  $\Sigma_{\text{strict}}$ . Then  $M_{\text{strict}}$  is a solution for the corrective evolution problem defined by  $M_0$ ,  $G$ , and  $C_1, \dots, C_n$ .*

*Proof.* To prove that  $M_{\text{strict}}$  is a solution for the corrective evolution problem defined by  $M_0$ ,  $G$ , and  $C_1, \dots, C_n$ , we have to prove that  $M_{\text{strict}} \equiv M_n$ , where  $\forall i, 1 \leq i \leq n$ ,  $M_i$  is computed such that  $\text{Exec}(M_i) = \text{Exec}(M_{i-1}, C_i)$ . This is sufficient for proving also that  $M_{\text{strict}}$  satisfies  $G$ , since we know that any  $M_n$  satisfies  $G$  from Lemma 4.

$M_{\text{strict}}$  is the translation of  $\Sigma_{\text{strict}}$ , and  $\Sigma_{\text{strict}}$  is the minimization of the simplified parallel product  $\Sigma$ . Based on Lemma 6, we know that there is a correspondence between runs of  $\Sigma$  and executions of  $M_n$ . This correspondence is maintained after simplifying  $\Sigma$ . The simplification first removes the transitions which cannot fire, and this does not affect the runs of  $\Sigma$ . The second step is to remove the guards and the labeling function. Since the transitions which remain in  $\Sigma$  are all fireable, this second step also does not affect the runs of  $\Sigma$ .

$\Sigma_{\text{strict}}$  has less states than the simplified  $\Sigma$ , but the same set of complete traces.  $M_{\text{strict}}$  is translated from  $\Sigma_{\text{strict}}$  such that there exists a one-to-one correspondence between complete traces of  $M_{\text{strict}}$  and complete traces of  $\Sigma_{\text{strict}}$ . This means that there is a one-to-one correspondence between

complete traces of  $M_{strict}$  and complete traces of  $\Sigma$ . Further, with Lemma 6, we know that there is a one-to-one correspondence between complete traces of  $M_n$  and complete traces of  $\Sigma$ . Therefore  $M_{strict}$  has the same complete traces as  $M_n$ , i.e.,  $Traces(M_{strict}) = Traces(M_n)$ .

With minimization, the conditions actions are not removed, and so in the translated  $M_{strict}$  these will be replaced with the branch conditions from  $M_0, M_1^a, \dots, M_n^a$ . Moreover, the activities are copied to  $M_{strict}$  with their preconditions and effects from  $M_0, M_1^a, \dots, M_n^a$ . Finally, for each correction  $C_i$ , we add a branch condition on  $\varphi_i$  before the application of the first activity in the adaptation model  $M_i^a$ . Therefore, if for some complete trace  $\pi$  of  $M_n$ , there is a condition appearing between two consecutive activities  $a$  and  $a'$ , then this condition appears also in  $M_{strict}$  for the complete trace  $\pi$  between  $a$  and  $a'$ . Since  $M_{strict}$  has the same complete traces as  $M_n$ , and also the same conditions and preconditions, then  $M_{strict}$  will allow the same executions as  $M_n$ .  $\square$

### 6.2.3 Completeness

#### Theorem 2 (Completeness)

*Assume a corrective evolution problem defined by a process model  $M_0$ , a goal  $G$ , and a sequence of corrections  $C_1, \dots, C_n$ , such that  $\forall i, 1 \leq i \leq n$ ,  $C_i = \langle strict, \pi_i, \varphi_i, \Delta_i \rangle$ . Then the strict corrective evolution approach always terminates and returns a solution  $M_{strict}$ .*

*Proof.* The steps in the strict corrective evolution approach are: encoding the inputs into STSs, building the parallel product STS  $\Sigma$ , minimizing  $\Sigma$ , and translating back into a process model. It is easy to see that each of these steps involves a finite number of operations. To see that the construction of  $\Sigma$  terminates, note that  $\Sigma$  has a finite number of states. Therefore, the approach always terminates on a strict corrective evolution

problem.

We know that there exists at least a solution to the problem defined by  $M_0, G$ , and  $C_1, \dots, C_n$ . This follows directly from Lemma 2 (existence of a corrected process model) and Lemma 4 (strict corrections do not affect goal satisfaction). Since the strict corrective evolution approach terminates, according to Theorem 1, it will return a process model  $M_{strict}$  which is a solution to the problem.

### 6.3 Discussion

In the strict corrective evolution approach, we minimize the parallel product STS according to completed trace equivalence (also called language equivalence). Completed trace equivalence is only one of the many possible equivalence relations for nondeterministic state transition systems. Choosing the appropriate equivalence relation is not a straightforward task. In general, the relation should preserve the properties of interest, should be efficient to compute, and at the same time be as coarse as possible.

Equivalences for transition systems have been classified along four different lines, corresponding to the differences in behavior that they distinguish [112]: branching time vs. linear time, interleaving vs. partial order semantics, treatment of internal actions, finite vs. infinite observation. Our STSs can perform at most one action at a time (sequential), they have no internal actions (concrete), and each state has a finite number of next states (finitely branching). Therefore, the relevant equivalences range in the linear time - branching time spectrum, with trace equivalence as the coarsest relation, and bisimulation the finest.

The property which must be maintained is that the minimized STS should have the same set of complete traces as the original STS. From this point of view, there are a range of equivalence relations in the linear

time - branching time spectrum which are all appropriate. The coarsest relation among them is completed trace equivalence, which therefore has the advantage of resulting in the greatest reduction. The disadvantage is that while bisimulation is decidable in polynomial time, (completed) trace equivalence is PSPACE-complete [41]. There is however one final criteria for deciding on the equivalence relation. This is the fact that to translate the minimized STS to a process model, this STS must be deterministic. We can always obtain a deterministic STS which is (completed) trace equivalent to our nondeterministic STS. Because of these criteria, completed trace equivalence is the only equivalence relation which is appropriate in our setting.

Since completed trace equivalence is the coarsest equivalence relation which preserves the set of complete traces, the minimized  $\Sigma_{strict}$  is actually a minimal STS according to this property. The process model  $M_{strict}$  is obtained by translating the minimal  $\Sigma_{strict}$  using the correspondences created when encoding the elements into STSs. Therefore, if the original process model  $M_0$  and the adaptation models  $M_1^a, \dots, M_n^a$  do not contain parallelism,  $M_{strict}$  will be the minimal solution to the strict corrective evolution problem. However, if any of  $M_0, M_1^a, \dots, M_n^a$  contain parallelism, this can be restored by applying post-processing techniques to  $M_{strict}$ .

Although we minimize the parallel product of our STSs, the resulting  $\Sigma_{strict}$  will still contain redundant states and transitions. As a translation of  $\Sigma_{strict}$ , the process model  $M_{strict}$  may potentially contain many duplicated activities, i.e., activity nodes which are labeled with the same activity. The reason is that with strict corrective evolution we construct our new process model by unfolding the original process model according to partial traces. If an adaptation includes a jump to an activity node for which the activity has already been executed, the activities in between this activity node and the current activity node will be duplicated in the new process

model. This redundancy in the new model is therefore intrinsic to strict corrective evolution, and can be removed by relaxing the corrections.

# Chapter 7

## Relaxed Corrective Evolution

In this Chapter, we design an automated technique for solving a second case of corrective evolution problems, the case when all corrections are either strict or relaxed.

By applying relaxed corrections, we may introduce “spurious” executions, that is, executions which do not achieve the goal; such executions are avoided with strict corrections. For relaxed corrective evolution it is therefore necessary not only to compose the process model and adaptations, but also to verify that the composition satisfies the goal. For this purpose, we devise a solution based on planning. The solution for strict corrective evolution is not sufficient, since it performs no verification.

We first encode the process model, domain objects, partial traces, conditions, and adaptations into STSs. This time, we encode also the goal. We compute the parallel product of all STSs. We use this parallel product as a planning domain and create a planning goal from the process goal. We then apply the approach in [9, 10], which generates a controller for the planning domain, in such a way as to satisfy the planning goal. If a controller exists, we translate it to a new process model using the encoding correspondences.

As in the previous case, we first present an overview of our automated

technique in Section 7.1. We prove that the approach is correct and complete in Section 7.2. We conclude with a discussion in Section 7.3.

## 7.1 Description of the Approach

A relaxed corrective evolution problem is defined by a process model  $M_0$  which satisfies a goal  $G$ , and a sequence of corrections  $C_1, \dots, C_n$  having either the type strict or relaxed. In other words, for all  $i$ ,  $1 \leq i \leq n$ ,  $C_i = \langle ct, \pi_i, \varphi_i, \Delta_i \rangle$ , and  $ct \in \{strict, relaxed\}$ .

An overview of the solution for relaxed corrective evolution is shown in Figure 7.1. As for strict corrective evolution, we first encode the process model into an STS  $\Sigma_{M_0}$ , the adaptations into  $\Sigma_{\Delta_1}, \dots, \Sigma_{\Delta_n}$ ,  $\Sigma_{semaphore}$ , the partial traces into  $\Sigma_{\pi_1}, \dots, \Sigma_{\pi_n}$ , the conditions into  $\Sigma_{\varphi_1}, \dots, \Sigma_{\varphi_m}$ , and the domain objects into  $\Sigma_{o_1}, \dots, \Sigma_{o_p}$ , as described in Chapter 5. This time, we encode also the goal statements included in  $G$  into  $\Sigma_{g_1}, \dots, \Sigma_{g_k}$ .

We then compute their parallel product. To distinguish the new parallel product from the parallel product  $\Sigma$  obtained with the strict approach in Section 6.1, we denote the new STS with  $\Sigma^+$ .  $\Sigma^+$  differs from  $\Sigma$  in two respects. First, in  $\Sigma^+$  the goal statements STSs  $\Sigma_{g_1}, \dots, \Sigma_{g_k}$  are taken into account. Second, if a correction  $C_i$  is relaxed, the corresponding trace STS  $\Sigma_{\pi_i}$  will contain only one state labeled with *trace<sub>i</sub>*. This way, the partial trace is not taken into account when the adaptation must be plugged in a relaxed way, and at the same time the other STSs which have the proposition *trace<sub>i</sub>* guarding their initial transitions do not need to be modified.

$\Sigma^+$  is as follows:

$$\begin{aligned} \Sigma^+ = & \Sigma_{M_0} \parallel \Sigma_{\Delta_1} \parallel \dots \parallel \Sigma_{\Delta_n} \parallel \Sigma_{semaphore} \parallel \Sigma_{\pi_1} \parallel \dots \parallel \Sigma_{\pi_n} \parallel \\ & \Sigma_{\varphi_1} \parallel \dots \parallel \Sigma_{\varphi_m} \parallel \Sigma_{o_1} \parallel \dots \parallel \Sigma_{o_p} \parallel \Sigma_{g_1} \parallel \dots \parallel \Sigma_{g_k} \end{aligned}$$

From  $\Sigma^+$  we first remove the transitions which can never fire, followed



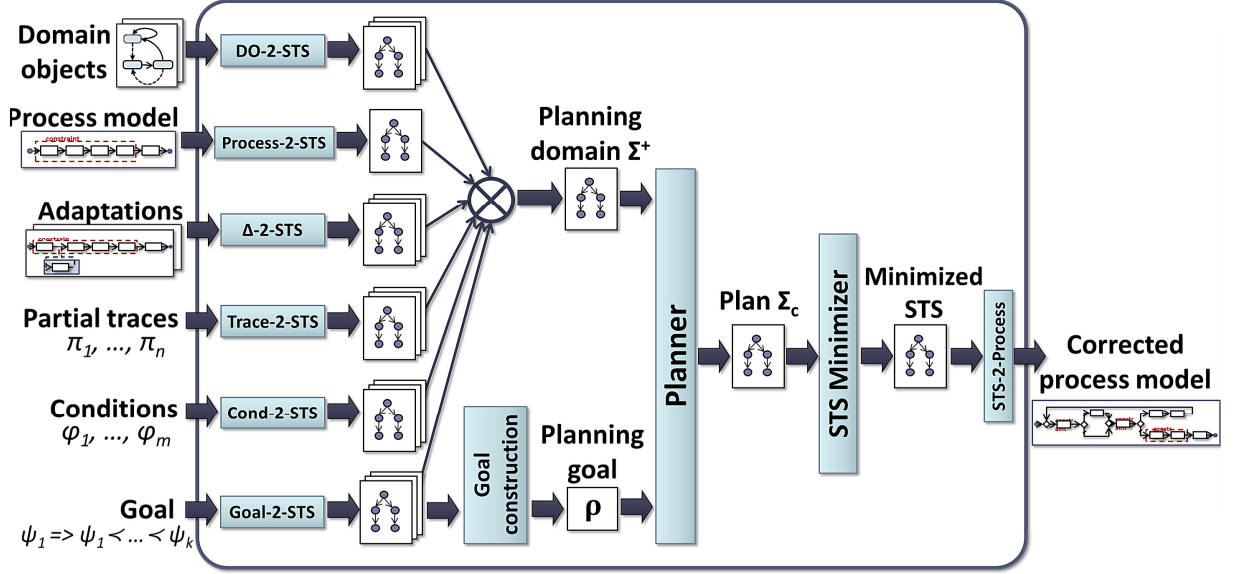


Figure 7.1: Relaxed corrective evolution: solution overview

by the guards and the labeling function. The resulting STS is our planning domain. We construct the planning goal  $\rho$  by combining the requirements generated in 5.7 for each goal statement. To combine the requirements, we use the flattening procedure described in [101], which returns a list of boolean formulas defined over the elements in the requirements, sorted to represent the combined preferences of these requirements.

On the planning domain  $\Sigma^+$  and planning goal  $\rho$ , we apply the technique proposed in [9, 10], which generates a controller  $\Sigma_c$  for the planning domain such that the controlled system satisfies the planning goal, i.e.,  $\Sigma_c \triangleright \Sigma^+ \models \rho$ . We use the following notion of a controlled system.

### Definition 22 (Controlled System)

Let  $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{S}^F, \mathcal{F} \rangle$  and  $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{I}, \mathcal{O}, \mathcal{R}_c, \mathcal{S}_c^F, \mathcal{F}_c \rangle$  be two STSs. STS  $\Sigma_c \triangleright \Sigma$ , describing the behaviors of system  $\Sigma$  when controlled by  $\Sigma_c$ , is defined as:

$$\Sigma_c \triangleright \Sigma = \langle \mathcal{S}_c \times \mathcal{S}, \mathcal{S}_c^0 \times \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}_c \triangleright \mathcal{R}, \mathcal{S}_c^F \times \mathcal{S}^F, \mathcal{F}_c \cup \mathcal{F} \rangle$$

where:  $\langle (s_c, s), (b_c \wedge b), a, (s'_c, s') \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$  if  $\langle s_c, b_c, a, s'_c \rangle \in \mathcal{R}_c$  and  $\langle s, b, a, s' \rangle \in \mathcal{R}$ .

We shortly describe the steps for computing  $\Sigma_c$ , according to the approach proposed in [9]. The first step in this approach is to build a belief-level system, by compiling away the silent actions in the given STS. Since our  $\Sigma^+$  does not contain silent actions, in our case this step is skipped. The parallel product  $\Sigma^+ = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{S}^F, \mathcal{F} \rangle$  is then interpreted as a fully observable planning domain  $D = \langle S, S^0, A, E, R, P \rangle$  which comprises standard actions  $A$  and exogenous events  $E$ . States in  $\Sigma^+$  are mapped into states in  $D$ , i.e.,  $S = \mathcal{S}$  and  $S^0 = \mathcal{S}^0$ . The labeling of states and the transition relation are preserved. The input actions  $\mathcal{I}$  are mapped into planning actions  $A$ , while the output actions  $\mathcal{O}$  correspond to exogenous events  $E$ .

The planning algorithm has two key steps. The first step is to restrict the domain  $D$  to states which are recoverable. Considering that the domain can move autonomously on exogenous events, these are the states for which it is possible, by executing a suitable course of action, to reach the states satisfying  $\rho$ . The second step is to identify, for each state in the restricted domain, which is the best action that must be performed to achieve the goals in  $\rho$  according to their preference order. The output of this algorithm is a plan whose possible runs on the domain are either finite and terminating in goal states, or infinite and traversing the goal states infinitely often. This plan is optimal, always performing the action which leads to the best achievable goal.

