PhD Dissertation

International Doctorate School in Information and Communication Technologies

DISI - University of Trento

Service Composition in Dynamic Environments: From Theory to Practice

Heorhi Raik

Advisor:

Prof. Marco Pistore

Fondazione Bruno Kessler

December 2012

"Adapt or perish, now as ever, is nature's inexorable imperative."
H. G. Wells

To my parents.

Acknowledgement

While recalling the four years of work that eventually resulted in this dissertation, I realize how lucky I was to meet and deal with so many nice people. First of all, I would like to thank my adviser Marco Pistore for providing invaluable support and readily sharing his knowledge and experience. Marco made a lot of effort to make me a researcher (hopefully, a good one) and was constantly bringing to me his unprecedented (but always well-grounded) optimism about the things we were working on together. I would also like to express my great appreciation to Raman Kazhamiakin who, through our daily discussions and collaboration, accompanied me in my first steps in research and helped me a lot in understanding various aspects of service-oriented computing (and research in general).

My grateful thanks are also extended to all the members of the SOA unit of the Fondazione Bruno Kessler who made up the creative environment in which the content of this dissertation was produced. In particular, I would like to mention Antonio Bucchiarone and Annapaola Marconi (for our joint work concerning process adaptation and for their valuable feedback on the first draft of this manuscript); Piergiorgio Bertoli (for our endless discussions of AI planning issues); Nawaz Khurshid and Claudio Antares Mezzina (for our collaboration in the development of ASTRO-CAptEvo platform). Moreover, I wish to say a lot of good words about the other two PhD students of our group, Adina Sirbu and Asli Zengin, who, after four years of studying together, have become my true friends.

I want to specially acknowledge the people of DoCoMo Euro-Labs (Munich, Germany), and in particular Massimo Paolucci and Matthias Wagner, for giving me a great opportunity to have a three-month internship in the company and making it possible for me to gain an extremely important experience of doing research in close vicinity to the industry.

Of course, I would like to express my deep gratitude to the dissertation reviewers Luciano Baresi, Paola Inverardi, Maurizio Marchese and Massimo Paolucci for taking the responsibility and spending their time on reading the thesis and preparing many excellent questions and comments. Without them, the last step of my PhD adventure would not have been possible.

Finally, I wish to thank my fiancée Dina Shakirova for her endless support, love and patience, and my best friends Siarhei Bykau, Maksim Khadkevich, Ivan Tankoyeu and Yury Zhauniarovich for greatly contributing to my personal development. Ultimately, I would like to express my deepest respect and love to my parents, who, from the very childhood, taught me how to make decisions and how to take responsibility. This dissertation is dedicated to them.

Abstract

In recent years, service-oriented architecture (SOA) has become one of the leading paradigms in software design. Among the key advantages behind SOA is service composition, the ability to create new services by reusing the functionality of pre-existing ones. Despite the availability of standard languages and related design and development tools, "manual" service composition remains to be an extremely error-prone and time-consuming task. No surprise, the automation of service composition process has been and still is a hot topic in the area of service computing.

In addition to high complexity, modern service-based systems tend to be dynamic. The most common examples of dynamic factors are constantly evolving set of available services, volatile execution context, frequent revision of business policies, regulations and goals, etc. Since dynamic changes of the execution environment can invalidate service compositions predefined within a service-based system, the cost of software maintenance in this case may increase dramatically. Unfortunately, the existing automated service composition approaches are not of much help here. Being design-time by their nature, they intensively involve IT experts, especially for analysing the changes and respecifying formal composition requirements in new conditions, which is still a considerable effort. To make service-based systems more agile, a new composition approach is needed that could automatically perform all composition-related tasks at run time, from deriving composition requirements to generating new compositions to deploying them.

In this dissertation, we propose a novel service composition framework that (i) handles stateful and nondeterministic services that interact asynchronously, (ii) allows for rich control- and data-flow composition requirements that are independent from the details of service implementations (iii) exploits advanced planning techniques for automated reasoning and (iv) exploits modeling methodology that is applicable in dynamic environment.

The corner stone of the framework is the explicit context model that abstracts composition requirements and constraints away from the details of service implementations. By linking services to the context model on the one side, and by expressing composition requirements and constraints in terms of the context model on the other side, we create a formal setting in which abstract requirements and constraints, though being implementation-independent, can always be grounded to available service implementations. Consequently, we show that in such framework it is possible to move most human activities to design time so that the run-time management of the composition life cycle is completely automated. To the best of our knowledge, it is the first composition approach to achieve this goal.

A significant contribution of the dissertation is the investigation of the problem of dynamic adaptation of service-based business processes. Here, our solution is based on the composition approach proposed. Within the thesis, the problem of process adaptation plays the role of the key motivator and evaluation use case for our composition-related research.

The most part of the ideas discussed in the thesis are implemented and evaluated to prove their practical applicability.

Keywords

service composition, planning, dynamic environment, process adaptation

Contents

1 Introduction **1**
1.1 Context-Aware Service Composition 5
1.2 Dynamic Process Adaptation 10
1.3 Structure of the Thesis 12
1.4 Thesis Context 14

2 State of the Art **15**
2.1 Service-Oriented Architecture 16
2.1.1 SOA with Web Services 18
2.1.2 Composition of Web Services 21
2.2 Automated Service Composition 24
2.2.1 Existing Approaches 29
2.2.2 Discussion 36
2.3 Adaptation of Service-Based Business Processes 40
2.3.1 Existing Approaches 44
2.3.2 Discussion 48
2.4 Problem Statement 51

3 Process Adaptation in Dynamic Environments **55**
3.1 Motivating Example 57
3.2 Approach Overview 59
3.2.1 Application model 59
3.2.2 Adaptation Mechanisms 64
3.2.3 Adaptation Strategies 67

3.3 Adaptable Pervasive Flows and APFL 70
3.4 Discussion 72

4 Context-Aware Composition of Fragments 75
4.1 Overview 76
4.2 Composition Model Elements 78
4.2.1 Context 78
4.2.2 Annotated Fragments 82
4.2.3 STS to APFL 89
4.2.4 Context-Aware System and Annotation Semantics . . 91
4.2.5 Composition Requirements 94
4.3 Problem of Context-Aware Fragment Composition 94
4.4 Composition Problem as Planning Problem 102
4.4.1 Grounded Context 104
4.4.2 Context-Aware Execution Domain From Grounding . 109
4.5 Algorithm 110
4.6 Discussion 117

5 Composition in Dynamic Environment 119
5.1 System Operation 119
5.1.1 General Adaptation Problem 122
5.1.2 Evolution of System Configuration 123
5.1.3 Managing Adaptation Strategies 130
5.2 Fragment Composition in Dynamic Environment 132

6 Context-Aware Composition of Services 139
6.1 Motivating case study 140
6.2 Composition Model Overview 145
6.3 Context Model 148
6.4 Services and Service Orchestration 150
6.4.1 Service Model 150
6.4.2 WS-BPEL Services as STS 150

6.4.3 Observable Behaviour 155
6.4.4 Services as Fragments 159
6.5 Service Annotations 162
6.6 Problem of Context-Aware Service Composition 164
6.7 Discussion 165

7 Advanced Topics in Service Composition 169
7.1 Control-Flow Requirements 170
7.1.1 Language and Semantics 170
7.1.2 Requirements as STS 176
7.1.3 Problem of Continuous Composition 186
7.1.4 Approach Architecture and Prototype Algorithm . . . 191
7.2 Data-Flow Requirements 204
7.2.1 Datanet Overview 205
7.2.2 Context-Aware Datanet 212
7.3 Discussion 215

8 Implementation and Evaluation 219
8.1 Composer 219
8.1.1 Implementation 219
8.1.2 Evaluation 222
8.2 ASTRO-CAptEvo 228
8.2.1 Implementation 229
8.2.2 Evaluation 237

9 Conclusions and Future Work 247
9.1 Future Work 249

Bibliography 253

List of Tables

3.1 Basic and structured activities of APFL 72

4.1 Translation of basic APFL activities into STSs 86
4.2 Translation of structured APFL activities into STSs 87

6.1 Translation of basic WS-BPEL activities into STSs 152
6.2 Translation of structured WS-BPEL activities into STSs . . 153
6.3 Service annotations from virtual travel agency scenario . . . 164

List of Figures

2.1 Web Services stacks 19
2.2 SOA with Web Services 20
2.3 Choreography and orchestration 23
2.4 Automated service composition 25
2.5 Adaptation concepts 41
2.6 Architecture of adaptable system 42

3.1 Process chain of the car logistics scenario 58
3.2 Artifacts of application model 60
3.3 Examples of context properties in car logistics scenario . . . 62
3.4 Adaptation mechanisms in car logistics scenario 63

4.1 Fragment composition model 76
4.2 Context property for car status 79
4.3 Fragment model of "Car Repair" fragment 83
4.4 Fragment annotation of "Car Repair" fragment 85
4.5 Annotated APFL process of Landing Manager 88
4.6 Landing Manager process as STS 89
4.7 Fragment composition approach 103
4.8 Search algorithm 111

6.1 Flight Booking Service 141
6.2 Flight Notification Service 141
6.3 Flight Cancellation Service 142
6.4 Hotel Management Service 143

6.5 User Protocol 144
6.6 Service composition model 146
6.7 Context properties in virtual travel agency scenario 148
6.8 Flight Booking Service as STS 154
6.9 Observable behavior of services from virtual travel agency scenario 158
6.10 Services with undecidable observable behaviour 159

7.1 Requirements as STS 182
7.2 Continuous composition approach 192
7.3 Goal filtering algorithm 196
7.4 Algorithm for deriving state-action tables 198
7.5 Main planning routine 200
7.6 Original Datanet 208
7.7 Architecture of context-based Datanet 213
7.8 Context-based datanet for virtual travel agency scenario . . 216

8.1 Planning procedure overview 222
8.2 Solution for virtual travel agency scenario 224
8.3 Performance scalability charts 226
8.4 ASTRO-CAptEvo architecture 229
8.5 Scenario Viewer 235
8.6 Process Viewer 237
8.7 Adaptation Viewer 238
8.8 Dependency between performance and number of services composed 242
8.9 Complexity distribution and performance scalability 243

Chapter 1

Introduction

Service composition is one of the cornerstone technologies within service-oriented architecture. It consists in reusing the existing services as building blocks for new services (applications) with higher-level functionality. Due to the fact that, according to the main SOA principles, services are designed to be platform-independent, loosely-coupled, abstract, autonomous and self-describing, service composition is an extremely powerful technique in hands of software engineers. First of all, it allows for rapid development of new applications, especially when all the necessary components are already available. Second, it promotes high reusability of the development results: composite services can further be used as building blocks for even more high-level services. Finally, service-based applications obtained as a result of service composition are very and can be quickly tuned and adjusted to new business requirements and changes in the service infrastructure.