This optimal plan is our controller  $\Sigma_c$ . If  $\Sigma_c$  exists, it corresponds to a synthesis of the original process model with the adaptations, which achieves the process goal. We minimize  $\Sigma_c$  according to complete trace equivalence and obtain  $\Sigma_{relaxed}$ , which we translate back into a process model.

**Definition 23** ( $\Sigma_{relaxed}$ )

Assume a corrective evolution problem defined by a process model  $M_0$ , a goal  $G$ , and a sequence of corrections  $C_1, \dots, C_n$ , such that  $\forall i, 1 \leq i \leq n$ ,  $C_i = \langle ct_i, \pi_i, \varphi_i, \Delta_i \rangle$ ,  $ct_i \in \{strict, relaxed\}$ . Let  $\Sigma^+$  be the parallel product of the encoding STSs,  $\Sigma_{no-labels}^+$  the simplified  $\Sigma^+$ , and  $\rho$  the planning goal. If there exists  $\Sigma_c$  such that  $\Sigma_c \triangleright \Sigma^+ \models \rho$ , then  $\Sigma_{relaxed}$  is the minimization of  $\Sigma_c$  according to completed trace equivalence.

**Example** In our scenario, we consider three problems for which at least one correction in the input is relaxed, and which can be solved with the relaxed corrective evolution approach.

The first problem corresponds to the case when correction  $C_1$  is relaxed and correction  $C_2$  is strict. The process model obtained in this case is shown in Figure 7.2. Note that the *Schedule repair* adaptation is applied two times. The first time is the strict case, when the car is damaged on the way to the storage area. The second time is due to the fact that the correction is relaxed. In this case, the car is again on the way to the storage area and is damaged a third time, after having applied the adaptation *Schedule repair* for the first damage, and the adaptation *Repair temporarily* for the second damage. Since the second correction is strict, the *Repair temporarily* adaptation is applied only once, when the car is damaged at the storage area.

The second problem corresponds to the case when correction  $C_1$  is strict and correction  $C_2$  is relaxed. The process model which results in this case is shown in Figure 7.3. Since the first correction is strict, the *Schedule repair* adaptation is applied only once, when the car is damaged the first time on the way to the storage area, and there are too many cars already in the queue. The second correction is relaxed, and therefore the *Repair temporarily* adaptation is applied not only once, but every time the car is

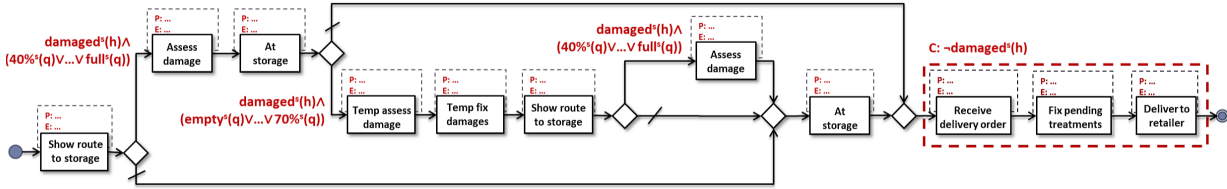


Figure 7.2: Corrected process model: correction  $C_1$  is relaxed,  $C_2$  is strict

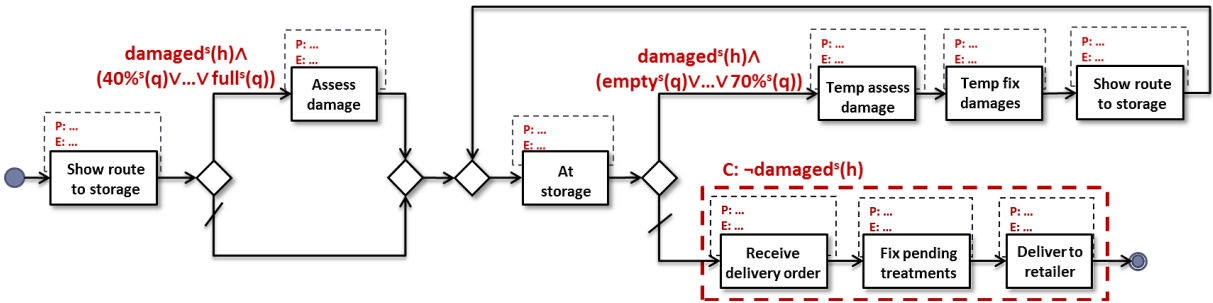


Figure 7.3: Corrected process model: correction  $C_1$  is strict,  $C_2$  is relaxed

damaged at the storage area and the queue is not full.

The third and final problem corresponds to the case when both corrections  $C_1$  and  $C_2$  are relaxed. This is the least restrictive case of the three. The process model which results after applying the relaxed corrective evolution approach is shown in Figure 7.4. Note that the adaptations *Schedule repair* and *Repair temporarily* are applied when the car is damaged on the way to storage, and respectively at storage, independent of how many damages already occurred.

### Comparison to strict corrective evolution

By applying corrections, we are introducing behavior into the process model. Therefore, the process model that results by applying relaxed corrective evolution will allow more executions than the original process

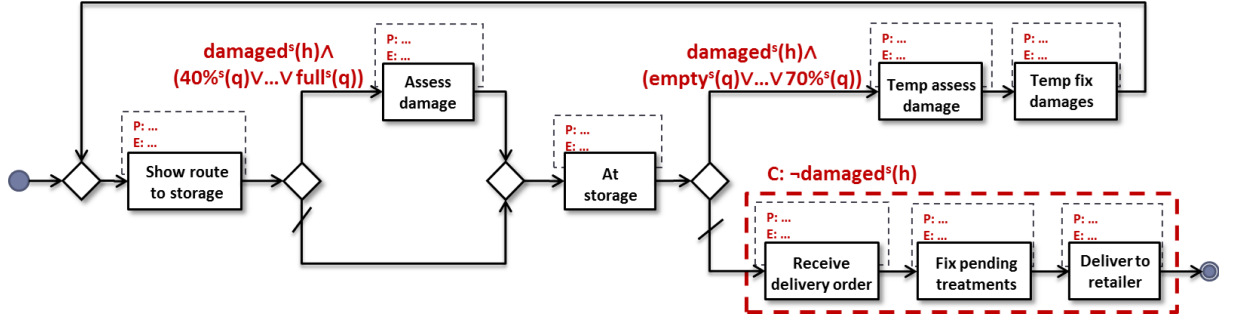


Figure 7.4: Corrected process model: both corrections are relaxed

model. Moreover, this process model will also allow more executions than the process model obtained with strict corrective evolution. In other words, when applying relaxed corrections instead of the corresponding strict corrections, we are introducing more behavior into the process model.

To see why this is the case, assume we have a strict corrective evolution problem defined by a process model  $M$ , a goal  $G$ , and a sequence of corrections  $C_1, \dots, C_n$ , such that for all  $i$ ,  $1 \leq i \leq n$ ,  $C_i = \langle \text{strict}, \pi_i, \varphi_i, \Delta_i \rangle$ . Then for all  $i$ ,  $M_{\text{strict}}$  will be able to replay all the executions that result by applying  $\Delta_i$  to  $M_{i-1}$  at  $\pi_i$  for  $\varphi_i$ . We now replace one or more corrections in  $C_1, \dots, C_n$  with corresponding relaxed corrections. Then for all  $i$ , if  $C_i$  is relaxed,  $M_{\text{relaxed}}$  will be able to replay all the executions that result by applying  $\Delta_i$  to  $M_{i-1}$  at  $\pi$  for  $\varphi_i$ , where  $\pi$  is any trace on  $M_{i-1}$  which ends with the activity corresponding to  $to_i$ . Since  $\pi_i$  is one such trace,  $M_{\text{relaxed}}$  can replay all the executions of  $M_{\text{strict}}$ .

## 7.2 Correctness of the Approach

In this Section, we prove that the relaxed corrective evolution approach is correct. We consider a corrective evolution problem defined by a process model  $M_0$ , a goal  $G$ , and a sequence of corrections  $C_1, \dots, C_n$ , such that each correction  $C_i$ ,  $1 \leq i \leq n$ , has either the type strict or relaxed.

We prove that the process model  $M_{relaxed}$ , which results by following the approach presented in the previous Section, is a solution to this problem.

In the following, we first prove that there exists a correspondence between the executions of  $M_n$  and the runs of  $\Sigma_n^+$ , for every  $n \in \mathbb{N}$ . Here,  $M_n$  is the corrected process model and  $\Sigma_n^+$  the parallel product STS, both obtained at step  $n$ . We then show that if  $M_n$  satisfies the goal  $G$ , then there exists a correspondence also between complete executions of  $M_n$  and complete runs of  $\Sigma_c^n$ , where  $\Sigma_c^n$  is the controller of  $\Sigma_n^+$ . In the correctness proof, we then use this second correspondence to establish the equivalence between a process model  $M_n$  which satisfies  $G$  and  $M_{relaxed}$ , where  $M_{relaxed}$  is the translation of the minimized  $\Sigma_c^n$ .

### 7.2.1 Correspondence Between Executions of a Corrected Model and Runs of the Parallel Product STS

#### Lemma 7

*Assume a corrective evolution problem defined by a process model  $M_0$ , a goal  $G$ , and an empty sequence of corrections. Let  $\Sigma_0^+$  be the STS encoding. Then for every execution of  $M_0$  there is a corresponding run of  $\Sigma_0^+$ , and for every run of  $\Sigma_0^+$  there is a corresponding execution of  $M_0$ .*

*Proof.*  $M_0$  is defined over a set of domain objects  $O = \{o_1, \dots, o_p\}$  and a set of activities  $\mathcal{A}$ . Since there are no corrections, the corresponding STSs are  $\Sigma_{M_0}^0$ ,  $\Sigma_{semaphore}^0$ ,  $\Sigma_{\varphi_0, \dots, \varphi_m}$ ,  $\Sigma_{o_1}^0, \dots, \Sigma_{o_p}^0$ ,  $\Sigma_{g_1, \dots, g_k}$ .

$\Sigma_{M_0}^0$  is the translation to an STS of  $M_0$ , according to the rules in Table 5.1.  $\Sigma_{\varphi_0, \dots, \varphi_m}$  encode the conditions in  $M_0$ .  $\Sigma_{o_1}^0, \dots, \Sigma_{o_p}^0$  encode the domain objects, with transitions on actions corresponding to activities in  $M_0$ , and on trigger actions from  $\varphi_0, \dots, \varphi_m$ .  $\Sigma_{semaphore}^0$  is the semaphore STS when there are no corrections, i.e., it contains only one state  $s_0$  labeled with  $flag_0$ . Finally,  $\Sigma_{g_1, \dots, g_k}$  are the STSs corresponding to the goal statements in  $G$ .

Their parallel product is:

$$\Sigma_0^+ = \Sigma_{M_0}^0 \parallel \Sigma_{semaphore}^0 \parallel \Sigma_{\varphi_0} \parallel \dots \parallel \Sigma_{\varphi_m} \parallel \Sigma_{o_1}^0 \parallel \dots \parallel \Sigma_{o_m}^0 \parallel \Sigma_{g_1} \parallel \dots \parallel \Sigma_{g_k}$$

( $\Rightarrow$ ) **For every execution of  $M_0$ , there is a corresponding run of  $\Sigma_0^+$ .** From Lemma 5, we know that there is a correspondence between executions of  $M_0$  and runs of  $\Sigma_0$ .

By construction,  $\Sigma_0^+ = \Sigma_0 \parallel \Sigma_{g_1} \parallel \dots \parallel \Sigma_{g_k}$ , and the actions in  $\Sigma_{g_1}, \dots, \Sigma_{g_k}$  do not appear in  $\Sigma_0$ . Let  $\omega$  be an execution of  $M_0$ , and  $\sigma$  the corresponding run in  $\Sigma_0$ . Then there exists at least one run  $\sigma'$  of  $\Sigma_0^+$ , such that the only difference between  $\sigma$  and  $\sigma'$  is that also transitions in the goal STSs may be triggered. The correspondence is maintained between  $\omega$  and  $\sigma'$ . Therefore, for every execution of  $M_0$  there is a corresponding run of  $\Sigma_0^+$ .

( $\Rightarrow$ ) **For every run of  $\Sigma_0^+$ , there is a corresponding execution of  $M_0$ .** For every run  $\sigma$  of  $\Sigma_0^+$  there is a run  $\sigma'$  of  $\Sigma_0$ , the only difference between  $\sigma$  and  $\sigma'$  being that in  $\sigma$  there are also transitions in the goal STSs. Since every run of  $\Sigma_0$  corresponds to an execution of  $M_0$ , we have that also every run of  $\Sigma_0^+$  corresponds to an execution of  $M_0$ .  $\square$

### Lemma 8

*Assume a corrective evolution problem defined by a process model  $M_0$ , a goal  $G$ , and a sequence of corrections  $C_1, \dots, C_n$ , such that  $\forall i, 1 \leq i \leq n$ ,  $C_i = \langle ct, \pi_i, \varphi_i, \Delta_i \rangle$ , and  $ct \in \{strict, relaxed\}$ . For all  $i, 1 \leq i \leq n$ ,  $M_i$  is computed such that  $Exec(M_i) = Exec(M_{i-1}, C_i)$ . Let  $\Sigma_n^+$  be the generated encoding. Then for every execution of  $M_n$  there is a corresponding run of  $\Sigma_n^+$ , and for every run of  $\Sigma_n^+$  there is a corresponding execution of  $M_n$ .*

*Proof.* The proof goes by induction.

**base case,  $n = 0$ .** Follows directly from Lemma 7.

**induction step,  $n-1 \rightarrow n$ .**  $\Sigma_n^+$  is the parallel product  $\Sigma_{M_0}^n \parallel \Sigma_{\Delta_1} \parallel \dots \parallel \Sigma_{\Delta_n} \parallel \Sigma_{semaphore}^n \parallel \Sigma_{\pi_1} \parallel \dots \parallel \Sigma_{\pi_n} \parallel \Sigma_{\varphi_1} \parallel \dots \parallel \Sigma_{\varphi_m} \parallel \Sigma_{o_1}^n \parallel \dots \parallel \Sigma_{o_p}^n \parallel \Sigma_{g_1} \parallel \dots \parallel \Sigma_{g_k}$ .

Here,  $\Sigma_{M_0}^n$  is the extension of  $\Sigma_{M_0}^{n-1}$  with a transition on the *?resume<sub>n</sub>* action.  $\Sigma_{semaphore}^n$  is the extension of  $\Sigma_{semaphore}^{n-1}$  with a state  $s_n$  and transitions to and from this state. For every domain object  $o$ ,  $\Sigma_o^n$  is the extension of  $\Sigma_o^{n-1}$  with new transitions: on actions corresponding to activities in  $M_n^a$  and on trigger actions introduced by  $\varphi_n$  and conditions in  $M_n^a$ . Since STSs are added or extended in step  $n$ , every run of  $\Sigma_{n-1}^+$  is also a run of  $\Sigma_n^+$ .

**( $\Rightarrow$ ) for every execution of  $M_n$  there is a corresponding run of  $\Sigma_n^+$ .**

We assume that for every execution of  $M_{n-1}$  there is a corresponding run of  $\Sigma_{n-1}^+$ .  $M_n$  is computed such that  $Exec(M_n) = Exec(M_{n-1}, C_n)$ . From Lemma 1, we know that  $Exec(M_{n-1}) \subseteq Exec(M_{n-1}, C_n)$ . Since runs of  $\Sigma_{n-1}^+$  are also runs of  $\Sigma_n^+$ , we have that for every execution of  $M_n$  which is also an execution of  $M_{n-1}$  there is a corresponding run of  $\Sigma_n^+$ . (1)

Let  $\omega = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k$  be an execution of  $M_n$  which is not an execution of  $M_{n-1}$ . Then there exists  $i, 1 \leq i \leq k-1$ , such that:

- $\langle a_1, \dots, a_i \rangle$  is a trace of  $M_{n-1}$ , and  $a_i$  is the label of *from<sub>n</sub>*;
- there exists  $\gamma'_i$  directly reachable from  $\gamma_i$  which satisfies  $\varphi_n$ ;
- $\gamma'_i \xrightarrow{a_{i+1}} \gamma_{i+1} \dots \gamma_{j-1} \xrightarrow{a_j} \gamma_j$  is an execution of  $M_n^a$ , with  $i < j < k$ ;
- finally,  $a_{j+1}$  is the activity label of *to<sub>n</sub>* in  $M_0$ .

The prefix of  $\omega$  up to step  $i$ ,  $\omega' = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{i-1} \xrightarrow{a_i} \gamma_i$ , is an execution of  $M_{n-1}$ . Therefore, there exists a run  $\sigma' = s_0, action_1, \dots, action_m, s_m$  of  $\Sigma_n^+$  which corresponds to  $\omega'$ . State  $s_m$  corresponds to  $\gamma_i$ . We need to prove that there exists a run of  $\Sigma_n^+$   $\sigma$  for which  $\sigma'$  is a prefix and which corresponds to  $\omega$ .

Up to step  $j+1$ , the run  $\sigma$  is constructed in the same way as for the proof of Lemma 6, for the strict case. First, the condition  $\varphi_n$  is triggered



if necessary, followed by transitions corresponding to the execution on  $M_n^a$ . Eventually, the control is given back to  $\Sigma_{M_0}^n$ , which moves to state  $s_{to}$ , with  $\Sigma_{semaphore}^n$  moving to  $s_0$ . Next, the action corresponding to  $a_{j+1}$  is applied, and the state reached corresponds to configuration  $\gamma_{j+1}$ .

Different to the strict case, from this point on the execution  $\omega$  does not necessarily continue on  $M_0$ , and may repeatedly go through adaptations. This can be matched in  $\sigma$ , since all the condition and adaptation STSs have been reset to their initial state, and can be triggered again. Therefore,  $\sigma$  corresponds to  $\omega$ . (2)

From (1) and (2), we have that for every execution of  $M_n$  there is a corresponding run of  $\Sigma_n^+$ .

**(<=) for every run of  $\Sigma_n^+$  there is a corresponding execution of  $M_n$ .** We assume that for every run of  $\Sigma_{n-1}^+$  there is a corresponding execution of  $M_{n-1}$ . We know that every run of  $\Sigma_{n-1}^+$  is a run of  $\Sigma_n^+$ . Moreover, from Lemma 1 we know that  $Exec(M_{n-1}) \subseteq Exec(M_n)$ , and therefore every execution of  $M_{n-1}$  is also an execution of  $M_n$ . Then, for every run of  $\Sigma_n^+$  which is also a run of  $\Sigma_{n-1}^+$  there is a corresponding execution of  $M_n$ . (1)

Let  $\sigma$  be a run of  $\Sigma_n^+$  which is not a run of  $\Sigma_{n-1}^+$ . Then  $\sigma$  must be such that at some point transitions in  $\Sigma_{\Delta_n}$  and (if it exists) in  $\Sigma_{\varphi_n}$  were triggered. For transitions in  $\Sigma_{\Delta_n}$  and  $\Sigma_{\varphi_n}$  to be possible, one of  $\Sigma_{M_0}, \Sigma_{\Delta_1}, \dots, \Sigma_{\Delta_{n-1}}$  must reach a state labeled with  $point_n$ , and at the same time  $\Sigma_{\pi_n}$  must be in a state labeled with  $trace_n$  (if the correction  $C_n$  is *relaxed*,  $\Sigma_{\pi_n}$  contains only this state).

Then  $\sigma = s_0, action_1, \dots, action_i, s_i, \dots$ , where the prefix of  $\sigma$  up to step  $i$  is a run  $\sigma'$  of  $\Sigma_{n-1}^+$ ,  $action_i$  corresponds to the activity label of node  $from_n$ , and  $s_i, \mathcal{F} \models point_n \wedge trace_n$ . The execution  $\omega' = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k$  of  $M_n$  which corresponds to  $\sigma'$  is such that  $a_k = l(from_n)$  and, if  $C_n$  is *strict*, it matches the trace  $\pi_n = \langle a_1, \dots, a_k \rangle$ .

(a) If  $s_i, \mathcal{F} \models \varphi_n$ , there are two possibilities. If  $\varphi_n$  does not include

uncontrollable literals, then  $\Sigma_{\varphi_n}$  has not been created, and the initial transition in  $\Sigma_{\Delta_n}$  is enabled from  $s_i$ . Otherwise, if  $\varphi_n$  includes uncontrollable literals, then  $\Sigma_{\varphi_n}$  has been created and the action in  $\sigma$  which follows  $s_i$  is the *!no-trigger<sub>n</sub>* action in  $\Sigma_{\varphi_n}$ . The next state  $s'_i$  has the same labels as  $s_i$ , since only  $\Sigma_{\varphi_n}$  moves on *!no-trigger<sub>n</sub>*. Configuration  $\gamma_k$  corresponds to  $s_i$  and in the second case also to  $s'_i$ . Therefore also  $\gamma_k \models \varphi_n$ .

(b) If  $s_i, \mathcal{F} \not\models \varphi_n$ , then since transitions in  $\Sigma_{\Delta_n}$  are triggered in  $\sigma$ , this can happen only if  $\varphi_n$  contains uncontrollable literals. The action in  $\sigma$  which follows  $s_i$  is the *!trigger<sub>n</sub>* action in  $\Sigma_{\varphi_n}$ . The relevant domain object STSs move on *!trigger<sub>n</sub>*, and the next state  $s'_i$  is such that  $s'_i, \mathcal{F} \models \varphi_n$ . In this case,  $\gamma_k$  corresponds to  $s_i$  and there exists a configuration  $\gamma'_k$  directly reachable from  $\gamma_k$  which corresponds to  $s'_i$ , such that  $\gamma'_k \models \varphi_n$ .

After state  $s_i$ , respectively  $s'_i$ , there may also be transitions in the goal STSs, however these only influence the state of their own STS. Eventually, in the run  $\sigma$  the first *action<sub>i+1</sub>* which does not trigger a transition in a goal STS corresponds to the initial transition in  $\Sigma_{\Delta_n}$ . Since both  $\Sigma_{\Delta_n}$  and  $\Sigma_{semaphore}^n$  move, next state  $s_{i+1}$  is such that  $s_{i+1}, \mathcal{F} \models flag_n$ . Therefore, until a transition on *?resume<sub>n</sub>* is reached in  $\Sigma_{\Delta_n}$ , the next transitions in  $\sigma$  are either controlled by  $\Sigma_{\Delta_n}$  or by the goal STSs (which affect only their state). Although transitions in the domain objects STSs and conditions STSs may be triggered as well, these are controlled by the state of  $\Sigma_{\Delta_n}$  through guards. Therefore, for the run  $\sigma'' = s_0, action_1, \dots, action_j, s_j$ , where  $action_j = ?resume_n$ , there is a corresponding execution of  $M_n$ ,  $\omega'' = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k \dots \gamma_{l-1} \xrightarrow{a_l} \gamma_l$ , where  $\gamma_k \xrightarrow{a_{k+1}} \gamma_{k+1} \dots \gamma_{l-1} \xrightarrow{a_l} \gamma_l$  is an execution of  $M_n^a$ .

Since  $\Sigma_{\Delta_n}$ ,  $\Sigma_{M_0}$ , and  $\Sigma_{semaphore}$  move on *?resume<sub>n</sub>*, state  $s_j$  is such that  $s_j, \mathcal{F} \models flag_0$  and  $\Sigma_{M_0}$  is in a state  $s_{to_n}$  from which there is a transition on *action<sub>to\_n</sub>* corresponding to the activity label  $a$  of  $to_n$ . If *action<sub>to\_n</sub>* can be fired in  $s_j$ , then  $s_j, \mathcal{F} \models pre_a$ . Therefore also  $\gamma_l \models pre_a$ ,  $a$  is the next

action in the execution corresponding to  $\sigma$ , and  $s_{j+1}$  corresponds to  $\gamma_{l+1}$ .

From  $s_{j+1}$  and until the end of the run  $\sigma$ , the STSs that can move are: the goal STSs,  $\Sigma_{M_0}$ , any  $\Sigma_{\Delta_i}$  and  $\Sigma_{\varphi_i}$  for which the correction  $C_i$  is relaxed, and controlled by  $\Sigma_{M_0}$  and  $\Sigma_{\Delta_i}$ , the domain object STSs and condition STSs. The transitions in the goal STSs influence only their own state, and will not be reflected in the execution. The proof that for the sequence in the run  $\sigma$  from  $s_{j+1}$  until the last state  $s_p$  ( $s_{j+1}, action_{j+2}, \dots, action_p, s_p$ ) there is a corresponding sequence in the execution  $\omega$ ,  $\gamma_{l+1} \xrightarrow{a_{l+2}} \gamma_{l+2} \dots \gamma_{m-1} \xrightarrow{a_m} \gamma_m$  goes in a similar way. If  $\Sigma_{M_0}$  moves on actions corresponding to activities, these activities will appear in  $\omega$ . If a plug-in point for a relaxed correction  $C_i$  is reached, then the corresponding condition  $\varphi_i$  may be re-triggered and the transitions in the adaptation STS  $\Sigma_{\Delta_i}$  are fired. In the execution  $\omega$ , an execution of the adaptation model  $M_i^a$  is started. If the plug in point for the next relaxed correction  $C_j$  is reached, condition  $\varphi_j$  may be re-triggered and the control is given to  $\Sigma_{\Delta_j}$ . In  $\omega$  the execution of  $M_i^a$  is interrupted and an execution of  $M_j^a$  starts. Therefore, also for a run of  $\Sigma_n^+$  which is not a run of  $\Sigma_{n-1}$  there is a corresponding execution of  $M_n$ . (2)

From (1) and (2), we have that for every run of  $\Sigma_n^+$  there is a corresponding execution of  $M_n$ .  $\square$

### 7.2.2 Correspondence Between Complete Executions of a Corrected Model and Complete Runs of the Controller

#### Lemma 9

*Assume a corrective evolution problem defined by a process model  $M_0$ , a goal  $G$ , and a sequence of corrections  $C_1, \dots, C_n$ , such that  $\forall i, 1 \leq i \leq n$ ,  $C_i = \langle ct, \pi_i, \varphi_i, \Delta_i \rangle$ , and  $ct \in \{strict, relaxed\}$ . For all  $i, 1 \leq i \leq n$ ,  $M_i$  is computed such that  $Exec(M_i) = Exec(M_{i-1}, C_i)$ . Assume  $M_n$  satisfies the goal  $G$ , and let  $\Sigma_c^n$  be the generated controller. Then for every complete execution of  $M_n$  there is a corresponding complete run of  $\Sigma_c^n$ , and for every*

complete run of  $\Sigma_c^n$  there is a corresponding complete execution of  $M_n$ .

*Proof.* ( $=>$ ) **for every complete execution of  $M_n$  there is a corresponding complete run of  $\Sigma_c^n$ .** From Lemma 8, we know that for every execution of  $M_n$  there is a corresponding run of  $\Sigma_n^+$ .

The runs of the controlled system  $\Sigma_c^n \triangleright \Sigma_n^+$  are a subset of the runs of  $\Sigma_n^+$ .  $\Sigma_c^n$  is constructed such that transitions on input actions are triggered only to reach goal states, while transitions on output actions are considered uncontrollable and are always triggered. Therefore, the runs of  $\Sigma_c^n \triangleright \Sigma_n^+$  are runs of  $\Sigma_n^+$  for which the transitions in the goal STSs are triggered as soon as they are enabled.

Let  $\omega = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k$  be a complete execution of  $M_n$  and  $\sigma$  the corresponding run of  $\Sigma_n^+$ , in which the transitions in the goal STSs are triggered as soon as they are enabled. We need to prove that  $\sigma$  is a complete run of  $\Sigma_c^n \triangleright \Sigma_n^+$ , or in other words that  $\sigma$  is such that the last state is an accepting state.

Since  $\omega$  is a complete execution of  $M_n$ , the last activity  $a_k$  corresponds to an end node in  $M_0$ . Therefore in  $\sigma$  the state  $s_p$  corresponding to  $\gamma_k$  is such that  $\Sigma_{M_0}$  is in the accepting state following the transition on the action corresponding to  $a_k$ . Further, the semaphore, adaptation, and condition STSs are in their initial state, which is also accepting. The domain object STSs and trace STSs have only accepting states. We need to prove that either the goal STSs are in an accepting state, and therefore  $s_p$  is an accepting state of  $\Sigma_n^+$ , or that from  $s_p$  the next actions in  $\sigma$  move the goal STSs to an accepting state.

Since  $M_n$  satisfies  $G$ , based on Definition 8, the following holds for every goal statement  $\psi_0 \implies (\psi_1 \succ \dots \succ \psi_m)$  in  $G$ . If  $\exists i, 0 \leq i \leq k$ , such that  $\gamma_i \models \psi_0$  or  $\gamma'_i \models \psi_0$ , where  $\gamma'_i$  is a configuration directly reachable from  $\gamma_i$  such that  $a_{i+1}$  is applicable in  $\gamma'_i$  and  $\gamma_{i+1}$  is reached by applying  $a_{i+1}$  to

$\gamma'_i$ , then  $\gamma_k$  satisfies at least one of  $\psi_1, \dots, \psi_m$ .

For each configuration  $\gamma_i$  in  $\omega$  there is a corresponding state  $s_j$  in  $\sigma$ . If the execution passes a configuration  $\gamma'_i$  directly reachable from  $\gamma_i$ , then there is a state  $s_l$  in  $\sigma$  corresponding to  $\gamma'_i$ , such that between  $s_j$  and  $s_l$  the transitions are triggered only in the condition STSs (and controlled by the condition STSs, in the domain object STSs), or in the goal STSs.

Therefore, if  $\gamma_i$  or  $\gamma'_i$  satisfies  $\psi_0$ , then also the corresponding state  $s$  in  $\sigma$  (either  $s_j$  or  $s_l$ ) is such that  $s, \mathcal{F} \models \psi_0$ . Then the action  $!a_{\psi_0}$  follows  $s$  in  $\sigma$ , and the goal statement STS  $\Sigma_g$  moves to a non-accepting state. From the non-accepting state,  $\Sigma_g$  may move repeatedly to one of the accepting states  $s_1, \dots, s_m$  and then back to the non-accepting state. We need to prove that the last state in  $\sigma$  is such that  $\Sigma_g$  is in one of the accepting states  $s_1, \dots, s_m$ . We recall that  $s_p$  is the state in  $\sigma$  corresponding to  $\gamma_k$ . Since  $\gamma_k$  satisfies at least one  $\psi_j$ ,  $1 \leq j \leq m$ , also  $s_p, \mathcal{F} \models \psi_j$ . Then, if the goal statement STS  $\Sigma_g$  is still in the non-accepting state, the action  $!a_{\psi_j}$  is triggered in  $s_p$ , and  $\Sigma_g$  moves to the accepting state  $s_j$ .

We now need to prove that if  $\psi_0$  is not satisfied in any  $\gamma_i$  or directly reachable  $\gamma'_i$ ,  $0 \leq i \leq k$ , then also  $\Sigma_g$  does not move from  $s_0$ . Since  $\gamma_i \not\models \psi_0$ , then also the corresponding state  $s$  in  $\sigma$  is such that  $s, \mathcal{F} \not\models \psi_0$  and  $\Sigma_g$  does not move. Let  $s_l$  be the state in  $\sigma$  corresponding to  $\gamma_{i+1}$ . Then between  $s$  and  $s_{l-1}$ , the only STSs that can move are the condition STSs and, controlled by the condition STSs, the domain object STSs. If any condition STS moves on a *!no-trigger* action, the domain object STSs do not move, and the resulting state  $s'$  is such that  $s', \mathcal{F} \not\models \psi_0$ . If any condition STSs moves on a *!trigger*, this can happen because of a condition between the nodes corresponding to activity  $a_i$  and  $a_{i+1}$ , or because of a condition from a correction. Therefore the resulting state  $s'$  corresponds to the configuration  $\gamma'_i$  directly reachable from  $\gamma_i$  which is such that  $a_{i+1}$  is applicable in  $\gamma'_i$ . Since  $\gamma'_i \not\models \psi_0$ , also  $s', \mathcal{F} \not\models \psi_0$ , and  $\Sigma_g$  does not move

from  $s_0$ .