Despite all the advantages service composition brings to software engineers, when performed manually it is still a very complex, time-consuming and error-prone task. Such complexity mostly comes from the fact that in order to build a composition, the developer has to know very well both composition requirements and numerous technical details of each service to be composed (efficient service discovery and selection is another dimension

of this complexity). Moreover, while developing the composition, a lot of easy-to-miss but critical technical aspects have to be reflected. Finally, languages used for describing both services and their composition, though standardized, are not at all easy to read and write (the XML-languages used here are rather machine-readable than human-readable). All this requires profound technical skills and knowledge, and takes a lot of effort and time to produce a composition; but even in this case the probability of errors is still quite high.

Such complexity becomes a real problem when we start to use SOA in *dynamic* application domains. For example, if we do business that requires frequent revision of partners (service providers) and constant correction of business policies and goals (composition requirements), it is very likely we would have to change/adapt our composition-based applications again and again. That would result in a higher cost of software maintenance and this is what we, as a business owner, would prefer to avoid. The solution lies in creating *automated service composition* techniques.

From the very origin of service composition, it was recognised as a good candidate for automation. First, services are self-descriptive and are supposed to provide complete information about themselves in machine-readable standard format (e. g., WSDL [120] and WS-BPEL [91]). Second, the behaviour of services and their composition can be described using the well-established mathematical models, such as state transition systems, Petri nets, etc. As a result, a lot of interest has been recently demonstrated in the field of automated service composition. Unfortunately, we have to admit that many approaches suffer from oversimplification and can only be applied to a very restricted set of composition problems. At the same time, some approaches have gained enough maturity to be used with a wide range of service composition problems of real-world complexity. For example, the approach developed in the context of the ASTRO

project [4] and published in [103, 79, 18] performs service composition of stateful, nondeterministic and asynchronous services using planning techniques. The approach elaborates both control- and data-flow requirements and can work with standard service specifications: the input is provided in form of WSDL and Abstract WS-BPEL service descriptions and the output is an executable orchestration expressed in WS-BPEL. To give an idea about to what extent automated tools may speed up the composition process, in [77] the authors tried to solve a real composition problem both manually and using the ASTRO technique. It turned out that the manual composition took around 20 hours, while using the automated approach the result was produced in around 1 hour. It was a strong evidence automated service composition really makes difference. However, what was important for us about this experiment is that in case of automated composition three quarters of time were spent on human operations such as analysing service description and formalizing composition requirements. Although such overhead was not a problem for the case study examined, in modern applications featuring unprecedented level of dynamicity, even the fact of human involvement may become a hurdle on the way to success.

While surveying application domains where modern SOA is used, we can notice that it often operates in so extremely dynamic setting, that service composition is considered to be a kind of "every-minute" routine activity. We can mention at least two examples of such systems that become extremely important in modern information technology. The first one is *pervasive systems* ([57]), which are mobile systems operating in close connection with the information about their surrounding (context). Once service composition is exploited in such a system, it has to be flexibly and quickly adapted to the rapidly changing environment. For instance, let us imagine there is a car that has to regularly perform some activity implemented through composition of surrounding near-field communication

services (e.g., car parking assisted by various parking services). Depending on the current surrounding (i.e., on the set of available services and on the context) the solution composition, though targeting conceptually the same objective, will always have different implementation. If a car has to repeat this activity once an hour under different conditions, it would really be unaffordable to involve a human to compose services, even if assisted by conventional composition tools. Another example is *user-centric systems* ([59]), whose operation evolves around the needs and constraints of a specific user. For instance, it could be a mobile application that allows the user to integrate (compose) multiple mobile services (local phone services, Internet services, near-field communication services etc.) and execute them consistently. In this case, the choice of services and the composition objectives are determined by user's surrounding and personal preferences/constraints/goals. Moreover, the need for such composition may arise at any point in time. Obviously enough, it is not feasible to involve an IT expert to produce this kind of composition. Unfortunately, predefined solutions are not of much help here either, since each user has her own idea about how the composition has to accomplish her tasks.

In this dissertation, we come up with a *novel approach to automated service composition* that is specifically designed to be used in dynamic execution environments (like the ones discussed above). In very general terms, the idea of the approach consists in organizing the composition life cycle is such a way that the most part of human activities can be accomplished at design time. As a consequence, the run-time composition management, from the derivation of composition requirements to the composition synthesis to the deployment of executable processes, requires no (or minimal indeed) human involvement. The approach works with a realistic service model and allows for rich control- and data-flow requirements. This makes it powerful enough to deal with composition problems of real complexity.

Along with the problem of service composition, we also investigate the problem of *dynamic adaptation of service-based business processes* and propose a solution based on our composition technique. Within this part of our research, not only do we show how to exploit service composition for the purposes of process adaptation, but also 1) propose an adaptable process model, 2) identify various adaptation strategies and 3) consider the problem of automated selection and enactment of adaptation strategies at run time. Being an important research problem per se, dynamic process adaptation is also a strong motivating case study and a realistic testing platform for our service composition approach.

As such our approaches to service composition and process adaptation make up the two major contributions of the thesis. In the following two sections we discuss them in more detail. After that, we outline the dissertation structure and briefly overview the publications and collaboration related to this PhD activity.

1.1 Context-Aware Service Composition

To gain some background for our composition approach we carefully examined the aforementioned ASTRO approach. In particular, we adopted with some changes the service model where services were modeled as state transition systems with different types of actions (in our case, controllable and uncontrollable). We found this model successful since it allowed us to work with stateful, nondeterministic services with asynchronous behaviour. We also generally took over some of the composition-as-planning principles proposed in ASTRO. At the same time, ASTRO was extremely useful to understand the limitations of conventional composition techniques in dynamic setting. While analyzing it, we realized that the most critical drawback of ASTRO from the perspective of dynamic environment lay in

the strong dependency between composition requirements and details of service implementation. It became the starting point in our research.

The central idea of our composition approach consists in detaching composition requirements from service implementations so that the former does not include any details about the latter. At the same time, we provide a mechanism for run-time linkage of such conceptual requirements with particular service implementations. As a result, the same group of requirements can be used with various sets of service implementations. We show that in this case a bigger part of the composition modeling effort can be shifted to design time so that composition life cycle is managed at run time automatically. In our approach, in addition to other technical details, we specifically elaborate control-flow and data-flow requirements. For control-flow requirements, we propose our own simple yet expressive abstract language that is able to express goals with preferences, procedural goals and reactive goals. For data-flow requirements, we adopt the approach of [79] and deeply modify it in order to make it compliant with our modeling methodology.

Services and Fragments

Through the dissertation we use two different types of reusable components: *services* and *process fragments* (or simply fragments). Services are conventional web services described by their interface (WSDL) and protocol (WS-BPEL). Process fragments ([39]) is a way to represent reusable process knowledge in process composition. To keep it simple, process fragments can be considered as an analogue of services in the world of business processes. In fact, process fragments encode elementary subprocesses that can be used as constructing blocks for more complex processes. To model service-based process fragments, we adopt a language specifically designed for that (APFL [23]).

The difference between a service protocol and a service-based process fragment can be described as follows. First, service protocols describe communication between the service and the client from the perspective of the service, while process fragments use the perspective of the client. This makes process fragments suitable not only for specifying the communication model but also for indicating internal operations (e.g., human operations, data manipulations) to be performed on the client side (which may sometimes be a big advantage). Second, process fragments may additionally include activities that are not directly related to services: for example, APFL fragments may include concrete activities (to model any custom activity, e.g., human operation) and abstract activities (to model complex activities whose implementation is to be refined at run time). This makes process fragments much more expressive and much more suitable for the purposes of process adaptation (this is essentially why we consider process fragments along with conventional service protocols). Finally, service protocol is associated with a single service, while a single process fragment may describe interaction with a group of services, which generally also makes fragments more flexible.

In the thesis, we first present the composition of fragments. Later, while considering service composition, we show that fragments can be effectively exploited to deal with partial observability of service behaviour. In brief, for each service protocol we define an APFL-like fragment (we call it *complementary fragment*) that contains only service communication activities and that reflects the acceptable behaviour of the client while communicating to this service. As such, service composition can be essentially reduced to the composition of respective complementary fragments and so the same composition technique can be used for both fragments and services.

In the rest of this chapter we predominantly use term “service” while meaning that it is valid for both services and fragments.

Context as Abstraction Layer

We indicated that the central idea of our approach consists in abstracting composition requirements from details of service implementations. We use explicit context model as such *abstraction layer*. The context is modeled using special state transition systems (context properties). They reflect possible context situations (states) and context events (transitions between states). Composition requirements are specified in terms of context model (for example, to reach some context state or to trigger some context event), which makes them service-independent. In turn, service specifications are equipped with special *annotations* connecting them to context model. Consequently, through service annotations context-based requirements can always be grounded on specific service implementations and, as such, can be combined with them into a composition problem.

Control Flow

To manage control flow of the composition, we introduce 1) a context-based (abstract) language for control-flow requirements and 2) control-flow annotations in service specifications. Our control-flow language allows for *reachability* goals (context states to be reached), *procedural* goals (context events to be triggered) and the mixture of them. In addition to that, it is possible to define a number of alternative goals ordered according to their *preferences* (goals with preferences). Finally, we introduce a way to specify *reactive* requirements, e.g., to achieve some goal in reaction to some context situation (e.g., context event). A lot of attention is paid to formally defining the semantics of this language.