Since  $\sigma$  reaches an accepting state which is also a goal state,  $\sigma$  is a complete run of  $\Sigma_c^n \triangleright \Sigma_n^+$ . In  $\Sigma_c^n$  there exists one complete run  $\sigma'$  matching  $\sigma$  (only the names of states change). Then, for every complete execution  $\omega$  of  $M_n$  there exists a corresponding complete run  $\sigma'$  of  $\Sigma_c^n$ .

**( $\Rightarrow$ ) For every complete run of  $\Sigma_c^n$ , there is a corresponding complete execution of  $M_n$ .** Every complete run  $\sigma$  of  $\Sigma_c^n$  matches a complete run  $\sigma'$  of  $\Sigma_c^n \triangleright \Sigma_n^+$ . The run  $\sigma'$  is a complete run of  $\Sigma_n^+$  for which the last state is a goal state. Since every run of  $\Sigma_n^+$  corresponds to an execution of  $M_n$  (Lemma 8), we have that also every complete run of  $\Sigma_c^n$  corresponds to a execution of  $M_n$ . We now need to prove that this execution is complete.

Since  $\sigma'$  is complete, the last state  $s_i$  in  $\sigma'$  is such that  $\Sigma_{M_0}$  is in an accepting state. In  $\Sigma_{M_0}$  there cannot be transitions on resume actions leading to this accepting state (the resume actions always lead to the start state of the transitions corresponding to the *to* nodes). The accepting state can then be reached only by applying the action corresponding to an end node activity in  $M_0$ . Therefore there exists *action<sub>j</sub>* preceding state  $s_i$  in  $\sigma'$ , which corresponds to an end node activity in  $M_0$ . From state  $s_{j+1}$ , the only STSs that can move are the goal STSs. Therefore, *action<sub>j</sub>* is the last action in  $\sigma'$  corresponding to an activity. In the corresponding execution of  $M_n$ , the last activity corresponds to an end node, and the execution is complete.  $\square$

### 7.2.3 Correctness

#### Theorem 3 (Correctness)

*Assume a corrective evolution problem defined by a process model  $M_0$ , a goal  $G$ , and a sequence of corrections  $C_1, \dots, C_n$ , such that  $\forall i, 1 \leq i \leq n$ ,  $C_i = \langle ct_i, \pi_i, \varphi_i, \Delta_i \rangle$ ,  $ct_i \in \{\text{strict}, \text{relaxed}\}$ . Let  $M_{relaxed}$  be the translation of  $\Sigma_{relaxed}$ .  $M_{relaxed}$  is a solution for the corrective evolution problem defined*

by  $M_0$ ,  $G$ , and  $C_1, \dots, C_n$ .

*Proof.* To prove the correctness of the approach, we have to prove that  $M_{relaxed} \equiv M_n$ , where  $\forall i, 1 \leq i \leq n$ ,  $M_i$  is computed such that  $Exec(M_i) = Exec(M_{i-1}, C_i)$ , and  $M_n$  satisfies  $G$ .

$M_{relaxed}$  is the translation of  $\Sigma_{relaxed}$ , and  $\Sigma_{relaxed}$  is the minimization of the controller  $\Sigma_c$ . Based on Lemma 9, we know that there is a correspondence between complete executions of  $M_n$  and complete runs of  $\Sigma_c$ .

$\Sigma_{relaxed}$  has less states than  $\Sigma_c$ , but the same set of complete traces as  $\Sigma_c$ . The condition actions from  $\Sigma_{M_0}, \Sigma_{\Delta_1}, \dots, \Sigma_{\Delta_n}$  are present in  $\Sigma_c$ , and therefore also in  $\Sigma_{relaxed}$ . The reason is that  $\Sigma_c$  is such that for every complete run of  $\Sigma_c$ ,  $\Sigma_{M_0}, \Sigma_{\Delta_1}, \dots, \Sigma_{\Delta_n}$  must all reach an accepting state, and the accepting states are only the initial and final states.

$M_{relaxed}$  is translated from  $\Sigma_{relaxed}$  such that there exists a one-to-one correspondence between complete traces of  $M_{relaxed}$  and complete traces of  $\Sigma_{relaxed}$ . Actions in  $\Sigma_{relaxed}$  are mapped into activities, which are copied to  $M_{relaxed}$  with their preconditions and effects from  $M_0, M_1^a, \dots, M_n^a$ . The condition actions in  $\Sigma_{relaxed}$  are replaced in  $M_{relaxed}$  with the branch conditions from  $M_0, M_1^a, \dots, M_n^a$ . For each correction  $C_i$ , we add branch conditions on  $\varphi_i$  before each application of the first activity in the adaptation  $\Delta_i$ .

We know that there exists a one-to-one correspondence between complete traces of  $M_{relaxed}$  and complete traces of  $\Sigma_{relaxed}$ , and  $\Sigma_{relaxed}$  has the same complete traces as  $\Sigma_c$ . Further, with Lemma 8, we know that there is a one-to-one correspondence between complete traces of  $M_n$  and complete traces of  $\Sigma_c$ . Therefore  $M_{relaxed}$  has the same complete traces as  $M_n$ , i.e.,  $Traces(M_{relaxed}) = Traces(M_n)$ . Moreover, if for some complete trace  $\pi$  of  $M_n$ , there is a condition appearing between consecutive activities  $a$  and  $a'$ , then this condition appears also in  $M_{relaxed}$  for the complete trace  $\pi$

between  $a$  and  $a'$ . Since  $M_{relaxed}$  has the same complete traces as  $M_n$ , and also the same conditions and preconditions, then  $M_{relaxed}$  will allow the same executions as  $M_n$ .  $\square$

#### 7.2.4 Completeness

##### Theorem 4 (Completeness)

*Assume a corrective evolution problem defined by a process model  $M_0$ , a goal  $G$ , and a sequence of corrections  $C_1, \dots, C_n$ , such that  $\forall i, 1 \leq i \leq n$ ,  $C_i = \langle ct_i, \pi_i, \varphi_i, \Delta_i \rangle$ ,  $ct_i \in \{strict, relaxed\}$ . Then the relaxed corrective evolution approach always terminates on the problem defined by  $M_0$ ,  $G$ , and  $C_1, \dots, C_n$ . If the approach does not return a solution  $M_{relaxed}$ , then no solution for this problem exists.*

*Proof.* Termination of the approach follows directly from the termination of the planning approach proved in [9].

We prove that the approach always returns a solution if a solution exists by contradiction. We assume that there exists a solution to the problem  $M_n$ , and the relaxed corrective evolution approach does not return a solution. This can happen only if the planning algorithm returned no controller  $\Sigma_c$ . This means that there exists an initial state in  $\Sigma^+$  for which there is no strong plan to reach a goal state. In other words, in  $\Sigma^+$  there exists an initial state for which there is no complete run leading to a recoverable goal state. Let  $s_0$  be this initial state.

With Lemma 8, we know that for every run of  $\Sigma^+$  there exists a corresponding execution of  $M_n$ . Therefore, each of the complete runs starting from  $s_0$  which do not lead to a recoverable goal state will have a corresponding execution of  $M_n$ . Since the corresponding runs do not lead to a recoverable goal state, also these executions will not satisfy the goal. We need to prove that at least one of these executions is complete.



For the state  $s_0$  in  $\Sigma^+$  there is exactly one corresponding initial configuration of  $M_n$ ,  $\gamma_0$ . For every initial configuration  $\gamma$  of  $M_n$ , if there is at least one execution from  $\gamma$ , then at least one such execution from  $\gamma$  is complete (otherwise, the process deadlocks). Assume  $\omega = \gamma_0 \xrightarrow{a_1} \gamma_1 \dots \gamma_{k-1} \xrightarrow{a_k} \gamma_k$  is one such complete execution from  $\gamma_0$ . With Lemma 8, there exists a run  $\sigma$  in  $\Sigma^+$  which corresponds to  $\omega$ . If  $\sigma$  is incomplete, then it does not reach a goal state (goal states correspond to accepting states in the goal STSs). Otherwise, if  $\sigma$  is complete, then it is one of the complete runs from  $s_0$  which does not lead to a goal state. Since no goal state is reached in the corresponding run, then also  $\omega$  does not satisfy  $G$ . Since  $\omega$  is a complete execution of  $M_n$ , this means that also  $M_n$  does not satisfy the goal  $G$ , leading to a contradiction.

Therefore, if a solution to the problem exists, then also the relaxed corrective evolution approach returns a solution.  $\square$

### 7.3 Discussion

Given a corrective evolution problem for which the corrections are either strict or relaxed, if a solution to the problem exists, it will be found using the relaxed corrective evolution approach described in this Chapter. However, it can be the case that a solution for the problem does not exist, or in other words, it can be the case that none of the process models obtained by applying the sequence of corrections satisfies the goal of the original process model.

To deal with this situation, we can combine the strict and relaxed corrective evolution approaches in the following way. We can first apply all the corrections as strict; with strict corrective evolution, we will obtain one evolved process model. We can then switch various corrections from strict to relaxed, considering all the combinations, and apply the relaxed

corrective evolution approach to the resulting problems. This way, we are guaranteed to find at least one evolved process model. If more than one evolved process models have been created, we can rank them according to intrinsic properties, such as size or complexity. We can also rank them based on the past performance of the adapted process instances that they represent.

In the following Chapter, we will perform exactly such a comparison between evolved process models. In particular, we will measure how much the evolved process models deviate behaviorally and structurally from the original process model, and how well they represent the adapted process instances.

# Chapter 8

## Evaluation

We implemented the solutions presented in Chapter 6 and Chapter 7 into a prototype tool. For strict corrective evolution, our tool generates the parallel product of the STSs using the NuSMV <sup>1</sup> model checker. For relaxed corrective evolution, the automated planning is realized using WSYNTH, one of the tools from the ASTRO <sup>2</sup> toolset. For both strict and relaxed corrective evolution, our tool minimizes STSs using one of the tools from the mCRL2 toolset <sup>3</sup>, called `ltsconvert`.

Using our prototype tool, we conducted a series of experiments with the aim of showing the tradeoffs between applying strict and relaxed corrections. As starting point for our experiments, we have chosen the event log from the 2012 edition of the BPI Challenge <sup>4</sup>.

In the following, we first shortly describe this log. We then present the common setup of our experiments, followed by each experiment. We close the chapter with a discussion, where we draw some general conclusions based on the results obtained.

---

<sup>1</sup><http://nusmv.fbk.eu>

<sup>2</sup><http://www.astroproject.org>

<sup>3</sup><http://www.mcrl2.org>

<sup>4</sup><http://www.win.tue.nl/bpi2012/doku.php?id=challenge>

## 8.1 Overview of the Event Log

The event log proposed in the 2012 BPI Challenge is a real-life log taken from a Dutch financial institute. The log represents data recorded over approximately six months, from October 2011 to March 2012. In total, the log contains 262.200 events in 13.087 traces. The events contain information regarding the resource processing the event, the timestamp, the name of the process activity, as well as the stage in the lifecycle of the activity (schedule, start, complete).

The traces in the log correspond to instances of a single process, an application process for a personal loan or overdraft. We shortly describe this process. A process instance starts when a customer submits a loan or overdraft application. This application first goes through some automatic checks. For applications which are considered suspicious, a further check for fraud may be performed. If these checks are successful, the customer is contacted by phone in order to obtain additional information. If the application is eligible, an offer is sent to the customer by mail. When the response to the offer is received, this response is assessed, and the customer may be contacted again for missing information. The application then goes through a final assessment, after which it is either approved, declined, or cancelled.

Without the support of log analysis and process mining tools, this log is an overwhelming mass of data. The complexity of the log does not come from the number of activities involved, the number of distinct activities being 36, a rather small number. For comparison, the number of distinct activities in the real-life log from the 2011 BPI Challenge, which concerned a patient treatment process, is 624. Instead, the complexity of the log seems to stem from a great variation in how the applications are processed, in terms of the order of executing the process activities, as well as the number

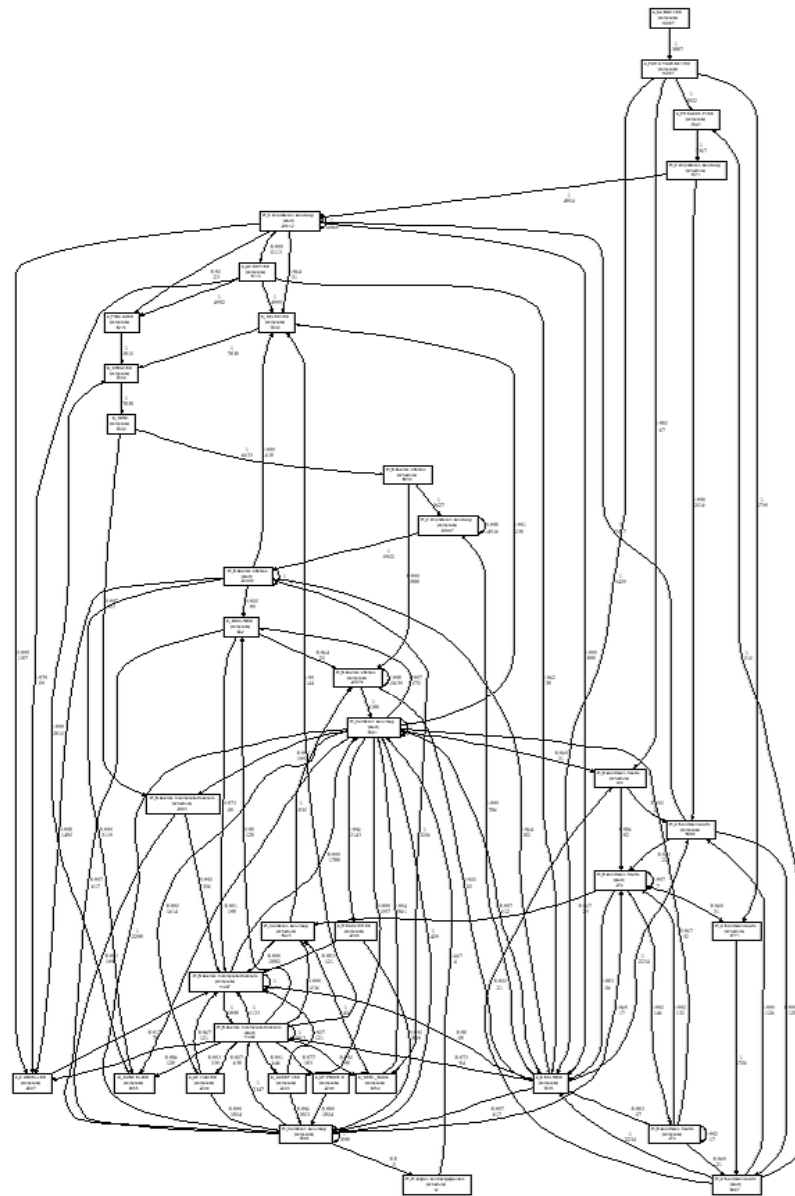


Figure 8.1: Mining result on the raw event log

of times certain execution sequences are re-executed.

To give an idea of this complexity, we used one of the basic process mining tools from the ProM framework <sup>5</sup>, namely the Heuristic Miner [119], to directly mine the log, without any filtering or pre-processing. The resulting heuristics net is shown in Figure 8.1. This result also shows that

<sup>5</sup><http://www.promtools.org>

a naive approach to mining the log is not sufficient, since mining the raw log yields an incomprehensible process model, a model that is not suitable to be analyzed by humans.

## 8.2 Experimental Setup

In this section, we describe the setup of our experiments. Since we started from an execution log, we needed to recreate all the other elements required by our approach: the domain objects, the original process model and its goal, as well as the corrections to the process model.

To make the BPI Challenge execution log compatible with our approach, we have made several assumptions. The basic assumption was that there exists a process model which corresponds to the log, such that the traces recorded in the log correspond to regular or adapted instances of this process model. Moreover, this process model does not fit perfectly to the log, or in other words at least one of the traces in the log corresponds to an adapted process instance. In general this is not necessarily the case, since the traces may correspond to a process model which fits perfectly to the log.

Our two other assumptions are related to the adaptation assumptions described in Section 3.4: that adaptation occurs only in case a design-time constraint is violated, and that the purpose of adaptation is to reach the original goal of the process model. We have used these two last assumptions when designing our elements, in particular the domain objects, the conditions on control edges, and the activity preconditions.

### 8.2.1 Designing the Domain Objects

We designed our domain objects based on the event log and the textual descriptions given as input in the challenge. In total, we created seven

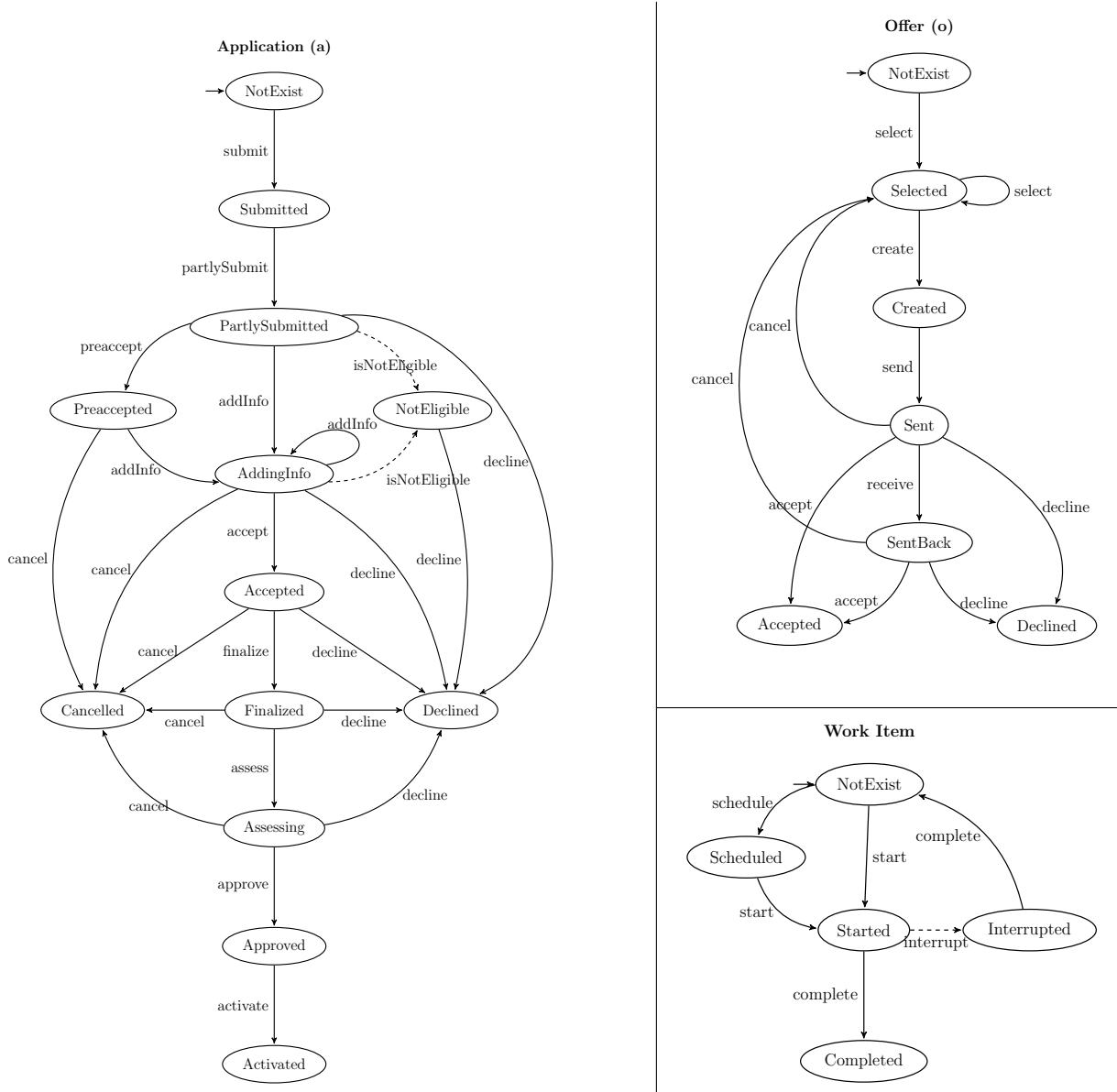


Figure 8.2: Domain objects for the loan application process

domain objects. Two of these domain objects, *Application* and *Offer*, are shown directly in Figure 8.2. The third diagram in Figure 8.2, which we called a *Work Item*, stands in fact for five domain objects which have different names, but have the same states, events, and transitions.

The Work Item domain objects represent phases in the execution of long running tasks. Their role is to restrict the order in which activities

triggering these phases can occur. To clarify the purpose of these work items in processing the loan application, we list here their names and the English description (we used the original names in Dutch):

- 'Afhandelen leads' - fixing incoming lead,
- 'Completeren aanvraag' - filling in information for the application,
- 'Valideren aanvraag' - assessing the application,
- 'Nabellen offertes' - calling after sent offers,
- 'Nabellen incomplete dossiers' - calling to add missing information to the application.

The Application and Offer domain objects correspond to artifacts created during the execution of the loan application process. The Application is a domain object created as soon as the client requests a loan, and in which all the necessary information about the client and the loan request is accumulated during the execution of the process. After the final assessment, the Application can be either approved and activated, declined, or cancelled.

The Offer object is created only in case the loan application is eligible. The domain object allows multiple iterations to be performed, in which the offer is selected, created, sent to the client, and received back. As soon as the current offer is cancelled, a new offer can be created and sent to the client. Eventually, the offer is either accepted or declined.

We designed the domain objects in two steps. To obtain the application object, we first filtered the event log to include only the activities related to application status. In the log, these are events having the name prefixed with "A\_". We mined the filtered log using the Transition System Miner and obtained an STS representing the lifecycle of the application, as it appears in the event log. We proceeded in a similar way with the offer and



work items, and filtered the log to include only activities related to offer status (name prefixed with “O\_”), respectively work item status. We then mined the filtered logs to obtain the lifecycle of the offer, respectively work item.

In the second step, we used the logs and the textual description given as input in the challenge to modify these STSs. For example, the textual description of applications states that after some initial automatic checks are performed, the application is complemented with additional information, after which a decision whether the application is eligible is taken. We therefore added in the Application domain object an intermediary state *AddingInfo*, from which the Application can move to state *Accepted*, *Cancelled*, *Declined*, or *NotEligible*. Since information can be added repeatedly, the object stays in *AddingInfo* as long as the event *addInfo* is triggered. To encode that the decision that the application is not eligible is taken outside the process, we added a state *NotEligible*, to which all incoming transitions are on the uncontrollable event *isNotEligible*. From state *NotEligible*, the application can only move to state *Declined*.

A similar example is that of the state *Assessing*. In this case, we know from the description that there is a final assessment of the application, before being approved and activated. We therefore added the intermediary state *Assessing* to the Application, from which the object can move to *Approved*, *Cancelled*, or *Declined*.

### 8.2.2 Annotating the Activities

We used the domain objects to annotate all the activities that appear in the log with preconditions and effects. For annotating the activities we used the activity names, as well as the textual description given as input in the challenge.

For example, we know from the textual description that the application

is complemented with additional information after passing the automatic check. The activity *W\_Completeren aanvraag SCHEDULE* (scheduling the filling in of information for the application) can then happen only if the Application object is in state *Preaccepted*. Therefore, the precondition for this activity is the formula:

$NotExist^s(Completeren\_aanvraag) \wedge Preaccepted^s(Application)$ ,

and the effect is the event:  $schedule^e(Completeren\_aanvraag)$ .

The next activity for this work item, *W\_Completeren aanvraag START*, is responsible for triggering the *addInfo* event in the Application object, and thereby moving the object in the state *AddingInfo*, if the object is not already in this state. We therefore annotated this activity with the precondition:  $Scheduled^s(Completeren\_aanvraag) \wedge (Preaccepted^s(Application) \vee AddingInfo^s(Application))$  and the effects:

$\{start^e(Completeren\_aanvraag), addInfo^e(Application)\}$

The last activity for this work item, *W\_Completeren aanvraag COMPLETE*, should be triggered if the corresponding work item object is in the state *Started* or *Interrupted* (the latter is a state reachable only through an uncontrollable event). While this activity triggers in both cases the same event *complete*, note that if the work item object is in state *Started*, it will move to state *Completed*, while if it is in the state *Interrupted*, it will be reset to the initial state. The precondition of this activity is the formula:  $Started^s(Completeren\_aanvraag) \vee Interrupted^s(Completeren\_aanvraag)$ , and the effect is:  $complete^e(Completeren\_aanvraag)$ .

### 8.2.3 Obtaining the Original Process Model

To create a process model which could be considered as the first representation of the application, we used both the event log and the description of the challenge. It is important to note that the process model that we were searching for did not have to represent all the behavior in the event log.

In fact, we knew that if this process model contained less behavior, then we would be able to generate more evolutionary corrections. However, the process model had to reflect some of the most frequent behavior in the log. Otherwise, a correction referring to very frequent behavior would lead to a process model which fits to the log to a much greater extent than a similar correction referring to less frequent behavior.

From the challenge description, we know that by the end of any completed process instance of the loan application process, a decision is taken whether to approve, decline, or cancel the application. We therefore created our original process model by first obtaining a process model corresponding to frequent traces in the log for which the application was eventually approved. We then used our mechanism for detecting frequent differences between the process model and the log, which we describe in Section 8.2.4, to correct this process model and represent the most frequent traces in which the application is declined, respectively cancelled.

To obtain the process model for approved applications, we proceeded as follows. Using a log filter, we noted that the most frequent end event for traces corresponding to successful applications is the event *W\_Valideren aanvraag COMPLETE*. We therefore considered only these traces, and applied a second filter, the process instance frequency filter, in order to consider only traces which appear multiple times. A threshold of 4 for this filter was sufficient to obtain, by mining the filtered log, a simple enough process model. We further simplified this model by removing the activities concerning application declining and cancelling, as well as the loops.

The reason why we first needed to filter the log to obtain this process model, instead of creating the process model directly from the most frequent traces, is related to an interesting property of the log. The traces in the log are such that applications are much more often declined than approved or cancelled. For example, the event *A\_DECLINED COM-*

*PLETE* occurs 7635 times in the log (2.9% of the events), while the event *A\_APPROVED COMPLETE* occurs only 2246 times (0.8%), and *A\_CANCELLED COMPLETE* 2807 times (1%). Therefore, mining the most frequent traces results in a process model which does not contain the behavior in which the application is approved, which is, in some sense, the successful behavior of the process.

We corrected the process model obtained for successful applications to include the most frequent behavior for declining, respectively cancelling the application. The process model we obtained, and which we used in all the experiments, is shown in Figure 8.3. Compared to the process model shown in Figure 8.1, our rough process model is much easier to follow, and, although it does not represent by far all the behavior in the log, from the way it was constructed we know it represents some of the most frequent behavior in this log.

After having created the process model, we defined a goal for this model based on the domain objects in Figure 8.2. Using the goal, we represented the fact that a state when the application is activated, declined, or cancelled must eventually be reached, and that having the application activated is preferred to having it declined or cancelled:

$$\top \implies \text{Activated}^s(\text{Application}) \succ \\ \text{Declined}^s(\text{Application}) \vee \text{Cancelled}^s(\text{Application})$$

#### 8.2.4 Generating Corrections

To generate corrections, we computed the most frequent differences between our rough process model and the traces in the log. We shortly explain our strategy for computing these differences. We started by trying to replay every trace in the log on the process model. In the moment where we could no longer replay a trace on the process model, we started recording the difference. We continuously tried to match each following

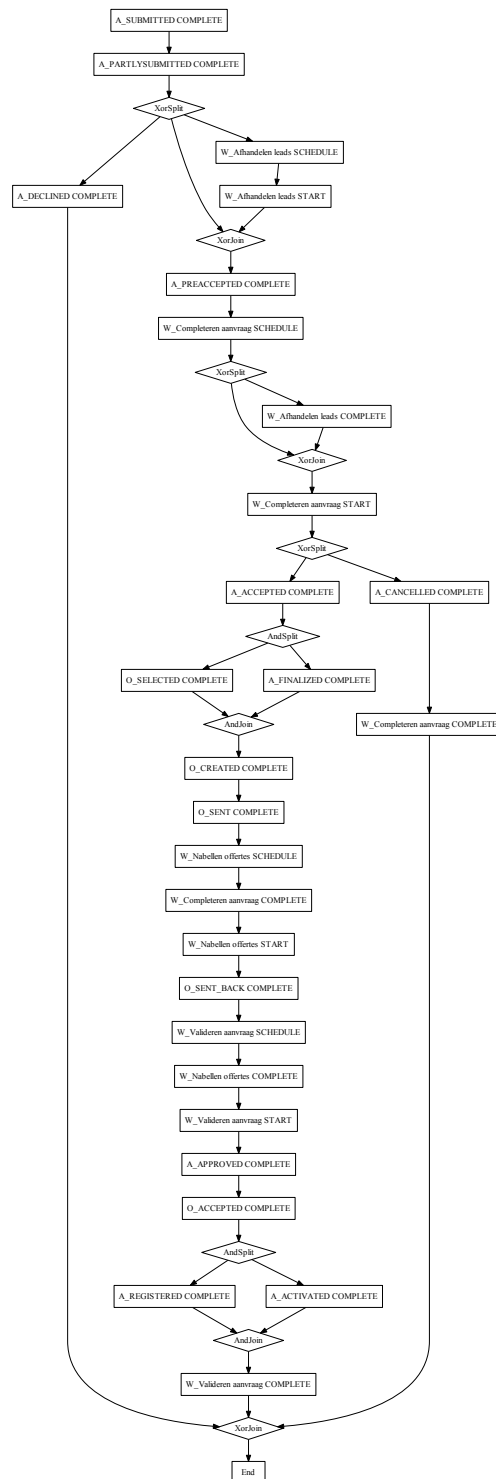


Figure 8.3: A rough process model

activity in the trace to the activity nodes in our process model. If such an activity was found, we completed the recording of the difference. This difference would then contain the partial trace which could be replayed on the process model, the node in the process model where the deviation started, the sequence of activities which could not be matched, and finally the node in the process model matching the activity in the trace. Note that for a single trace multiple differences to the process model could be recorded.

When matching activities in the trace to the activity nodes in the process model, we used the following assumption. We assumed that there is a preference order in applying adaptation strategies, with the most preferred strategy being the inclusion of a process fragment (the *from* and *to* nodes of the adaptation are consecutive nodes), followed by the strategy of including a fragment and performing a forward jump, and finally the strategy of including a fragment and performing a backward jump. Therefore, we first tried to match the activity with the nodes which follow the last matched node, continuously increasing the distance from this last matched node. If unsuccessful, we then tried to match the activity with nodes appearing before the last matched node.

After collecting all the differences, we computed their frequencies, and used the most frequent differences to create our corrections. Since each difference includes a *from* node and a *to* node from our original process model, as well as a sequence of activities which could not be replayed on this model, the adaptation can be generated in a straightforward way. To generate the condition, we assumed that an adaptation must have been performed if the next activity in the model could not be executed, and combined the negation of the precondition of this next activity with the precondition of the first unmatched activity. Then, to create a strict correction, we used the partial trace in the difference, the condition, and the

adaptation. To create the corresponding relaxed correction, we used only the condition and the adaptation.

### 8.3 Evaluating Tradeoffs Between Strict and Relaxed Corrections

We evaluated the tradeoffs between applying strict and relaxed corrections by comparing the resulting process models with the original process model along three dimensions:

- *fitness* - how much of the behavior in the log is captured by the corrected process models;
- *precision* - how much extra behavior is allowed by the corrected process models when compared to the original model;
- *structure* - how much the corrected process models deviate structurally from the original model.

We used several metrics devised for evaluating process mining results, which are implemented in the ProM framework. In the following, we describe describe in turn each metric used and the results obtained.

#### 8.3.1 Fitness

To measure fitness, we used the  $f$  metric introduced in [90], implemented in ProM as *token-based fitness*. This is a fine-grained metric, which quantifies the extent to which the traces in the log can be replayed on the process model, punishing problems caused by activities that are not activated or that remain activated.