The control flow of services is connected to the control flow of context through annotations. In particular, activity *effects* are used to indicate that activity execution triggers certain context events. Similarly, activity

preconditions are used to indicate that an activity can be executed only in certain context situations (states). In fragments, abstract activities are additionally annotated with *contextual goals* indicating their abstract objectives. Annotations are used to ground abstract control-flow requirements on service implementations.

Data Flow

Although in this work data flow receives less attention than control flow, realizing its importance we develop a prototype solution for managing data flow. It relies on the Datanet technique presented in ([79]), which is significantly modified in order to be compliant with our modeling methodology.

The idea of the Datanet technique is very intuitive: it explicitly links various data parts of service ports in order to show how the composition can calculate the data of outgoing messages from the data of incoming messages. However, the Datanet is implementation-dependent (it depends on concrete service ports). To tackle this problem, we modify the Datanet approach so that it conceptually follows th modeling methodology used for control-flow requirements: an implementation-independent (abstract) part of requirements is specified at design time while services are provided with annotations linking them to the abstract requirements. Our data-flow requirements can generally be handled in planning using the same principles as in [79].

Composition as Planning

A significant amount of this dissertation is devoted to the problem of transforming a formal composition problem comprising services, context and composition requirements, into a planning problem. In brief, the planning domain is obtained by fusing service STSs, context properties and requirements STSs. The requirements STSs are derived from composition

requirements in such a way that in the resulting planning domain for each state it is possible to say if requirements are satisfied in it (and with which preference) or not. Consequently, the states where requirements are satisfied become goal states of a planning problem.

Implementation and Evaluation

As a planning technique, our approach uses planning via model checking. We show that even in the presence of context, for simple cases of control-flow composition requirements (reachability of context states) it is possible to reuse the existing algorithm introduced in ASTRO ([18]). However, when we express control-flow requirements using our new language, it is necessary to build a conceptually new type of plans, so called *continuous plans*. Differently from a conventional plan, a continuous plan not only indicates how to reach goal states from the initial state, but also shows how to bring the system back to goal states ones it is forced to leave the goal state already reached. Moreover, our new algorithm is implemented in the presence of goals with preferences. The correctness of the new planning algorithm is evaluated on the complex Virtual Travel AGency case study. Moreover, we evaluate the performance scalability using a number of artificial scalable scenarios.

1.2 Dynamic Process Adaptation

Although our adaptation framework relies on the aforementioned composition approach, there are some important adaptation-related issues that have to be addressed to make our solution possible.

Adaptation Strategies

Adaptation strategy indicates how the original process has to be changed in order to adapt to certain situations. We show that many adaptation strategies for process adaptation can be defined. For example, one strategy may consist in trying to solve a problem without "touching" the original process so that the execution of the latter can be resumed as if nothing happened (local adaptation). Another strategy may imply process rollback with compensation of the effects of the activities already executed.

We demonstrate that having APFL processes and process fragments with context annotations, it is possible to define a compact set of adaptation strategies that can effectively cover a large portion of problems that may happen during the execution. We also show that our context-based formal framework allows for efficient mechanisms for problem detection. A special discussion is devoted to the problem of proper selection of adaptation strategies to be used in certain situations.

Adaptation as Composition

Despite the diversity of the adaptation strategies used in our approach, all of them are realized through context-aware composition of fragments. In this regard, the difference between the strategies consists in 1) how many compositions a strategy requires to be implemented, 2) how the composition goal is derived from the specification of the process to be adapted and from the current status of the execution environment and 3) how the targeted process is changed as a result of adaptation and how the results of composition are embedded into it. We also pay attention to the issue of executing "statically" designed composition in dynamic environment.

Implementation and Evaluation

In order to demonstrate our adaptations ideas in action, we implement the demo platform ASTRO-CAptEvo based on the car logistics scenario. In fact, the platform realizes a very simple custom SOA infrastructure that is additionally equipped with facilities for dynamic adaptation. The platform is organized as a set of entities operating within the scenario and collaborating with each other in order to achieve their business objectives. Each entity follows its own business process, which can be adapted in case of problems. Moreover, an entity exposes process fragments that can be used by other entities in order to collaborate with it. We can say that ASTRO-CAptEvo models a pervasive environment. It allows us to demonstrate the possibilities of APFL language with context annotations, the effectiveness of adaptation strategies and mechanisms introduced, the applicability of our composition techniques to the problem of process adaptation and the flexibility of our modelling methodology. We use the platform to evaluate most of the ideas presented in the thesis, both qualitatively and quantitatively.

1.3 Structure of the Thesis

In the context of the thesis, the two main topics (automated service composition and process adaptation) are very much connected to each other and interweave a lot. As a result, it was not always easy to decide on the order in which different elements of the thesis should be presented. We hope that the final structure we came up with makes the dissertation a single story that is easy to follow.

The work consists of nine chapters. Chapter 2 provides background information on the main topics of the thesis. This includes some basic information on SOA and related standards and more profound description

of the state of the art in the areas of automated service composition and process adaptation. The chapter also identifies some important open issues in the respective areas. These issues make up the problem statement of the dissertation. Chapter 3 provides an overview of our approach to dynamic adaptation of service-based business processes. We cover this topic first because 1) it is the key motivator for our research on run-time automated service composition, 2) it gives a nice example of what we mean by dynamic execution environment and 3) it introduces some concepts that are important to understand the chapters to come (e.g., APFL fragments). In Chapter 4 we introduce our approach to the composition of process fragments. In Chapter 5 we show how the fragment composition approach of Chapter 4 can be integrated into the adaptation framework of Chapter 3. Here, we also consider the issues related to the execution of compositions in dynamic environment. In Chapter 6 we discuss the relation between process fragments and services. We show that the partial observability of services can be successfully treated by replacing services with fragments. In this case, the approach of Chapter 4 can be applied to services practically unchanged. Chapter 7 explains how the service composition approach of Chapter 6 can be extended with advanced control- and data-flow composition requirements. Here we also present the modified planning algorithm that supports these requirements. Chapter 8 is devoted to the implementation and evaluation of the ideas proposed in the thesis. Finally, Chapter 9 contains concluding remarks and ideas for future work.

We would like to mention, that many concepts presented in the thesis are demonstrated in action in our platform ASTRO-CAptEvo, that is described in Chapter 8 and that is freely available on the Web[1].

[1] `http://www.astroproject.org/captevo.php`

1.4 Thesis Context

In this section, we briefly describe research activities and publications related to this dissertation.

Most of the ideas presented in Chapter 4, 6 and 7 were originally developed in the scope of the YourWay! project. YourWay! was a three-year bilateral collaboration between the Service-Oriented Research Unit of FBK-IRST and DoCoMo Euro-Labs (Munich, Germany). The project aimed at developing new resource-driven service composition approach for user-centric service provisioning in mobile environments. The related publications covered context model as abstraction layer and the novel context-based language for control-flow requirements (ICWS 2009, [17]), the corresponding planning algorithm for continuous composition (ICAPS 2009, [16]) and the user-centric aspects of the composition (CoopIS 2010, [60]).

The development of our approach to dynamic process composition (Chapters 3 and 4) was initiated in the scope of the two research projects funded by the European Commission under the 7th Framework Programme. The S-Cube project [2] was the European Network of Excellence in Software, Services and Systems and the ALLOW project [1] was a research project with the goal of developing a new, flow-based programming paradigm for adaptable pervasive systems. The related publication described the basic principles of our adaptation framework (SOCA 2011, [26]) and the prototype implementation of the ASTRO-CAptEvo framework (demo track at ICSOC 2011, [24]).

The further development of the approach included the investigation of a realistic scenario from the car logistics domain (PESOS 2012, [22]), elaboration of adaptation strategies (Best Paper Award at ICWS 2012, [25]) and the demonstration of a new edition of the ASTRO-CAptEvo platform at the Service Cup competition (winner of the Service Cup 2012, [105]).

Chapter 2

State of the Art

In this chapter we discuss the most significant and recent advances in the areas of service-oriented computing that are central to the dissertation. The first section (Section 2.1) provides the overview of the principles of service-oriented architecture, introduces important definitions and describes the main standards currently in use. The remaining sections of the chapter are devoted to the current progress in two areas of interest. In particular, in Section 2.2 we explain the general idea of automated service composition and present the main existing approaches addressing this important problem. Section 2.3 surveys the most important works in the area of adaptation of service-based business processes. In addition to the critical assessment of the existing works, each area-specific section features a brief discussion where we identify important problems that have not or have weakly been addressed by the existing approaches and, as such, have become the focus of this dissertation. In the conclusion of the chapter we show how the aforementioned research problems are related to each other and how the thesis is supposed to extend the state of the art in the respective areas of interest.

2.1 Service-Oriented Architecture

Service-oriented architecture (SOA) is a software engineering paradigm in which software is designed and developed in form of interoperable services. The creation of SOA was inspired by the necessity to develop and support complex cross-enterprise information systems that can be quickly and cost-efficiently adapted to the changes in the operational environment. Conceptually, SOA is the next step in the evolution of distributed computing, whose development has chronologically gone through the steps of client-server systems [41], multi-tier systems [40] and RPC-based systems such as CORBA [93].

The basic intuition of SOA is to construct business applications of loosely-coupled, autonomous and reusable components called services. However, behind this simple idea there is a bunch of principles that have to be followed by SOA implementations in order to release the SOA potential. The very core of them are as follows [45]:

- *Standardized service contract.* Services adhere to a communication agreement (or service contract) as defined by service description documents. Within the same service inventory, services use the same service description standards;
- *Service loose coupling.* Service contract is not tightly coupled with customer requirements nor with service implementation. In this case the contract can evolve without affecting service consumers and service implementations;
- *Service abstraction.* Besides the details in the service contract, services hide their internal logic;
- *Service reusability.* Service logic is arranged so that to promote its reuse.

- *Service autonomy.* Services have a high level of control of the underlying run-time execution environment;
- *Service statelessness.* Services should be maximally freed up from the management of state data.
- *Service discoverability.* Services are equipped with communicative meta data by means of which they can be discovered by consumers;
- *Service composability.* Services can be effectively composed into new services with the functionality of arbitrary complexity.

The analysis of service-oriented systems suggests that the above principles make business applications much more flexible and significantly decrease the cost of their initial development (especially, when new applications are "composed" on top of existing services) and further support. The high level of abstraction of service contracts makes it possible to efficiently integrate and manage services on the level of business functionality, no matter which underlying platforms they use. Finally, such features as discoverability and composability allow for applications with unprecedented level of adaptability to environmental changes.

The implementation of SOA is a complex task that involves such subjects as networking, knowledge representation and semantics, artificial intelligence, security, data management and others [95]. Moreover, to enforce the compliance with the core principles, and to cover various aspects of information systems from the perspective of service orientation, SOA often relies on numerous standards.

Service-oriented architectures can be implemented using a wide range of technologies such as REST [107], Java RMI [94] and many others. Nonetheless, the most popular and complete implementation of SOA is based on Web Services [127]. Although, our research is detached from particular

SOA implementations, in the rest of this dissertation, we agree to use Web Services and their related terminology, standards and specifications for the demonstration and explanation purposes.

2.1.1 SOA with Web Services

In very general terms, Web Services can be considered as a method of communication between two partners via the Web. The development and standardization of Web Services technology is coordinated by the World Wide Web Consortium (W3C) in the framework of Web Services Activity [127]. The Web Services Glossary by W3C [124] succinctly defines Web Services as follows:

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

In the Web Services Activity Statement, it is also explicitly stated that Web Services only define the way components interact and do not impose any restrictions on the underlying implementation technologies and platforms. It is also mentioned that one of the important aspects of Web Services is the ability to combine services "in a loosely coupled way in order to achieve complex operations". As a result, "programs providing simple services can interact with each other in order to deliver sophisticated added-value services" [128]. As we can see these definitions introduce the main principles of SOA such as loose coupling, standardized service contract, abstraction, composability, etc.

The success of Web Services is based on the set of specifications that standardize all important aspects of the technology, from service communication interface and protocol to coordination within service-based applications [70]. The simplified stack of Web Services standards is given in Fig. 2.1 (the complete version can be found in [55]). As one can see, the stack covers various levels of Web Services infrastructure. The message transportation relies on standard Web protocols such as HTTP and HTTPS. The format of messages is standardized by Simple Object Access Protocol (SOAP) [119], where the data format is defined using XML.

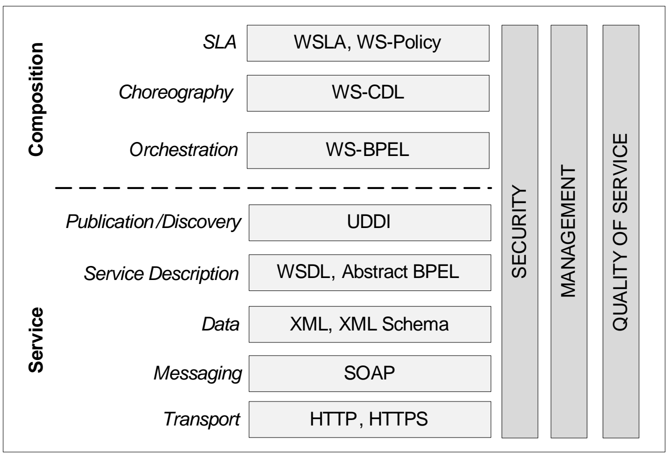


Figure 2.1: Web Services stacks

In order to let the customer know how a web service can be exploited, web services provide unified description documents. The most important of them concern service interface (a set of operations supported) and service protocol (valid operation sequences). Web Service Description Language (WSDL) [120] is a standard for specifying web service interfaces. Here a service interface is a set of operations, each defining its input and output message. The structure of the data within a message is defined by XML Schema [125]. A service protocol describes how the service state is affected by operation execution and suggests valid operation sequences. In fact, a

service protocol can be defined using any workflow language. Nevertheless, there are languages specifically created for Web Services, and that are compliant with other related standards (e.g., WSDL). One of them is Abstract BPEL, which is a part of Web Service Business Process Execution Language (WS-BPEL) [91], a language for executable service-based business processes. It is worth to mention that description standards are not limited to the two aforementioned. There are numerous standards for covering various other aspects of web services. For example, OWL-S [121] and WSMO [126] are attempts to provide rich service descriptions for Semantic Web.

The centralized storage and advertising of service descriptions is necessary to provide discoverability of services. For that purpose, the Universal Description Discovery and Integration (UDDI) [90] specification has been proposed. UDDI standardizes registries that services can exploit to advertise themselves on the Web. In turn, the customer can use the registries to discover, locate and execute services.

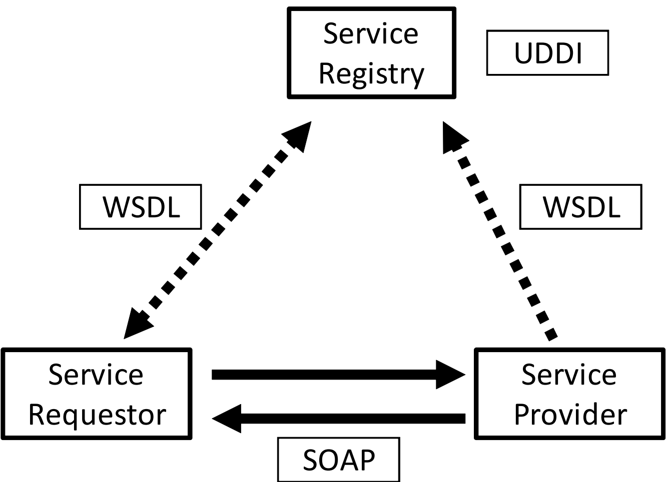


Figure 2.2: SOA with Web Services

The conventional model of service-oriented architecture based on Web Services is depicted in Fig. 2.2. Here, a *service provider* develops a ser-

vice and accompanies it with service description documents (e.g., a WSDL document). Service description comprising the WSDL document and other information is submitted to a *service registry* (UDDI registry) and becomes available to potential customers. A *service requester* (i.e., the customer) uses the service registry to discover available services and obtain their descriptions. Having a service description, the service requester can communicate to the service by means of SOAP messaging.

The Web Service Architecture [122] shows that this basic model can be extended in different directions. For example, the architecture does not impose any particular restrictions on the service descriptions accessible via UDDI repository (though they should be machine-readable), the binding of services may be dynamic or static, the process of service selection and discovery "may be performed by an agent, or by an end-user" using different selection criteria etc.

Substantial extensions of the basic model origin from the necessity to coordinate the execution of multiple services. In fact, the full potential of Web Services is released only when we speak about an ecosystem of numerous service providers and service consumers collaborating with each other in order to achieve certain business goals. In this situation, services are combined into business applications (or *service compositions*) that provide coordinated execution of multiple services driven by complex business logic. No surprise, a significant amount of technologies and standards in SOA address the problem of service composition.

2.1.2 Composition of Web Services

One of the driving ideas of SOA is service composition [62], which is the possibility to quickly create new services and applications (*composite services*) by "composing" the existing ones (*component services*). Service composition comes into play when services that are currently present do

not provide the needed functionality or when coordinated execution of existing services is required in the context of some business objective. Very often the process of service composition is recursive, i.e., the newly created composite service can consequently be used as a component in future compositions.

Service composition is a complex problem that generally consists of a few steps. First, based on the business requirements, suitable services have to be selected from those available on the Web [73]. Then, service composition has to be formally specified using one of standard languages (e.g., WS-BPEL [91] or WS-CDL [123]). This may involve automated synthesis algorithms [103]. Finally, the service composition has to be verified against the compliance with business and composition requirements [61], executed, monitored [100] and, if needed, adapted [78].

One might notice that any of these steps is by itself a complex research problem. For instance, the problem of requirements engineering for service composition often has to consider such aspects as functionality, quality of service, security, business policies and others. Similar set of features have to be taken into account while selecting services. Moreover, to allow for intelligent selection, services have to be properly annotated with relevant information. The same high level of complexity is attributed to the steps of composition specification, verification, monitoring and adaptation.

There exist two conventional ways two compose services: *choreography* and *orchestration* [98]. The difference in the interaction model of these composition types is explained in Fig. 2.3. Choreography describes interaction protocols between different participants of the composition from the global perspective. As such, the composition logic is distributed among all participants. Orchestration is characterized by the existence of a central component (orchestrator) that has full control of the composition logic and facilitates interoperability between component services. In this case, the

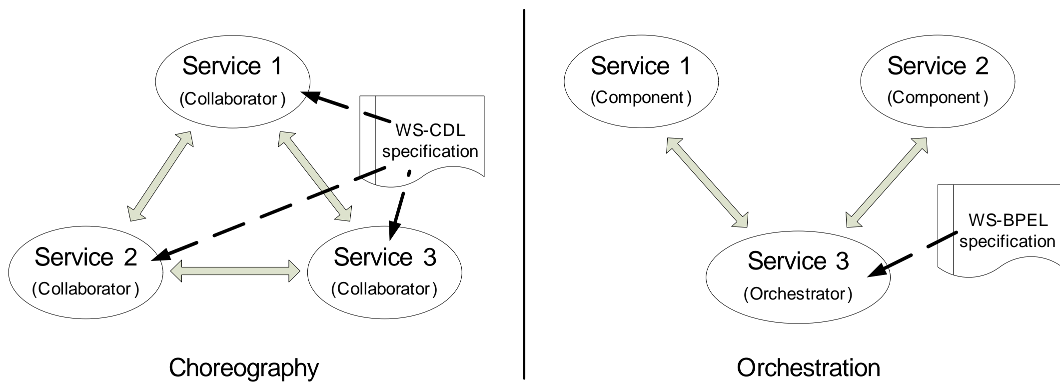


Figure 2.3: Choreography and orchestration

components are normally not aware of the composition. It is worth to mention that two types of composition may co-exist within one service-based application, e.g., a choreography of orchestrations. In addition to that, from the choreography specification it is always possible to derive a number of orchestrations, where choreography participants play a role of orchestrators [83]. In [3], very specific minor composition models are additionally distinguished (service coordination and service assembly).