We were interested not only to compare the two types of corrections, but also to evaluate the fitness of corrected process models over time. For

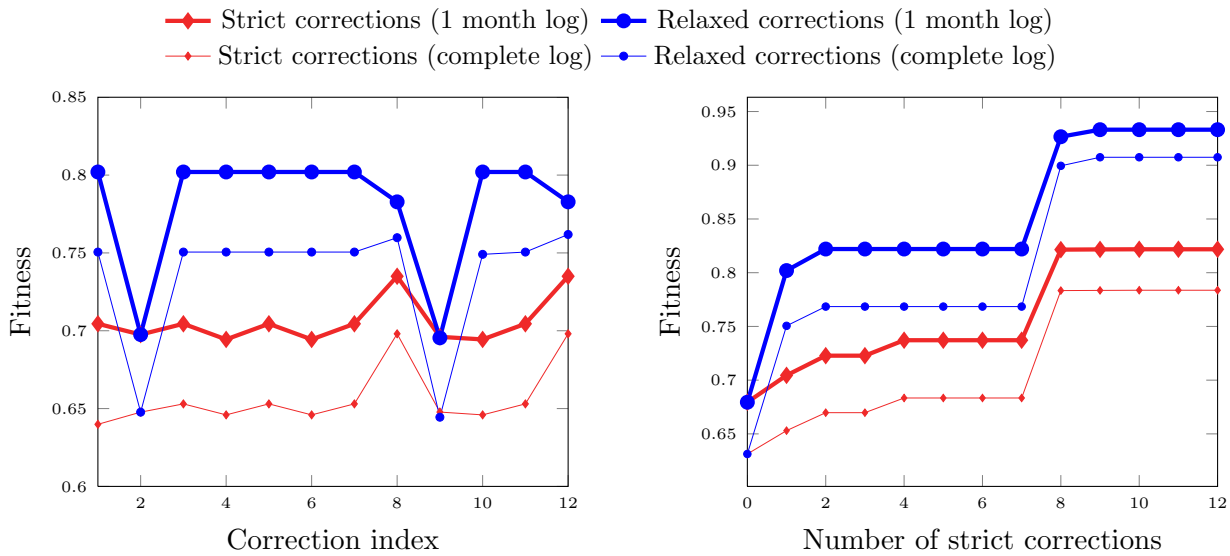


Figure 8.4: Fitness: corrections applied (1) individually; (2) incrementally

this purpose, we used only a fragment of the entire log (roughly one sixth, corresponding to the first month) to generate our corrections. This way, we simulate the situation when the process model is evolved after having been executed for one month, based on the adapted process instances collected during this month. We then measured the fitness of the corrected process models obtained by applying strict, respectively relaxed, corrections on the one-month log fragment, as well as on the entire log. The results are shown in Figure 8.4.

We measured the fitness of the corrected models on the one-month log fragment in order to test if applying corrections increases the fitness of process models, and also which of the two correction types leads to a higher fitness. By measuring the fitness of the corrected models also on the entire log, we simulate the passing of time, and therefore test if the corrections applied based on the adaptations performed in the first month are still relevant over the entire six months period.

Figure 8.4(1) shows the fitness of corrected process models when corrections were applied individually, i.e., at each step we applied exactly one



correction on the original process model. Figure 8.4(2) shows the fitness when corrections were applied incrementally. At step 0, we measured the fitness of the original process model. Then, in the strict version, at step  $n$  we corrected the original process model with  $n$  strict corrections. In the relaxed version, at step  $n$  we corrected the original model with  $m \leq n$  relaxed corrections. The number of strict and relaxed corrections is not necessarily equal, since several strict corrections may correspond to the same relaxed correction. As can be seen in Figure 8.4(2), the fitness of process models increases when applying corrections, for both correction types. However, the fitness is higher for relaxed corrections, and it remains higher also when tested against the entire log.

### 8.3.2 Behavior

To measure changes in behavior, we used the *behavioral precision*  $B_P$  metric introduced in [22].  $B_P$  quantifies how much extra behavior a process model allows with respect to a reference process model and an event log.  $B_P$  is lower when the deviation in behavior is higher. Traces in the log are weighted by frequency, such that the more frequent a deviating behavior is, the lower the value obtained for  $B_P$ .

Similar to the previous measurements, we measured  $B_P$  of corrected process models when the corrections are applied both individually and incrementally. For the event log, we used the one-month log fragment. The results are shown in Figure 8.5. As expected, we observe that strict corrections introduce less behavior than relaxed corrections.

As it can be seen from Figure 8.5(1), the highest difference between the values measured for process models when corrections were applied individually occurs for the corrections 4 and 10. The reason for this difference is that in both cases the relaxed correction introduces a loop in the process model. In Figure 8.6 we show the process models obtained by applying cor-

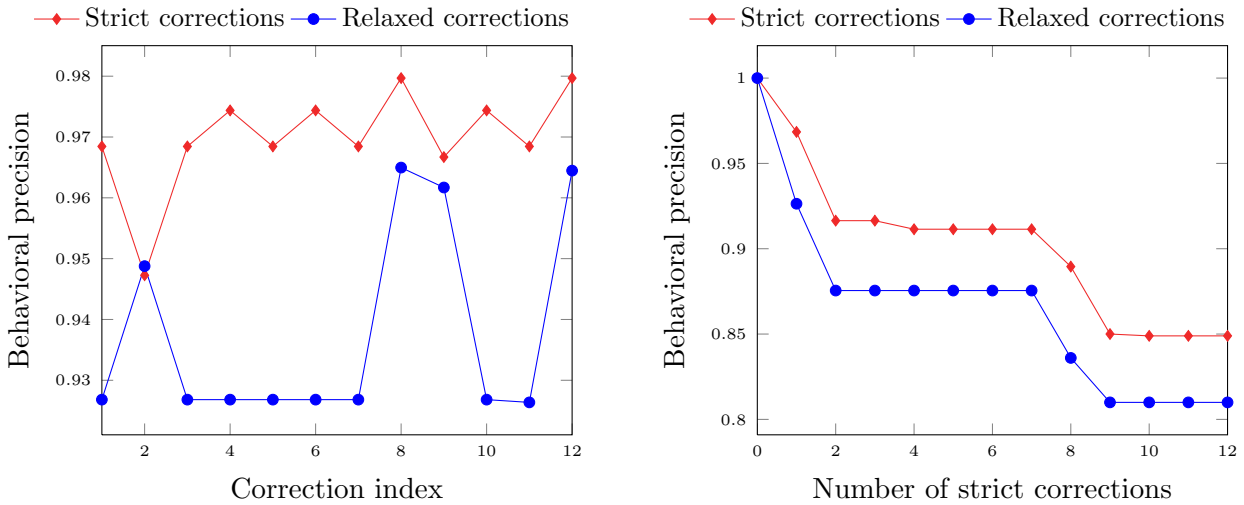


Figure 8.5: Behavioral precision: corrections applied (1) individually; (2) incrementally  
 rection 4 as a strict, respectively as a relaxed correction. In both process  
 models, we highlighted the fragment which is introduced by the correction.

### 8.3.3 Structure

To measure the deviation in structure, we used the *structural precision*  $S_P$  metric introduced in [22], which assesses how many connections a process model has that are not in a reference process model. Similar to the behavioral precision metric,  $S_P$  is lower when the deviation is higher. We measured  $S_P$  of corrected process models when the corrections are applied individually and incrementally. The results are shown in Figure 8.7. We observe that relaxed corrections lead to smaller structural changes than the corresponding strict corrections.

An interesting case is that of corrections 8 and 12, which lead to the lowest values for the strict corrections in Figure 8.7(1). For each of these strict corrections, the partial trace to the plug-in point passes through the first And-block in the process model in Figure 8.3. In order to apply the correction, the process model had to be unfolded up to the plug-in point.

### 8.3. EVALUATING TRADEOFFS BETWEEN STRICT AND RELAXED CORRECTIONS

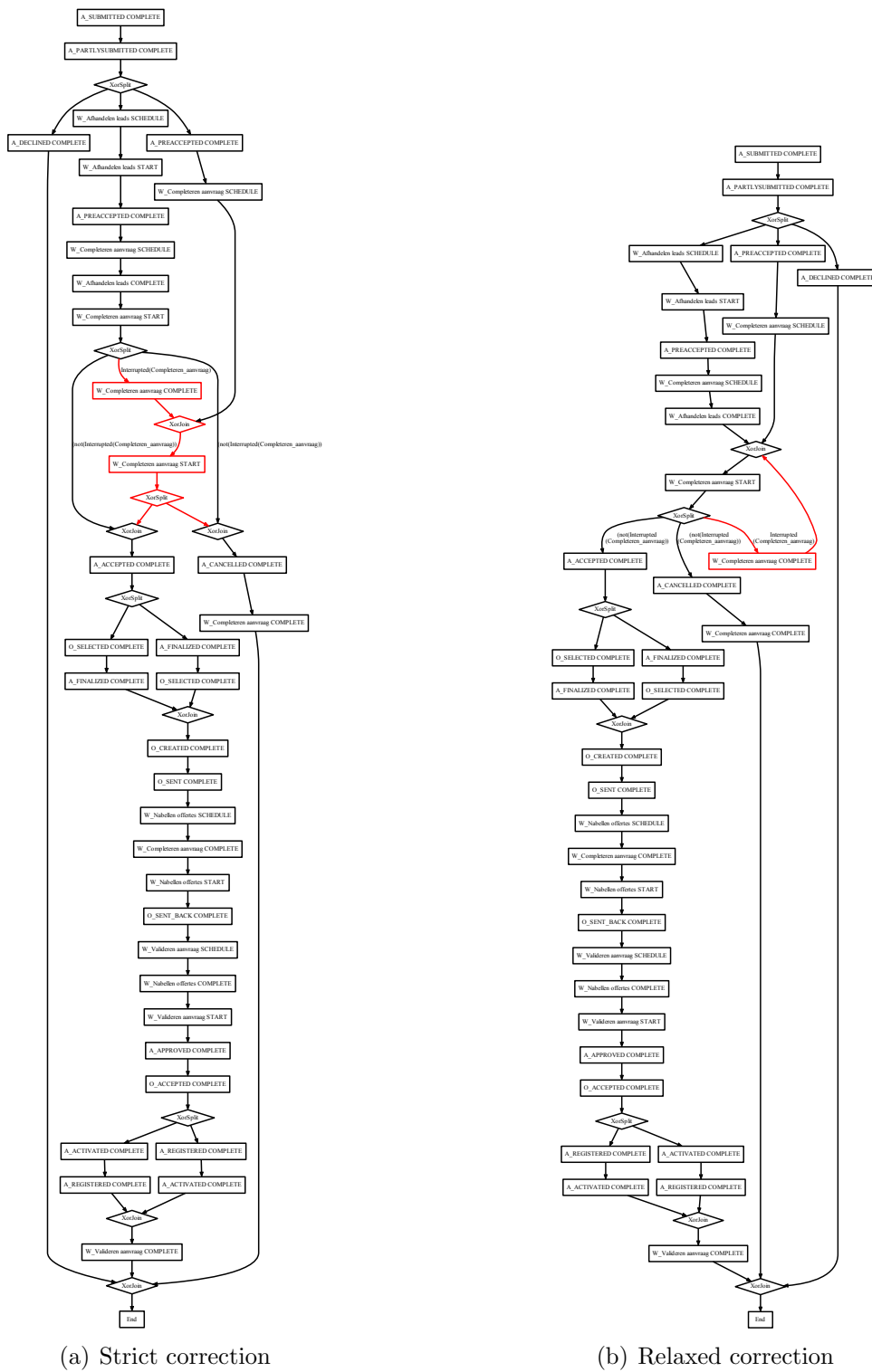


Figure 8.6: Corrected process models for the 4th correction

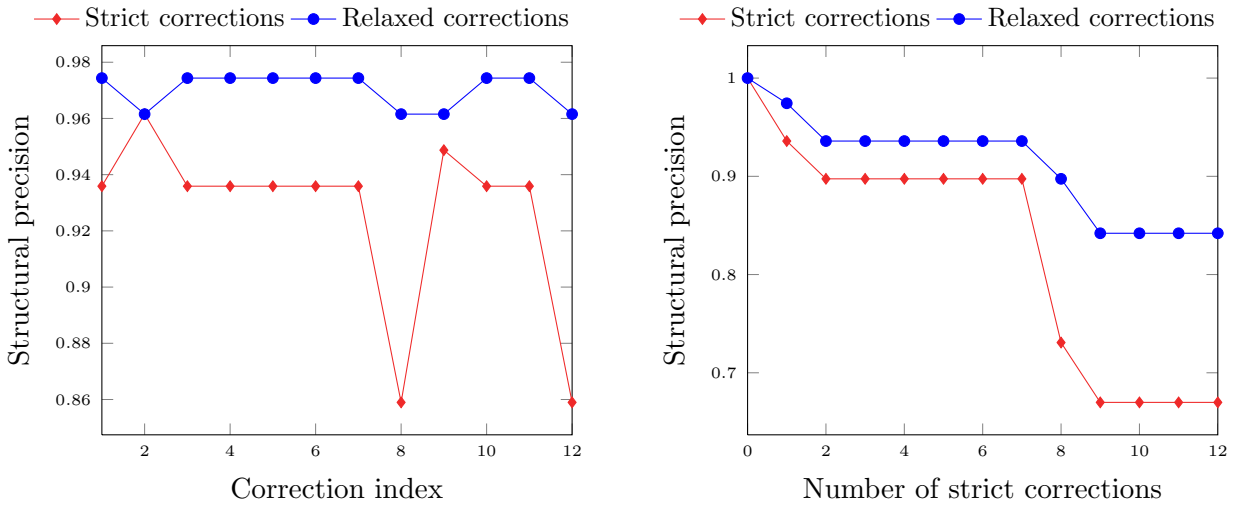


Figure 8.7: Structural precision: corrections applied (1) individually; (2) incrementally

No such unfolding was necessary for the corresponding relaxed corrections.

To give an idea of the difference between the process models which result by applying correction 8 as a strict, and respectively relaxed correction, we included a picture of these process models in Figure 8.8. Both process models contain more activity nodes than the original process model, due to the corrections, and also to the fact that the And-blocks have been unfolded. However, while the process model which results by applying the relaxed correction has in total 36 activity nodes, the process model for the strict correction has in total 45.

## 8.4 Discussion

Based on the results we obtained in our experiments, we conclude that for this scenario there is a tradeoff between applying strict and relaxed corrections. Specifically, relaxed corrections introduce more behavior, but lead to process models which have a higher fitness and less structural changes than strict corrections. Therefore, these results provide a first confirmation that by choosing different correction types we can obtain process models which

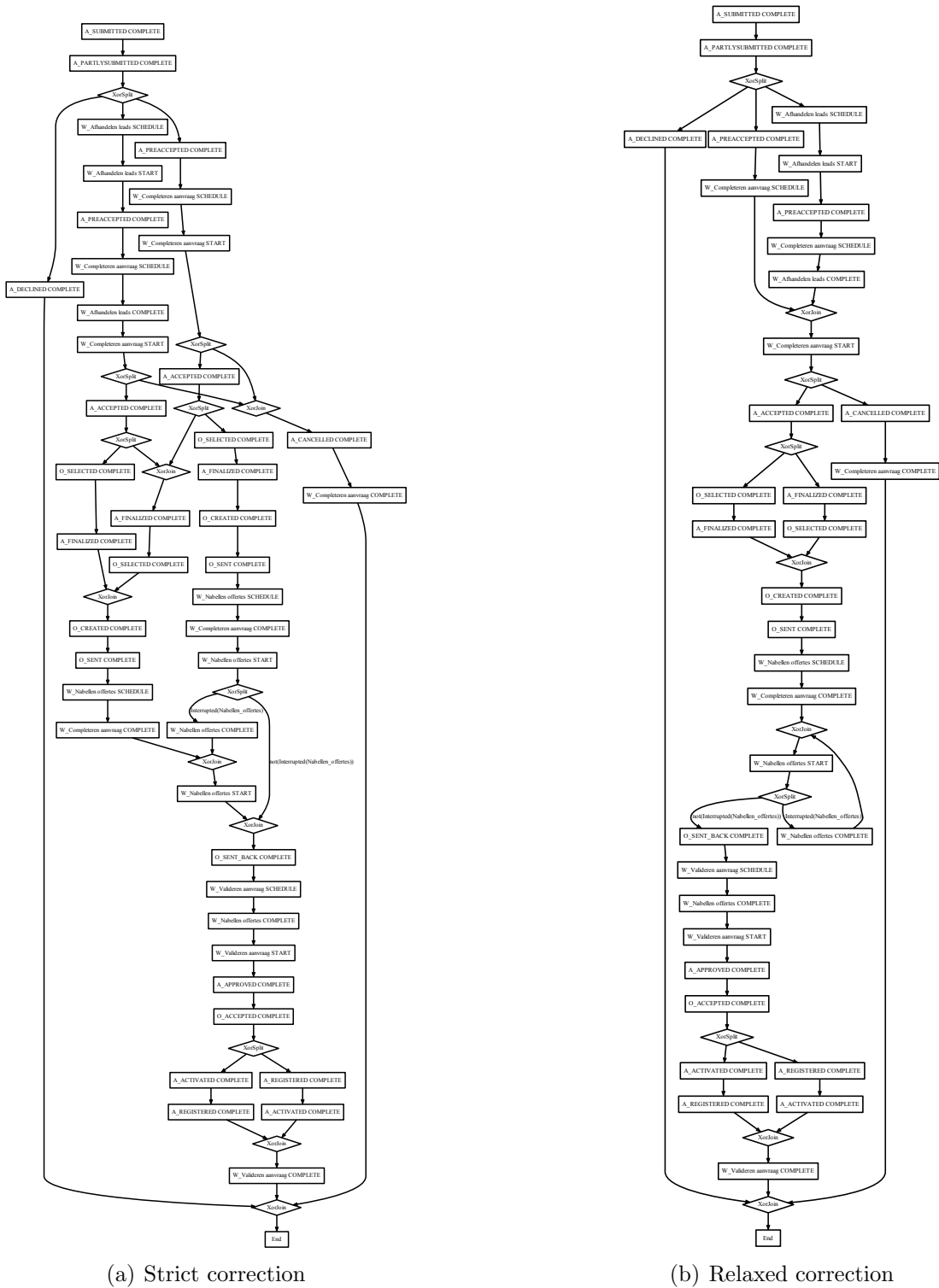


Figure 8.8: Corrected process models for the 8th correction

differ both in terms of behavior and in terms of structure. Moreover, since the fitness is higher for process models obtained with relaxed corrections, this is an indication that these process models would require less future adaptations than if strict corrections were applied.