One of the main standards for specifying choreographies is Web Services Choreography Description Language (WS-CDL) [123]. This language is used to describe peer-to-peer collaborations between parties through the definition of their observable behaviour (message exchange) from the global perspective. When the message exchange between the partners is organized according to the WS-CDL specification, a certain business goal is achieved.

The standard languages for orchestrations are represented, first of all, by Business Process Execution Language (BPEL). The part of this standard devoted to the description of executable service-based business process is called Executable BPEL. It is used to define workflows comprising such activities as service invocation, message send and receipt (for asynchronous interaction), events, operations on variables. The language is equipped with a set of standard control-flow constructions (sequential and parallel

execution, loops, conditional branching etc.) in order to define workflows of arbitrary structure and complexity. The process specification can introduce variables and perform various manipulations on them. There exist numerous engines for the direct execution of BPEL processes (e.g., Apache ODE [47], ActiveVOS [44]).

Speaking about BPEL, it is also worth to mention Business Process Model And Notation (BPMN) [92], a powerful approach to defining business processes on conceptual level. Although there is no one-to-one correspondence between BPEL and BPMN [71], BPMN is frequently used for specifying service-based business processes. No wonder a number of middleware vendors have recently added direct support for BPMN to their process engines [44].

On the basis of BPEL, a number of extensions have been created to bring aspects that are not present in the original BPEL. For instance, WS-BPEL Extension for People (BPEL4People) [54] aims to introduce human activities to BPEL in order to let it define general purpose business processes rather that orchestrations of web services. Of special interest for us is an Adaptable Pervasive Flow Language (APFL [23],[78]) that facilitates the definition of highly adaptable business processes operating in dynamic execution environments. In addition to conventional BPEL activities, the language introduces internal activities, human activities, context events and abstract activities (i.e., activities whose final specification is postponed to run time).

2.2 Automated Service Composition

Despite the existence of numerous standard languages for defining service composition, this task, when performed manually, tends to be extremely time-consuming, error-prone and thus costly. A considerable amount of

effort in service computing community aims at creating tools that automate this process.

Figure 2.4: Automated service composition

The conceptual model of automated service composition is shown in Fig. 2.4. The *composer* automatically derives *executable composition* from *service descriptions* and *composition requirements*. For different solutions, the structure of the inputs and output of the composer may widely vary. Service description normally includes specifications of service interface and/or protocol. They are sometimes supplemented with additional annotations, e.g., semantic annotations [64], non-functional property annotations [131], etc. As a rule, custom annotations enable richer composition requirements [17]. The composition requirements express the user's expectations about the service composition. Although they can also be quite diverse, to enable automated reasoning they must be expressed in terms of service descriptions. In the literature, two types of composition requirements are often distinguished: *control-flow* requirements and *data-flow* requirements [11]. Control-flow requirements impose restrictions on the order of message exchange within a composition. Data-flow requirements regulate the handling of message data. For many approaches, it is not possible to clearly separate these two since rules on data manipulation may affect the control flow and vice versa [79]. Finally, the outcome of the composer may be either orchestration (in the vast majority of cases) or choreography.

For the last decade, it has been repeatedly shown that the creation of

a composer that could deal with realistic and general enough service composition tasks is an extremely complex research problem. Let us outline some of the major issues that an industry-ready automated composition solution should address. First of all, real services may be stateful rather than atomic and are described by complex workflows (e.g., following the recursive model of BPEL, some of the components can themselves be service orchestrations with complex behaviour). Consequently, stateful services may be asynchronous, nondeterministic (with unpredictable outcome of some operations) and partially observable (the full real status of a service is hidden and only partially accessible through interactions). To this, we have to add a need for an expressive enough requirements language (and, if needed, service annotation language), that reflects both control- and data-flow requirements and can be processed by the composer.

The experts distinguish two different strategies in synthesizing service compositions: *top-down* and *bottom-up* [3]. In top-down approaches, the composition is first defined at a higher level of abstraction (e.g., as a UML model [114], or as an abstract process model with QoS constraints [131]) and then is converted to an executable composition specification. The conversion usually consist of 1) discovery and selection of suitable services and 2) final synthesis based on selected services and the initial abstract definition. In bottom-up approaches, the services are composed based on their definitions (often, extended compared to standard documents like WSDL and Abstract BPEL) and abstract composition goals using an automated reasoner (e.g., AI planning [103]). We remark that sometimes it is not easy to distinguish a strategy used by a certain approach. As an example, an abstract process defining desired interaction model for a composite service can be considered both as a composition requirement and as an abstract model to refine.

The further classification and analysis of automated service composition

solutions is complicated by the existence of numerous classification criteria and by the fact that many approaches are hardly comparable to each other because they differently perceive a composition problem. Several works concerning the classification of automated service compositions approaches can be found in the literature ([3], [106], [7], [75]). Using them, we can enumerate the main classification criteria as follows:

- *Orchestration/Choreography.* In the first case, the goal is to create a centralized orchestrator that fulfills the composition requirements. In the second case, the result is usually a number of orchestrations attached to services that form a choreography fulfilling the composition requirements. It is also worth to mention approaches [9] and [86], that involve the distribution of a centralized orchestrator among participants, thus transforming an orchestration into a choreography. In [9], for instance, to that purpose WS-BPEL processes are converted into attributed graphs and special rules for graph transformation are used to split them into a set of related graphs (distributed pieces of the orchestration). Graph analysis is also used in [86], but the authors also consider dependencies among the parts obtained to minimize communication costs and maximize the throughput of the composition.

- *Static/Dynamic.* Static approaches are supposed to produce composition using statically predefined set of services and composition requirements. On the contrary, dynamic approaches can "monitor" the execution environment and once the need for composition/re-composition emerges, they can derive all necessary input data (service descriptions and composition requirements) automatically. To the best of our knowledge, none of the existing approaches can be named completely dynamic. The point is to what extend an approach is dynamic or static, i. e., how much involvement of the designer it

requires to perform compositions at run time. Since the problem of dynamicity is one of the central in the dissertation, we have to agree that with respect to composition we use terms static/design-time and dynamic/run-time interchangeably.

- *Data-flow/Control-flow.* The criterion shows to what extent data and control aspect can be reflected in composition requirements and resulting composition. It is possible to find polar cases here. On the one hand, there are quite many approaches that ignore the data flow. On the other hand, mashup-like approaches ([72]) provide data integration of information-producing services and cannot work with stateful services. Finally, a large amount of solutions consider to different extent both data and control flows.

- *Requirements expressiveness.* The expressiveness of requirement language determines a range of composition problems that can be encoded with them. For instance, desirable control-flow can be expressed through termination conditions, event handlers, constraints, transactionality requirements etc. Requirements can also consider functional and non-functional properties of the composition.

- *Service Model.* The difference between a service model and real properties of services may substantially affect the applicability of the approach to real composition problems. For example, if services are considered to be atomic the approach will not be able to work with stateful services which is a serious limitation. Other details may include determinism/nondeterminism, synchronicity/asynchronicity etc.

In our survey and consequent high-level comparison, we will try to consider the existing solutions from the perspective of the classification criteria above.

2.2.1 Existing Approaches

To structure the survey, we grouped approaches according to the reasoning mechanism they exploit. From this perspective, we clearly distinguished two groups: approaches that do not exploit AI planning (*non-planning approaches*) and approaches that exploit AI planning (*planning approaches*). Such division is justified by a large amount and diversity of approaches using planning, which by quantity can be compared to all other approaches joined together. Planning approaches transform a composition problem into a planning problem and apply planning algorithms to resolve it. Similarly, non-planning approaches reduce composition problems to other well-known models (e. g., graphs) and apply appropriate techniques to resolve them.

We remark that the presence of numerous approaches to service composition does not allow us to mention all of them here. That is why we primarily chose those that, from our point of view, are more mature, represent the diversity of composition methods and are interesting to be compared to our approach. For the broader survey of other service composition synthesis techniques (especially for model-driven composition, QoS-aware service composition, semantic web composition) we encourage the reader to check out works [3], [106], [97], [38] and others.

Non-planning Approaches

In [13, 14] a **logic-based** approach is proposed. Component services are encoded as finite state machines (FSM) on the basis of their exported behaviours (i.e., protocols). The user specifies a desired behaviour of a composite service as a tree of actions which is, again, transformed into FSM. Analyzing all available services (graphs), the approach figures out if it is possible to build a composite service from available components re-

specting both the specified composite service behaviour and the behaviours of component services. If this is the case, it synthesizes a composition. A further extension of this work [12] allows for more advanced control flow requirements relying on semantic-like annotations of web services in terms of effects on the real world (the world is modeled as a database). It also addresses basic data-flow requirements in terms of data pieces that services can receive and send. The problem of composition is transformed into a Proportional Dynamic Logic (PDL) formula, and logical reasoning is used to derive an orchestrator satisfying it.

A more profound approach based on a graph search algorithm can be found in [57]. Component services and composition requirements are modelled as directed graphs with rich semantic attributes. To make search more efficient, all services stored in the registry are joined into an aggregated graph representing collective behaviour of a service system. Once a goal graph is specified, a search algorithm is used to find solution in the aggregation graph. The implementation also addresses the problems related to pervasive systems, such as dynamic recomposition of a solution in changing conditions and unevenness of resources available to different devices (low performance devices rely on those with high performance for performance-demanding tasks like discovery and composition).

In [29], a **simulation-based** approach is proposed. Services (possibly, nondeterministic) are modelled as transition systems and a composition requirement is represented with a goal transition system sharing operations with component services. The goal transition system represents desirable interface of a composite service. Nondeterministic simulation is used to find a transition system of all possible satisfying paths in an asynchronous product of service transition systems. The possibility to generate the best solutions with respect to non-functional properties is added in [30]. Here, the transitions in service models are additionally given a weight, which

reflects the cost of operation execution. The path with minimal weight is taken as the best.

The approach of [20] provides a way to build *service aggregations* of stateful services. Service aggregation is a special type of composition in which composition requirements are not expressed explicitly. The goal of service aggregation is to build a composite service that will expose all behaviours of a set of component services that are correct with respect to their behaviours and inter-service dependencies. The approach uses a rich model of service contract reflecting both service protocol (encoded with YAWL language [118]) and its ontology-based inputs and outputs. An aggregation algorithm analyzes control and data flow of services and comes up with a contract of a service aggregation.

A **rule-based** service composition system is described in [104]. In this work, services are modeled in terms of rules that specify what input data is necessary for a service to produce certain output data. Having requirements for the inputs and outputs of a composite service and rules for available component services, a rule-based expert system checks whether it is possible to build a corresponding composition. The authors focus mainly on data-flow requirements, while services are considered to be atomic, without observable complex behaviour. This is an example of the approach where control-flow requirements are not explicitly presented but partially defined by data-flow requirements. The fact that one service can be executed only when certain data is received from another service, imposes restrictions on the acceptable order of service executions. Although rules are used instead of semantic web services, the approach is conceptually close to semantic service composition ([117]).

A similar approach of composition through **data integration** is described in [116, 115]. Here services are modelled as data sources and functional dependencies between them. A composition goal is defined as a

desirable output and a set of query templates it has to support. An integration plan is a number of queries to data sources respecting binding patterns. One of the focuses of the work is a solution optimization with respect to the number of queries to data sources by eliminating redundant queries.

In **workflow-based** approaches, services are treated as workflows and advanced techniques for processing workflows are applied. For example, [65] introduces a Transactional Workflow Ontology, in which service workflows and composition workflows are defined, and implements a process engine that can run ontology-based workflows. The composition consists in finding services that match tasks in a goal workflow. The solution is inferred from definitions of a goal workflow and service workflows by a semantic reasoner (DAMLJessKB).

The authors of [84] present an interesting approach to service composition based on **heuristic search**. In this paper, the authors try to overcome the limitations of planning-based algorithms by using heuristic search as a reasoner. Since heuristic search algorithms have quite some limitations themselves, the authors aim to create a new heuristic search algorithm that would better address the problem of service composition. In particular, the authors require that the algorithm allows for parallel control flow, uncertainty in service effects and initial state of the execution environment, alternative control flow for the same problem. Services are modelled as atomic operations with input, output and ontology-based preconditions and effects. Composition requirements are specified as ontology-based initial and goal states and a service domain containing all available services. The solution is a partially ordered set of service invocations.

Planning Approaches

AI planning ([49]) is by far the most popular reasoning mechanism for automated service composition. In general, planning based approaches consist in converting a service composition problem into a planning problem so that the latter can be resolved using well-established planning algorithms. As a rule, service descriptions form a planning domain and composition requirements are transformed into planning goals. Despite all the works below are based on AI planning, the way they exploit it may differ a lot from work to work.

The **logic-based planning** is exploited by the works [80, 87] by McIlraith et al. The approach adopts high-level logic programming language Golog [69] based on situation calculus. The authors show that Golog can be adapted for the purposes of semantic web service composition. The language is extended to allow for generic and customizable programs. The central idea is to write generic Golog programs that encode certain tasks and, upon user's request, can be customized with user constraints and can be bound to available services. The component services are originally described semantically using DARBA Agent Markup Language-Services (DAML-S) [28] and are later translated into situational calculus to be compatible with Golog-based reasoning. The authors also present an extended version of ConGolog (Concurrent Golog) [50] interpreter that implements the novel ideas of their approach and allows for calling real web services. The translation of OWL-S semantic web services into ConGolog programs is proposed in [99].

In [96], the author encodes composition problems as planning problems using Planning Domain Definition Language ([48]). Service interfaces expressed in WSDL are additionally annotated with *semantic* information similar to that of OWL-S. The problem of statefulness of services is solved

by introducing relation that express the "payload" of service operations, i.e., how operations affect the service state. To describe complex composition goals that go beyond reachability, the approach uses Java programs that encode goal logic (although problems with formal verification of compositions may occur). In a similar way, nondeterminism is addressed. if during the execution unpredictable behaviour occurs, the fault handling strategy can be explicitly added to goal-specifying program. The approach introduces a novel idea of using planning in combination with other reasoning techniques.

In [6], the authors perform service composition by dynamically binding services to an abstract process. As such, composition requirements are specified in form of abstract process whose activities have to be further associated with services. To model abstract processes, WS-BPEL is extended with semantic annotations and a new process engine is implemented. The dynamic binding takes into account inter-service dependencies and is realized using planning.

In [113], Sirin et al. present a semi-automated approach to semantic web service composition. In this approach the user firstly has to find a service in a repository that can produce a needed piece of information. From the service semantic description in DAML-S [28] the system "understands" what input information is necessary to run the service. Then it explores the repository for services that can produce such information and let the user choose manually those of them that better fit the user's needs. As such, the system guides the user through the iterative process of service composition that results in a composite service that satisfied the requirements. To automate this method, the authors use a **Hierarchical Task Network (HTN) planning** techniques [46]. In particular, they exploit SHOP2 HTN planner [88]. In [130], all service semantic descriptions (DAML-S processes) from the repository are translated into SHOP2 oper-

ators and methods in order to build an HTN, i.e., a planning domain. The composition requirements are defined through a semantic description of a composite process and its inputs (this description has the same structure as the descriptions of components in the repository). Such a description is further translated into a planning goal. After that, the algorithm delivers all the possible service workflows that satisfy the requirements.

The problem of bridging the gap between high-level user's perception of the composition goal and low-level service descriptions is addressed in [10]. Here, deterministic service protocols are expressed in YAWL. Service operations are additionally associated with abstract capabilities services can provide and a hierarchical structure is defined to express how high-level tasks relate to various service capabilities. Finally, service operation parameters are linked to an ontology. Having a user's need expressed as a task to accomplish, GraphHTN [74] planning engine is used to derive a service composition that satisfies the user's need. Hierarchical structure is used to figure out which capabilities have to be involved and linkage to data ontology provides mapping between service operations parameters.

The work of [35] proposes to model the knowledge about the domain in the form of a state transition system, with transitions associated to service actions of available services. Such a domain reflects how the execution of service actions affects the domain. Being implemented as state transition systems, services can be synchronously joined with the domain to form a large state transition system encoding all possible service orchestrations and their respective effect on the domain. In the approach, the goal is defined as a linear abstract process, where each action is a reachability goal over the domain which can be resolved using standard planning algorithms.

A complete approach to service composition has been gradually developed in [102, 103, 76, 18] by Pistore et al. The approach was proposed in the context of the ASTRO framework ([4]) supporting all activities

related to service composition life cycle (composition synthesis, verification, monitoring and adaptation). Concerning the composition synthesis, the framework provides automated assistance at all phases, from specifying composition requirements to generating WSDL and Executable BPEL specification of a composite service, to deploying it. A service description includes WSDL interface and Abstract BPEL protocol. A procedure for translating Abstract BPEL processes to state transition systems is provided. The formal model carefully reflects such aspects of services as partial observability, nondeterminism and asynchronous behaviour. The planning domain is obtained as asynchronous product of component state transition systems. Control-flow composition requirements are defined as a reachability goal over the states in service protocols and can be additionally extended using CTL temporal logic formulas [43] or the language for extended goals EaGLe [67]. Control-flow requirements are later transformed into a planning goal for the planning domain. The planning problem is resolved using variations [15, 111] of **planning-as-model-checking** technique ([32]), exploiting symbolic model checking ([27]). Once the plan is found, it is translated into an Executable BPEL process. The approach is formally proved to be complete and correct.

The approach of [102, 103, 76, 18] also received support for data handling by planning at the knowledge level ([101]) and by introducing explicit dataflow requirements in the form of DataNet notation ([79]). The techniques proposed have been successfully tested on real case studies ([77]).

2.2.2 Discussion

As we mentioned, there are quite some works concerning the survey and classification of automated service composition approaches (e.g., [3], [106], [97], [75], [66]). However, there is a very limited choice of publication (especially, recent ones) that, in addition to overview, try to compare the

existing approaches and identify the most important open issues to be challenged next (works [66, 75] could be suggested). Concluding this section, we come up with this discussion reflecting our understanding of the most important open issues in the area of automated service composition.

From our point of view, all the approaches presented in our survey share the two main problems, namely 1) detachment from real SOA and 2) low applicability in dynamic SOA.

Detachment from real SOA. Many existing approaches feature formal model that does not adhere to adopted standards and operational semantics of real services and service compositions. As a result, these solutions are usually applicable to a very limited set of real composition problems and cannot be adopted by the industry as a general-purpose composition engine. In other words, for any critical aspects of real service composition there usually exists a whole bunch of solutions that carefully take this aspect into account. Nevertheless, there is hardly a single approach that addresses all or at least a substantial number of such critical aspects, which would give it enough comprehensiveness to deal with a wide range of real composition problems. What is even more disappointing is that possessing conceptually different formal models and using different techniques, approaches targeting different issues are difficult or even impossible to be integrated together.

For example, it is widely recognised that services are often stateful components that feature complex communication protocol. However, many approaches (e.g., [104, 115, 80, 113]) consider services to be atomic operations characterized by input/output and sometimes precondition and effect. Although this restriction is reasonable for some ad-hoc composition problems (e.g., dynamic service binding, data integration of information services), it is too limiting for a wide variety of others.

The same is true with nondeterminism, which is a natural property of

real services. For example, approaches [104, 130, 6, 10] consider services and their protocols to be deterministic.

An important issue is the expressiveness of requirements languages for control flow. The point is that very often requirements languages of existing solutions are not appropriate for capturing real composition problems. In most cases, existing approaches specify control-flow requirements 1) as an abstract process (e.g., [14, 57, 29, 65]) or abstract protocol (e.g., [103]) to be implemented by the service composition or 2) as a rechability goal (e.g., [104, 130, 18]) in component workflows. In the former, the specification of an abstract process model encoding the solution requires exhaustive analysis of available services and deep knowledge of application domain in order to understand the step-by-step strategy which is optimal and correct. This approach also lacks flexibility with respect to changes in service implementations. In the latter, the reachability goals, although do not have drawbacks of workflow-based goals, are frequently too “primitive” to express complex expectations of the customer about the service composition (e.g., such expectation may involve the maintenance of some properties, goal preferences, partially defined operation order etc.)