In general, we expect relaxed corrections to introduce more behavior than the corresponding strict corrections as soon as there is more than one trace leading to the plug-in point. Moreover, relaxed corrections which realize backward jumps will introduce significantly more behavior than the strict corrections, since by applying these relaxed corrections we introduce loops in the process model.

In case there is more than one trace to the plug-in point, the relaxed corrections should also introduce less structural changes, since the strict corrections will require unfolding the process model up to this plug-in point, possibly duplicating many activity nodes.

Regarding fitness, a relaxed correction should be more effective than the corresponding strict correction if the process instance adaptation is commonly performed at the given point in the execution of the process model, independent of the trace followed up to that point.

# Chapter 9

## Conclusion

In this dissertation, we presented a new approach for process evolution. With process evolution we mean changing the process model in order to, for example, implement optimizations, accommodate legal or policy changes, improve the design of the process model.

We evolve process models based on the structural adaptations of process instances. Various approaches which use the structural adaptations of process instances to evolve process models or to support evolution already exist. Similar to our approach, they may use the logs of process instance adaptations, also known as change logs (e.g.,[87, 117, 105]). Other approaches support process evolution by considering the process variants that result by structural adaptation (e.g.,[37, 47, 53]). An aspect which is insufficiently explored in these approaches is how to ensure that the evolved process model achieves the same goal as the original model. Addressing this aspect is the main aim of the work presented in this dissertation.

### 9.1 Achieved Results

Process evolution is a rather late phase in the lifecycle of a process-based application, and requires the existence of a framework which supports also the modeling, execution, and adaptation of this application. We therefore

started by describing such a framework. Our process evolution framework allows to model an application as a collection of process models, together with the domain objects that these process models influence, and the goals that the models achieve. In the execution phase, the process models and corresponding domain objects are instantiated, and the process instances are executed. Existing runtime adaptation techniques such as [12, 21, 28] can be used for adapting the process instances. The execution and adaptation events are recorded into logs. In the analysis phase, these logs can be examined using existing analysis techniques such as [122, 105]. Adaptations which are successful according to performance indicators may be proposed for inclusion in the process model, thereby triggering evolution.

For evolving process models, we started from the idea that a process instance adaptation is tightly coupled to the context and trace for which it is used, and may even be harmful if used for different contexts or traces. When deciding to evolve the process model to include an instance adaptation, one straightforward possibility is to restrict the adaptation to the exact same context and trace as it was used in the process instance. We considered also the case when the adaptation should be inserted in the process model in a less restrictive way, allowing more freedom in selecting the context or the trace. We make sure that also in this case the adaptations that we introduce in the process model are not harmful by specifying the goals achieved by the original model, and verifying that these goals are achieved also by the evolved process model.

The main contributions of the research work presented in this dissertation are: (i) we presented a formal model which ensures that the evolved process model achieves the same goal as the original process model, (ii) we designed automated techniques for solving two increasingly general corrective evolution problems, (iii) we evaluated the tradeoffs between applying strict and relaxed corrections on a scenario built on a real event log.



**A formal model for process evolution which ensures goal compliance.** We presented a formal model for evolving process-based applications, which allows us to evolve a process model based on process instance adaptations, and at the same time ensure that the evolved process model continues to satisfy the goal of the original model.

To represent the process-based application, we modeled the domain knowledge using domain objects, the business logic using process models, and the relations between business logic and domain knowledge using process model annotations and goals. Based on this formal representation, we defined concepts related to the execution and adaptation of the application, as well as goal satisfaction criteria. These concepts were then used for defining evolutionary correction as the operation of changing a process model to include an instance adaptation at a certain point and for a certain condition. Finally, we defined the corrective evolution problem as the problem of applying a sequence of corrections, such that the resulting process model satisfies the original goal. By formulating the problem as the application of a sequence of corrections, rather than only one, we addressed a more general problem, since solving a corrective evolution problem may return more solutions than applying one correction at a time.

**Automated corrective evolution techniques.** We developed automated techniques for solving two cases of the corrective evolution problem: when all corrections are strict, and when the corrections are either strict or relaxed.

If all corrections are strict, a solution can be constructed using a naive approach. However, the resulting process model will contain many duplicated nodes. To find a solution with as few duplicated nodes as possible, we devised an automated approach based on state transition systems. If the original process model and the adaptation models do not contain parallelism, the process model obtained using this approach is the minimal

solution to the corrective evolution problem. Otherwise, the parallelism can be restored by applying post-processing techniques. Finally, we proved the correctness and completeness of the approach.

If corrections are either strict or relaxed, it is necessary not only to compose the process model and adaptations, but also to verify that the composition satisfies the goal. For this purpose, we devised an automated approach based on planning. The resulting process model includes all the corrections and is at the same time guaranteed to satisfy the goal of the original process model. Also in this case, we proved the correctness and completeness of the approach.

**Evaluation of tradeoffs between strict and relaxed corrections.** We implemented the two automated corrective evolution techniques into a prototype tool. Using this tool, we evaluated the tradeoffs between applying strict and relaxed corrections on a scenario built on a real event log. We compared the corrected process models with the original process model along three dimensions: *fitness* (how much of the behavior in the log is captured), *precision* (how much behavior is introduced), and *structure* (how many structural differences are introduced). For this scenario, we observed the following tradeoffs: relaxed corrections introduce more behavior, but lead to a higher fitness and less structural changes than strict corrections. These experimental results provide a first confirmation that the choice of correction type influences both the behavior and the structure of the evolved process model.

## 9.2 Future Directions

The work presented in this dissertation can be extended in several directions. The first and most important direction is to design and evaluate

solutions for the general case, when corrections can also be relaxed with conditions. If at least one correction is relaxed with conditions, the corrective evolution problem becomes a search problem. For each such correction, the partial traces on which the correction can be applied are not specified. We therefore need to search for the traces on which the adaptation can be applied, such that, by applying the other corrections, we can obtain a process model which satisfies the goal. Since there may be an infinite set of such partial traces, the challenge is to understand how to group these traces such that the number of tests to be performed is finite.

Designing a search strategy is difficult also because the set of traces on which a correction which is relaxed or relaxed with conditions can be applied is not fixed, and depends on the other corrections. This is due to the fact that the corrections introduce traces. The problem gets significantly more complex if there is more than one relaxed correction with conditions, due to the combinatorial explosion. We therefore need to design search techniques which can deal efficiently with this complexity, for example using heuristics.

A different extension concerns the specification of corrections, and the possibility to automatically derive suggestions for new corrections. The idea would be to integrate our approach with an approach able to analyze previous process instance executions and adaptations and derive process model changes. For example, the approach presented by Soffer et al. in [105] groups process instances based on similar contextual properties, traces, and outcomes, and can be used for answering questions about performance and for identifying successful adaptations. These successful adaptations, together with the traces and situations in which they can be applied, can then be used as corrections in our corrective evolution approach.

By integrating our process evolution approach with a log analysis ap-

proach, we will also have the possibility to evaluate our process evolution approach in a dynamic setting. The empirical evaluation discussed in this dissertation is static, in the sense that since we create and evolve the process model based on an execution log, we cannot measure how the evolved process model influences the performance of the application. If our approach is integrated with a log analysis approach, we can perform a different evaluation, using a running system. An interesting experiment would be to apply corrections to a process model used in a running system, allowing corrections to be either strict, relaxed, or relaxed with conditions. The idea would be to measure which process models require less adaptations and perform best in the long run. Regarding the number of adaptations, since relaxed corrections (with conditions) introduce more behavior, they should require less adaptations, but only if this behavior is actually used. Regarding performance, this new behavior introduced by relaxed corrections (with conditions) may be unknown, so we should measure how the overall performance is affected by the new behavior.

Finally, two other research directions for our process evolution approach concern extending the goal language, respectively the process modeling language. Regarding goals, we can extend the language to include domain object events, similar to the language for composition requirements in [10], respectively try-catch statements similar to the language proposed in [43]. Regarding process models, an interesting extension would be to consider a concrete process modeling language, such as BPMN [1] or WS-BPEL [2]. For both goal language and process modeling language extensions, we would have to extend our definitions of goal satisfaction, trace, and execution, as well as our encoding into state transition systems. We should then be able to apply the two corrective evolution techniques discussed in this dissertation to this extended setting without any further changes.

# Bibliography

- [1] Business Process Modeling Notation (BPMN) Specification, Final Adopted Specification. Technical report, Object Management Group (OMG), February 2006.
- [2] *OASIS: Web Services Business Process Execution Language Version 2.0, Organization for the Advancement of Structured Information Standards*, 2007.
- [3] M. Adams, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Worklets: A service-oriented implementation of dynamic flexibility in workflows. In *Proc. CoopIS 2006*, pages 291–308, 2006.
- [4] R. Agrawal, C. M. Johnson, J. Kiernan, and F. Leymann. Taming compliance with sarbanes-oxley internal controls using database technology. In *ICDE*, page 92, 2006.
- [5] A. Awad, G. Decker, and M. Weske. Efficient Compliance Checking Using BPMN-Q and Temporal Logic. In *BPM*, pages 326–341, 2008.
- [6] L. Baresi and S. Guinea. Self-supervising bpel processes. *IEEE Trans. Software Eng.*, 37(2):247–263, 2011.
- [7] L. Baresi, S. Guinea, and L. Pasquale. Self-healing BPEL processes with Dynamo and the JBoss rule engine. In *ESSPE 2007*, pages 11–20.

- [8] L. Baresi, A. Marconi, M. Pistore, and A. Sirbu. Corrective evolution of adaptable process models. In *BMMDS/EMMSAD*, volume 147 of *LNBIP*, pages 214–229. Springer, 2013.
- [9] P. Bertoli, R. Kazhamiakin, M. Paolucci, M. Pistore, H. Raik, and M. Wagner. Continuous orchestration of web services via planning. In *Proc. of the 19th Int. Conference on Automated Planning and Scheduling (ICAPS)*, 2009.
- [10] P. Bertoli, R. Kazhamiakin, M. Paolucci, M. Pistore, H. Raik, and M. Wagner. Control Flow Requirements for Automated Service Composition. In *Proc. ICWS'09*, pages 17–24, 2009.
- [11] P. Bertoli, M. Pistore, and P. Traverso. Automated composition of web services via planning in asynchronous domains. *Artificial Intelligence*, 174(3-4):316–361, 2010.
- [12] A. Bucchiarone, A. Marconi, M. Pistore, and H. Raik. Dynamic adaptation of fragment-based and context-aware business processes. In *Proc. ICWS'12*, pages 33–41, 2012.
- [13] A. Bucchiarone, A. Marconi, M. Pistore, and A. Sirbu. A context-aware framework for business processes evolution. In *EDOCW*, pages 146–154, 2011.
- [14] B. Burmeister, M. Arnold, F. Copaciu, and G. Rimassa. Bdi-agents for agile goal-oriented business processes. In *Proc. AAMAS 2008 (Industry Track)*, pages 37–44.
- [15] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow evolution. *Data & Knowledge Engineering*, 24(3):211–238, 1998.
- [16] F. Casati and M.-C. Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26(3):143–163, 2001.

- [17] M. Castellanos, F. Casati, M.-C. Shan, and U. Dayal. ibom: A platform for intelligent business operation management. In *ICDE*, pages 1084–1095, 2005.
- [18] C. Combi and M. Gambini. Flaws in the flow: The weakness of unstructured business process modeling languages dealing with data. In *OTM Conferences (1)*, pages 42–59, 2009.
- [19] C. Courbis and A. Finkelstein. Towards an Aspect Weaving BPEL Engine. In *Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2004.
- [20] T. H. Davenport. *Mission critical: realizing the promise of enterprise systems*. Harvard Business Press, 2000.
- [21] M. de Leoni, M. Mecella, and G. D. Giacomo. Highly dynamic adaptation in process management systems through execution monitoring. In *Proc. BPM'07*, pages 182–197.
- [22] A. K. A. de Medeiros, W. M. P. van der Aalst, and A. J. M. M. Weijters. Quantifying process equivalence based on observed behavior. *Data & Knowledge Engineering*, 64(1):55–74, 2008.
- [23] R. M. Dijkman, M. La Rosa, and H. A. Reijers. Managing large collections of business process models - current techniques and challenges. *Computers in Industry*, 63(2):91–97, 2012.
- [24] M. Dumas, L. García-Bañuelos, M. La Rosa, and R. Uba. Fast detection of exact clones in business process model repositories. *Information Systems*, 2012.
- [25] C. C. Ekanayake, M. L. Rosa, A. H. M. ter Hofstede, and M.-C. Fauvet. Fragment-based version management for repositories of business process models. In *Proc. CoopIS*, pages 20–37, 2011.

- [26] C. A. Ellis, K. Keddera, and G. Rozenberg. Dynamic change within workflow systems. In *COOCS*, pages 10–21, 1995.
- [27] D. Fahland and W. M. P. van der Aalst. Repairing process models to reflect reality. In *BPM*, pages 229–245, 2012.
- [28] G. Friedrich, M. Fugini, E. Mussi, B. Pernici, and G. Tagni. Exception handling for repair in service-based processes. *IEEE Transactions on Software Engineering*, 36(2):198–215, 2010.
- [29] C. Fritz, R. Hull, and J. Su. Automatic construction of simple artifact-based business processes. In *Proc. of the 12th Int. Conf. on Database Theory (ICDT)*, pages 225–238, 2009.
- [30] M. Gambini, M. L. Rosa, S. Migliorini, and A. H. M. ter Hofstede. Automated error correction of business process models. In *BPM*, pages 148–165, 2011.
- [31] A. Ghose and G. Koliadis. Auditing business process compliance. In *ICSOC*, volume 4749 of *LNCS*, pages 169–180. Springer, 2007.
- [32] M. Golani and A. Gal. Optimizing exception handling in workflows using process restructuring. In *Business Process Management*, pages 407–413, 2006.
- [33] F. Gottschalk, W. M. P. van der Aalst, M. H. Jansen-Vullers, and M. L. Rosa. Configurable workflow models. *Int. J. Cooperative Information Systems*, 17(2):177–221, 2008.
- [34] G. Governatori, Z. Milosevic, and S. W. Sadiq. Compliance checking between business processes and business contracts. In *EDOC*, pages 221–232. IEEE Computer Society, 2006.