An important element of service composition requirements that is often neglected even by strong approaches (e.g., [57, 29, 35]) is data-flow requirements.

Finally, we would like to mention such important problem as modelling overhead. Very often, the strength of composition approaches is achieved at the expense of initial modelling overhead such as defining multiple ontologies (e.g., [28, 10, 35]). In this case, the overhead has to either be decreased or distributed among partners, or be highly reusable through the life cycle of the execution environment.

Low applicability in dynamic SOA. Modern SOA tends to be dynamic. Among the dynamic factors we can distinguish volatile context,

dynamic availability of services, constantly evolving business policies etc. As a rule, these factors negatively affect the applicability of automated service composition solutions.

Every time a dynamic change occurs in execution environment, the existing compositions may be broken (e.g., disappearance of a service invalidates all compositions using it). As a result, recomposition may be needed. The sticking point here is how much designer's effort is necessary to adjust composition requirements to new conditions in order to enable recomposition. Indeed, it can be easily observed that although automated service composition significantly reduces the amount of "manual" work to produce a composition, there is still a considerable effort on the modelling side (composition requirements, service annotations, ontologies, goal abstract processes etc.). If dynamic changes happen frequently, the redesign of composition requirements and accompanying specifications may become the main item in the cost of application support.

In this regard, requirements models relying on implementation details of services (e.g., [14, 103, 29, 96, 35]) are very inflexible since any change to service implementation or replacement of one implementation by another is likely to result in invalidation of composition requirements. At the same time, abstract requirement models may also be inflexible. For example, abstract goal process may require considerable redesign in case business policies have changed. Even service unavailability may break the business strategy implemented by an abstract process.

From our point of view, the two aforementioned problems, (i) detachment from real SOA and (ii) low applicability in dynamic SOA, will remain the driving challenges in the development of automated service composition in the near future. That is why the creation of an automated service composition technique addressing them is one of the focuses of this thesis.

2.3 Adaptation of Service-Based Business Processes

Modern enterprise-level SOA-based systems are characterized not only by complex structure but also by increasing dynamicity. The major dynamic factors include changes in service QoS, unavailability of services, exogenous changes in the operational context, changes in business policies etc. Many of these factors may negatively affect the normal operation of the whole system or of some of its components. That is why, the problem of creating business infrastructures that can rapidly and automatically adapt to environmental changes (*adaptability*) in order to facilitate further achievement of business goals is one of the critical issues in enterprise SOA.

Since the scope of adaptation of service-based systems is quite broad, we start with positioning our contribution inside it. First of all, adaptation may take place at different levels of abstraction of service-oriented architecture. These levels (or layers) are differently identified in the literature (e.g., [3, 129]), but normally at least three layers are distinguished: 1) infrastructure layer 2) service layer and 3) process layer. Our adaptation research concerns the last one, which is also the most abstract in this hierarchy. This level comprises mechanisms for coordinated execution of services, which are commonly realized through service-based business processes. Adaptation may further be divided into short-term adaptation (often simply called *adaptation*) and long-term adaptation (also know as *evolution*) [129]. The former implies temporal changes in the system to address a particular problem or exceptional case. The latter implies definitive changes to the system that will change its future operation. In respect of business processes, adaptation stands for changes to a particular process instance while evolution stands for changes to a process model so that they will be propagated to all its future instances. In the rest of this section we will concentrate on the short-term adaptation of business processes. We

also remark that the evolution of business process on the basis of the short-term adaptation approach proposed is considered as one of the principal future steps and is briefly discussed in Chapter 9 (Conclusions and Future Work). Finally, we encourage the reader to consult [3] and [109] for the broader overview and taxonomy of adaptation in service-based systems.

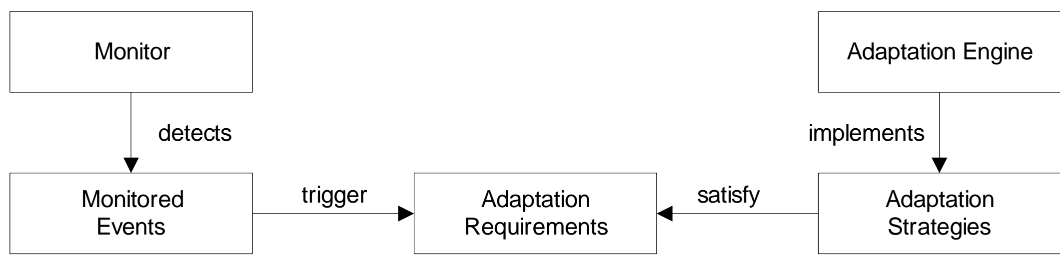


Figure 2.5: Adaptation concepts

Business processes is currently the main tool for specifying complex and structured business activities (and applications) in SOA. That is why their ability to flexibly adapt to various changes in the execution environment is of high importance for enabling adaptable service-based systems. The main concepts related to process adaptation are shown in Fig. 2.5 [3]. The *monitor* is supposed to check if the parameters of the execution environment evolve as expected. Critical violations of expected behaviour happen in the form of *monitored events* detected by the monitor. Monitored events may describe wide range of conditions in the system, from simple infrastructure failures to the violation of QoS properties. Monitored events trigger *adaptation requirements* that express the expectations about the process/system operation and indicate how to improved the situation (e.g., to apply certain modification to a process instance) in order to enable further achievement of the business goal. Adaptation requirements are fulfilled through *adaptation strategies*, which are general techniques for achieving adaptation goals (e.g., service rebinding, process reconfiguration or replanning etc.) Finally, it is the *adaptation engine* who implements

adaptation strategies for every particular situation and set of adaptation requirements.

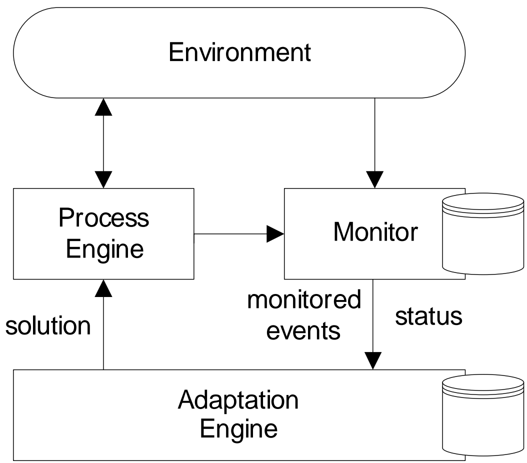


Figure 2.6: Architecture of adaptable system

The general architecture of a system for process adaptation is presented in Fig. 2.6. The *process engine* is responsible for executing business processes. The status of process execution and the status of execution environment is monitored by the *monitor* and constantly checked against requirements violation. As soon as violation is detected, along with the status information, it is reported as a monitored event to the *adaptation engined*. The adaptation engine identifies which of the adaptation requirements are violated and chooses an adaptation strategy to follow. Finally, a solution is derived, sent to the process engine and deployed.

There are quite many classification criteria for process adaptation techniques. The most important of them are as follows:

- *Adaptation Requirements.* Three types of approaches can be distinguished. *Built-in* approaches statically embed adaptation logic into the process specification. The process defined at design time does not change its structure at run time. *Rule-based* approaches use situation-action rules that explicitly indicate actions to be taken in

order to transform a process (for example, certain service rebinding or re-execution) in case a particular situation occurs. *Goal-based* approaches implement adaptation on the basis of abstract goals describing the adaptation objectives. These approaches usually rely on automated reasoning mechanisms such as planning;

- *Autonomicity.* Autonomicity shows how much involvement of the designer is required during the adaptation. Automated approaches can be roughly characterized as semi-automated (require some manual work) and automated (do not require manual work);

- *Adaptation Timing. Reactive* adaptation is where adaptation is undertaken when a problematic situation is reached. *Pro-active* adaptation tries to detect potential problems before they really happen and act proactively to avoid them. *Post-mortem* adaptation is usually associated with situations where the normal process execution cannot be restored and process recovery is applied in order to terminate it with minimal loss;

- *Adaptation strategies.* Different approaches may implement various strategies such as *re-binding*, *re-configuration*, *recovery*, *re-execution* etc. More advanced techniques may dynamically switch between strategies depending on the run-time situation;

- *Environmental Awareness. Environment-Aware* adaptation logic can benefit from the information about the current status of the execution environment to perform adaptation better and more robustly. In certain sense, environmental awareness reflects the dynamicity of the approach. One particularly important type of environmental awareness is context-awareness, where an approach can perform adaptation based on the current critical parameters of the real world.

2.3.1 Existing Approaches

The survey is structured according to the adaptation requirements criterion, i.e., all approaches are divided into built-in, rule-based and goal-based. In the approach descriptions and in the final discussion we also pay attention to other criteria that appeared in the classification above.

Built-in Approaches

Built-in adaptation is the most basic (and chronologically first) type of process adaptation. It consists in statically embedding the adaptation logic into a process specification. The most primitive tool for built-in adaptation is exception handling ([36]). Some business process languages provide their own facilities for that. For instance, in WS-BPEL ([91]) fault, event and compensation handlers can be used to specify sub processes that have to be executed in exceptional situations. What is essential is that even these simple tools allow for a few strategies to be implemented (e.g., rebinding, re-execution with compensation, recovery, etc.)

There are also some works that try to extend standard languages in order to improve their flexibility and robustness. As a rule, such extensions go along with modifications to process engines facilitating them. For instance, in [58] the authors propose to extend WS-BPEL in order to allow for dynamic swapping of participating web service instances (rebinding adaptation strategy). For that purpose, all WS-BPEL communication activities (invoke, receive, reply) are extended with the additional element of selection policy. Before the activity execution the extended engine 1) discovers services that have WSDL port type compatible with the activity, 2) selects one of them according to selection policy (e.g., QoS parameters) and 3) binds it to the activity. The adaptation does not change the process structure. A similar approach to service re-binding based on QoS

properties of services in proposed in [132].