- [35] C. W. Guenther, S. Rinderle-Ma, M. Reichert, W. M. van der Aalst, and J. Recker. Using process mining to learn from process changes in evolutionary systems. *Int'l J. of Business Process Integration and Management*, 3(1):61–78, 2007.
- [36] A. Hallerbach, T. Bauer, and M. Reichert. Context-based configuration of process variants. In *Proc. of the 3rd Int. Workshop on Technologies for Context-Aware Business Process Management (TCoB'08)*, pages 31–40, 2008.
- [37] A. Hallerbach, T. Bauer, and M. Reichert. Capturing variability in business process models: the provop approach. *Journal of Software Maintenance*, 22(6-7):519–546, 2010.
- [38] J. Hoffmann, P. Bertoli, and M. Pistore. Web service composition as planning, revisited: In between background theories and initial state uncertainty. In *AAAI*, pages 1013–1018, 2007.
- [39] J. Hoffmann, I. Weber, and G. Governatori. On compliance checking for clausal constraints in annotated process models. *Information Systems Frontiers*, 14(2):155–177, 2012.
- [40] R. Hull. Artifact-centric business process models: Brief survey of research results and challenges. In *OTM Conferences (2)*, pages 1152–1163, 2008.
- [41] P. C. Kanellakis and S. A. Smolka. Ccs expressions, finite state processes, and three problems of equivalence. In *PODC*, pages 228–240, 1983.
- [42] D. Karastoyanova and F. Leymann. BPEL'n'Aspects: Adapting Service Orchestration Logic. In *Proc. of the 2009 IEEE International Conference on Web Services (ICWS)*, pages 222–229. IEEE, 2009.

- [43] R. Kazhamiakin, M. Paolucci, M. Pistore, and H. Raik. Modelling and automated composition of user-centric services. In *OTM Conferences (1)*, pages 291–308, 2010.
- [44] M. Kradolfer and A. Geppert. Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In *CoopIS*, pages 104–114, 1999.
- [45] C. Krzysztof and E. Ulrich. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [46] M. Kunze, M. Weidlich, and M. Weske. Behavioral similarity - a proper metric. In *BPM*, pages 166–181, 2011.
- [47] J. M. Küster, C. Gerth, A. Förster, and G. Engels. Detecting and resolving process model differences in the absence of a change log. In *Proc. BPM'08*, pages 244–260, 2008.
- [48] J. M. Küster, K. Ryndina, and H. Gall. Generation of business process models for object life cycle compliance. In *BPM*, pages 165–181, 2007.
- [49] M. La Rosa, M. Dumas, R. Uba, and R. M. Dijkman. Merging business process models. In *OTM Conferences (1)*, pages 96–113, 2010.
- [50] M. La Rosa, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. Questionnaire-based variability modeling for system configuration. *Software and System Modeling*, 8(2):251–274, 2009.
- [51] R. Laue and J. Mendling. Structuredness and its significance for correctness of process models. *Inf. Syst. E-Business Management*, 8(3):287–307, 2010.

- [52] R. Lenz and M. Reichert. It support for healthcare processes - premises, challenges, perspectives. *Data & Knowledge Engineering*, 61(1):39–58, 2007.
- [53] C. Li, M. Reichert, and A. Wombacher. Discovering reference models by mining process variants using a heuristic approach. In *Proc. BPM'09*, pages 344–362, 2009.
- [54] Y. Liu, S. Müller, and K. Xu. A static compliance-checking framework for business process models. *IBM Systems Journal*, 46(2), 2007.
- [55] R. Lu and S. Sadiq. On the discovery of preferred work practice through business process variants. In *Proc. ER'07*, pages 165–180, 2007.
- [56] R. Lu and S. W. Sadiq. Managing process variants as an information resource. In *Proc. BPM'06*, pages 426–431, 2006.
- [57] L. T. Ly, S. Rinderle, and P. Dadam. Semantic correctness in adaptive process management systems. In *Business Process Management*, pages 193–208, 2006.
- [58] L. T. Ly, S. Rinderle, and P. Dadam. Integration and verification of semantic constraints in adaptive process management systems. *Data & Knowledge Engineering*, 64(1):3–23, 2008.
- [59] A. Marconi, M. Pistore, P. Poccianti, and P. Traverso. Automated Web Service Composition at Work: the Amazon/MPS Case Study. In *ICWS*, pages 767–774, 2007.
- [60] A. Marconi, M. Pistore, A. Sirbu, F. Leymann, H. Eberle, and T. Unger. Enabling Adaptation of Pervasive Flows: Built-in Contextual Adaptation. In *Proc. ICSOC 2009*, pages 445–454.

- [61] A. Marconi, M. Pistore, and P. Traverso. Automated Composition of Web Services: the ASTRO Approach. *IEEE Data Engineering Bulletin*, 31(3), 2008.
- [62] S. A. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In *KR*, pages 482–496. Morgan Kaufmann, 2002.
- [63] J. Mendling. *Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness*, volume 6 of *Lecture Notes in Business Information Processing*. Springer, 2008.
- [64] J. Mendling, H. A. Reijers, and W. M. P. van der Aalst. Seven process modeling guidelines (7pmg). *Information & Software Technology*, 52(2):127–136, 2010.
- [65] M. Minor, D. Schmalen, A. Koldehoff, and R. Bergmann. Structural adaptation of workflows supported by a suspension mechanism stand by case-based reasoning. In *Proceedings of WETICE 2007*, pages 370–375, 2007.
- [66] M. Minor, A. Tartakovski, and D. Schmalen. Agile workflow technology and case-based change reuse for long-term processes. *International Journal of Intelligent Information Technologies (IJIT)*, 4(1):80–98, 2008.
- [67] S. Modafferi, B. Benatallah, F. Casati, and B. Pernici. A methodology for designing and managing context-aware workflows. In *IFIP International conference MOBIS 2005*, pages 91–106, 2005.

- [68] D. Müller, M. Reichert, and J. Herbst. A new paradigm for the enactment and dynamic adaptation of data-driven process structures. In *CAiSE*, pages 48–63, 2008.
- [69] R. Müller, U. Greiner, and E. Rahm. AgentWork: a workflow system supporting rule-based workflow adaptation. *Data & Knowledge Engineering*, 51(2):223–256, 2004.
- [70] B. Mutschler, M. Reichert, and J. Bumiller. Unleashing the effectiveness of process-oriented information systems: Problem analysis, critical success factors, and implications. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 38(3):280–291, 2008.
- [71] E. D. Nitto, C. Ghezzi, A. Metzger, M. P. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Autom. Softw. Eng.*, 15(3-4):313–341, 2008.
- [72] OASIS WSBPEL Technical Committee. *Web Services Business Process Execution Language Version 2.0*, 21 2005. Committee Draft, work in progress.
- [73] C. Ouyang, M. Dumas, S. Breutel, and A. H. M. ter Hofstede. Translating standard process models to bpel. In *CAiSE*, pages 417–432, 2006.
- [74] C. Ouyang, M. Dumas, W. M. P. van der Aalst, A. H. M. ter Hofstede, and J. Mendling. From business process models to process-oriented software systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 19(1), 2009.
- [75] M. Pesic, M. H. Schonenberg, N. Sidorova, and W. M. P. van der Aalst. Constraint-based workflow models: Change made easy. In *OTM Conferences (1)*, pages 77–94, 2007.

- [76] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite BPEL4WS web services. In *Proc. ICWS 2005*, pages 293–301, 2005.
- [77] K. Ploesser, M. Peleg, P. Soffer, M. Rosemann, and J. Recker. Learning from context to improve business processes. *BPTrends*, 6(1):1–7, 2009.
- [78] A. Polyvyanyy, L. García-Bañuelos, and M. Dumas. Structuring acyclic process models. In *BPM*, pages 276–293, 2010.
- [79] F. Puhmann, A. Schnieders, J. Weiland, and M. Weske. Variability mechanisms for process models. PESOA-Report TR 17/2005, Hasso-Plattner-Institut, Potsdam, 2005.
- [80] H. Raik, A. Bucchiarone, N. Khurshid, A. Marconi, and M. Pistore. Astro-captevo: Dynamic context-aware adaptation for service-based systems. In *SERVICES*, 2012.
- [81] G. Regev, I. Bider, and A. Wegmann. Defining business process flexibility with the help of invariants. *Software Process: Improvement and Practice*, 12(1):65–79, 2007.
- [82] M. Reichert and P. Dadam. Adept<sub>flex</sub>-supporting dynamic changes of workflows without losing control. *J. Intell. Inf. Syst.*, 10(2):93–129, 1998.
- [83] M. Reichert, S. Rinderle-Ma, and P. Dadam. Flexibility in process-aware information systems. *T. Petri Nets and Other Models of Concurrency*, 2:115–135, 2009.
- [84] M. Reichert and B. Weber. *Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies*. Springer, 2012.

- [85] H. A. Reijers, R. S. Mans, and R. A. van der Toorn. Improved model management with aggregated business process models. *Data & Knowledge Engineering*, 68(2):221–243, 2009.
- [86] S. Rinderle, M. Reichert, and P. Dadam. Disjoint and overlapping process changes: Challenges, solutions, applications. In *CoopIS/DOA/ODBASE (1)*, pages 101–120, 2004.
- [87] S. Rinderle, B. Weber, M. Reichert, and W. Wild. Integrating process learning and process evolution - a semantics based approach. In *Proc. BPM'05*, pages 252–267, 2005.
- [88] M. Rosemann and W. M. P. van der Aalst. A configurable reference modelling language. *Information Systems*, 32(1):1–23, 2007.
- [89] A. Rozinat and W. M. P. van der Aalst. Decision mining in prom. In *Business Process Management*, pages 420–425, 2006.
- [90] A. Rozinat and W. M. P. van der Aalst. Conformance checking of processes based on monitoring real behavior. *Information Systems*, 33(1):64–95, 2008.
- [91] N. Russell, W. M. P. van der Aalst, and A. H. M. ter Hofstede. Workflow exception patterns. In *CAiSE*, volume 4001 of *LNCS*, pages 288–302. Springer, 2006.
- [92] N. Russell, W. M. P. van der Aalst, A. H. M. ter Hofstede, and P. Wohed. On the suitability of uml 2.0 activity diagrams for business process modelling. In *APCCM*, pages 95–104, 2006.
- [93] S. W. Sadiq, G. Governatori, and K. Namiri. Modeling control objectives for business process compliance. In *BPM*, pages 149–164, 2007.

- [94] S. W. Sadiq, M. E. Orłowska, and W. Sadiq. Specification and validation of process constraints for flexible workflows. *Information Systems*, 30(5):349–378, 2005.
- [95] W. Sadiq, O. Marjanovic, and M. E. Orłowska. Managing change and time in dynamic workflow processes. *Int. J. Cooperative Information Systems*, 9(1-2):93–116, 2000.
- [96] A. Schnieders and F. Puhmann. Variability mechanisms in e-business process families. In *BIS*, pages 583–601, 2006.
- [97] H. Schonenberg, R. Mans, N. Russell, N. Mulyar, and W. M. P. van der Aalst. Process flexibility: A survey of contemporary approaches. In *CIAO! / EOMAS 2008*, pages 16–30.
- [98] H. Schonenberg, B. Weber, B. F. van Dongen, and W. M. P. van der Aalst. Supporting flexible processes through recommendations based on history. In *Proc. BPM'08*, pages 51–66, 2008.
- [99] H. Schuschel and M. Weske. Integrated workflow planning and coordination. In *DEXA*, pages 771–781, 2003.
- [100] H. Schuschel and M. Weske. Triggering replanning in an integrated workflow planning and enactment system. In *ADBIS*, pages 322–335, 2004.
- [101] D. Shaparau. Complex Goals for Planning in Nondeterministic Domains: Preferences and Strategies. Ph.D. Dissertation, University of Trento, 2008.
- [102] A. Sirbu and J. Hoffmann. Towards scalable web service composition with partial matches. In *Proc. ICWS*, pages 29–36, 2008.



- [103] A. Sirbu, A. Marconi, M. Pistore, H. Eberle, F. Leymann, and T. Unger. Dynamic composition of pervasive process fragments. In *Proc. ICWS'11*, pages 73–80, 2011.
- [104] E. Sirin, B. Parsia, D. Wu, J. A. Hendler, and D. S. Nau. Htn planning for web service composition using shop2. *Journal of Web Semantics*, 1(4):377–396, 2004.
- [105] P. Soffer, J. Ghattas, and M. Peleg. A goal-based approach for learning in business processes. In *Intentional Perspectives on Information Systems Eng.*, pages 239–256. Springer, 2010.
- [106] W. M. P. van der Aalst and T. Basten. Inheritance of workflows: an approach to tackling problems related to change. *Theor. Comput. Sci.*, 270(1-2):125–203, 2002.
- [107] W. M. P. van der Aalst, M. Dumas, F. Gottschalk, A. H. M. ter Hofstede, M. L. Rosa, and J. Mendling. Correctness-preserving configuration of business process models. In *FASE*, pages 46–61, 2008.
- [108] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: yet another workflow language. *Information Systems*, 30(4):245–275, 2005.
- [109] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
- [110] W. M. P. van der Aalst, A. H. M. ter Hofstede, and M. Weske. Business process management: A survey. In *Business Process Management*, pages 1–12, 2003.
- [111] W. M. P. van der Aalst, M. Weske, and D. Grünbauer. Case handling: a new paradigm for business process support. *Data & Knowledge Engineering*, 53(2):129–162, 2005.

- [112] R. J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In J. C. M. Baeten and J. W. Klop, editors, *CONCUR*, volume 458 of *LNCS*, pages 278–297. Springer, 1990.
- [113] J. Vanhatalo, H. Völzer, and J. Koehler. The refined process structure tree. In *BPM*, pages 100–115, 2008.
- [114] J. Vanhatalo, H. Völzer, F. Leymann, and S. Moser. Automatic workflow graph refactoring and completion. In *ICSOC*, volume 5364 of *LNCS*, pages 100–115, 2008.
- [115] B. Weber, M. Reichert, J. Mendling, and H. A. Reijers. Refactoring large process model repositories. *Computers in Industry*, 62(5):467–486, 2011.
- [116] B. Weber, M. Reichert, and S. Rinderle-Ma. Change patterns and change support features - enhancing flexibility in process-aware information systems. *Data & Knowledge Engineering*, 66(3):438–466, 2008.
- [117] B. Weber, M. Reichert, S. Rinderle-Ma, and W. Wild. Providing integrated life cycle support in process-aware information systems. *Int. J. Cooperative Information Systems*, 18(1):115–165, 2009.
- [118] B. Weber, S. W. Sadiq, and M. Reichert. Beyond rigidity - dynamic process lifecycle support. *Computer Science - R&D*, 23(2):47–65, 2009.
- [119] A. Weijters and W. M. van der Aalst. Rediscovering workflow models from event-based data using little thumb. *Integrated Computer Aided Engineering*, 10(2):151–162, 2003.

- [120] M. Weske. Workflow management systems: Formal foundation, conceptual design, implementation aspects. *Habilitationsschrift Fachbereich Mathematik und Informatik*, 2000.
- [121] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007.
- [122] B. Wetzstein, P. Leitner, F. Rosenberg, I. Brandic, S. Dustdar, and F. Leymann. Monitoring and analyzing influential factors of business process performance. In *EDOC*, pages 141–150. IEEE Computer Society, 2009.
- [123] M. Wieland, O. Kopp, D. Nicklas, and F. Leymann. Towards context-aware workflows. In *CAiSE'07 Proceedings of the Workshops and Doctoral Consortium*, 2007.
- [124] P. Wohed, W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, and N. Russell. On the Suitability of BPMN for Business Process Modelling. In *Business Process Management*, volume 4102 of *LNCS*, pages 161–176. Springer, 2006.
- [125] X. Zhao and C. Liu. Version management in the business process change context. In *Business Process Management*, volume 4714 of *LNCS*, pages 198–213. Springer, 2007.
- [126] M. zur Muehlen and J. Recker. How Much Language Is Enough? Theoretical and Practical Use of the Business Process Modeling Notation. In *Proc. CAiSE*, pages 465–479, 2008.