The approach of [78] introduces WS-BPEL extension for context-aware process execution. The central idea is to constantly monitor the context and to introduce specific constructs to WS-BPEL that very the execution according to the current contextual conditions. In particular, these constructions include context handlers (similar to error handlers but triggered by contextual conditions), contextual branching (contextual *if*), contextual process variants with dynamic "jumps" from one variant to another, etc. The broader scope of this work is the Adaptive Pervasive Flow Language (APFL [23]) developed in the context of the ALLOW project ([1]). One of the objectives of APFL is to offer rich adaptation possibilities, such as abstract activities that can be refined to concrete sub processes at run time and constructions for context-aware execution.

Rule-based Approaches

Rule-based approaches introduce situation-action rules that explicitly state the process instance transformations to be undertaken in order to correct it.

Conventional rule-based approaches **explicitly** state a system of rules. For example, rules for dynamic binding of services are proposed in [33]. Service are annotated with service roles and process activities can be associated to these roles to enable service discovery. Binding rules specify how a certain activity can be dynamically bound to a service instance with the same role. Additional binding preferences (e.g., QoS properties) are allowed in the rules. The rule system is completely separated from process specification. We remark that this approach is conceptually very close to that of [58]. The only difference is that in the former the adaptation logic in form of rules is separated from the process itself, whereas the latter embeds it into the process specification. This example shows that sometimes

the border between classification groups is really vague.

In [8], a rule-based approach to defining self-healing processes is proposed. In this work, conventional WS-BPEL processes are accompanied with constraints and adaptation rules. The former determines the expectations from the correct process execution and the latter indicates how to adapt a process instance if constraints are violated. The both additions use their own languages and are completely separated from the WS-BPEL definition. The adaptation rules includes high-level strategy-related constructions (retry, rebind) that can become parts of more complex strategies.

The authors of [68] present a theoretical model of rule-based adaptable application (technology-independent) and later show how it can be implemented in SOA using the Jolie language ([85]). The work contains interesting discussion on when to apply the rules (various types of proactiveness), how to choose the order of the rules to apply, and how to classify the rules. Defined in the detachment from concrete technologies, the framework can also be used with other SOA technologies and beyond.

A number of approaches try to implement adaptability through **process variants**. In this case, critical process sections can have a few predefined variants that are chosen depending on the run-time situation. This class of approaches differs from built-in adaptation since the structure of the process may dramatically change here. At the same time, the choice of process variants is normally realized through a set of rules, which let us consider these approaches to be rule-based.

The basic idea behind the process variants is nicely presented in [5]. The authors introduce worklets as reusable process fragments that can be used to accomplish certain tasks. The main process is allowed to contain special abstract activities (called tasks) that can be replaced with appropriate worklets at run time. The approach allows for nested tasks (i.e., worklets can themselves include tasks). A context model containing a set of dis-

crete values is proposed. Worklet selection for a certain task is based on context-aware situation-worfket rules. The rules can be joined into complex hierarchies to allow for their compact description. The selection is performed by an external component that is detached from process engine.

A comprehensive framework for managing process variants called PROVOP (PROcess Variant by OPtions) is presented in [51, 52]. Here process variants are flexibly defined through a basic variant and a number of elementary transformations (such as activity insertion, deletion, move and modification) that have to be applied to obtain other variants. Moreover, complex relations between variants can be specified (dependency, mutual exclusion etc.) Finally, PROVOP introduces a context model. Elementary transformation operations may then be annotated with context conditions in which they are applicable (which, in fact, makes them similar to transformation rules), thus allowing for context-aware process adaptation. A very similar approach is also proposed in [56].

An example of the use of **aspect-oriented** methodology in process adaptation can be found in [63]. In this paper, the adaptation is performed by identifying the most common types of mismatches and specifying adaptation templates for them. The points in the process where adaptation is needed are equivalent to aspect-oriented joinpoints (they are identifiable through queries to a BPEL specification of a process to adapt) and adaptation actions are equivalent to aspect-oriented advices. Query-advice pairs work very much like rules in rule-based approaches.

Goal-based Approaches

Goal-based approaches are those that express their adaptation needs in form of abstract goals to be achieved. Automated goal-based approaches have to essentially solve two problems: 1) how to derive the adaptation goal from the environment status and 2) how to derive the adaptation

procedure for a given adaptation goal.

Our analysis of the literature on goal-based adaptation approaches revealed only one approach that intentionally targets the problem of process adaptation. It is the SmartPM approach presented in [37]. The initial process is defined as an abstract workflow that is dynamically bound to services. The process is formally modeled with IndiGolog language [108] based on situation calculus. Services are atomic activities that can perform certain abstract tasks under certain conditions and change the execution environment in certain way. At every step of execution, next task is assigned to a service and executed. If the current situation does not allow such assignment, the adaptation is triggered. The adaptation goal is to reach the environmental situation from which the further execution of the main process is possible. The adaptation consists in generating an adaptation process whose execution from the current situation would reach the goal situation. The problem of deriving an adaptation process is reduced to classical planning problem.

2.3.2 Discussion

In this discussion we consider and compare the approaches surveyed above from the perspective of their robustness against dynamic changes in the environment. We remark that dynamic factors of the execution environment are not limited to service availability or changes in QoS properties of services, but also include such aspects as emergence of new and more efficient services, changes to service implementations, volatile context, dynamic changes to business policies etc.

The built-in static approaches ([36, 78]) provide very basic level of robustness. The main drawback of such approaches is that all the critical situation requiring adaptation have to be recognised and addressed at design time. Indeed, such critical situations may be two many to be com-

pletely analysed manually. Moreover, statically defined adaptation process is vulnerable to dynamic factors that go beyond predefined exceptions or critical contextual situation. Indeed, dynamic service availability or constantly changing business policies may corrupt the predefined adaptation procedures and require their complete redesign. Consequently, the cost of support for built-in static adaptation may explode. The other built-in approaches ([58, 132]), though provide run-time rebinding of services, can deal only with very specific dynamic factors (non-functional properties or service unavailability), implement only one strategy (rebinding) that does not allow for complex structural process changes and feature a very simple service model (atomic, synchronous and deterministic services).

The rule-based approaches provide a more robust and flexible adaptation mechanisms compared to built-in approaches. First of all, a system of adaptation rules is normally much more compact (and thus easy to support) compared to an equivalent built-in system. Moreover, rules allow for complex dynamic restructuring of the initial process instance (e.g., [5, 51, 56]) and can be extremely flexible with respect to adaptation timing (e.g., [68]). Rules are also more flexible with respect to policy changes since a change in a policy is likely to affect only a small portion of a rule system (though further verification of the whole changed system may be required).

Unfortunately, the situation-action rules in most of the approaches surveyed above specify concrete (implementation-dependent) "action" part. In other words, adapting actions are mostly specified as concrete subprocesses ([8]) or process transformations dependent on implementation details of concrete services. This makes rule-based approaches vulnerable to such common dynamic factors as service unavailability or modification of service implementations. From this point of view, approaches with abstract rules look better (e.g., in [5], abstractness is achieved through worklets that may

contain abstract tasks).

However, there is still one major drawback that cannot be overcome in rule-based approaches (it can be also attributed to built-in adaptation). Since rules are specified at design time, the designer has to choose a particular adaptation tactic for a certain extraordinary situation (such as "in this situation do this and that"). At run time, it may happen that the tactic chosen is not applicable (e. g., since it requires the usage of a service that became unavailable) but a different tactic for the same problem still exists. Similarly, it may happen that newly emerged services allow for a more efficient tactic rather than the one currently encoded in rules. In such situations, maintaining an up-to-date and consistent system of adaptation rules may become a very time-consuming and error-prone task that requires profound knowledge of the execution environment, as well as advanced supporting tools. Moreover, our evaluation suggests that sophisticated adaptation tactics usually require considerable amount of rules to be specified.

The drawback attributed to built-in and rule-based approaches are mostly overcome by goal-based adaptation. Specifying adaptation needs in form of abstract goals and relying on advanced reasoning mechanisms such as planning, this type of adaptation potentially allows for highly automated solutions that can deal with various kinds of dynamic factors with the minimal involvement of the process designer. For instance, [37] demonstrates the ability to deal with dynamic set of services, can flexibly address changes in business policies and can dynamically identify a suitable adaptation tactic by analyzing available service and their execution and business policies. Being a pioneer in goal-based adaptation, [37] still has some disadvantages, among which we can mention detachment of adaptation framework from real standards, oversimplified service model (stateless synchronous services), limited adaptation strategies (only replanning for precondition

violations is allowed) and lack of nondeterminism in adaptation processes (limitation of classical planning).

The bottom-line is that goal-based approaches, though yet not well-developed, offer high potential for extremely flexible and robust adaptation of business processes. We note that goal-based approached can benefit from advanced automated service composition techniques, where composition requirements are often described as reachability goals. The main issues that have to be addressed are 1) how to deal with realistic services and processes , 2) how to derive abstract adaptation (composition) goals automatically and 3) how to realize various adaptation strategies.

2.4 Problem Statement

In the concluding discussions for Sections 2.2 and 2.3 we have already discussed in detail the main challenges that have to be faced in order to bring the solutions in the respective areas to conceptually new level. Although the challenges related to the automated service composition may look unconnected from those related to process adaptation in dynamic environments, through this dissertation we show that once we have a service composition engine ready to be used with realistic services and in dynamic environments, it can serve as a core for a state-of-the-art approach to dynamic process adaptation.

The first contribution of the thesis is the creation of a state-of-the-art engine for automated composition of services and process fragment that has the following main properties:

- **Realistic services**. The ability of the engine to compose realistic services featuring statefulness, nondeterminism and asynchronicity. The engine has to adhere to existing standards for specifying services and

processes, so that it supports complete composition life cycle: from taking specifications of components and composition requirements as input to delivering a solution in form of executable specification as output;

- **Rich composition requirements**. The engine has to allow for rich control-flow and data-flow requirements, so that realistic composition problems of high complexity can be expressed in it;
- **Abstract requirements**. The composition requirements have to be detached from service implementation to enable 1) the ability to derive them automatically at run time from various abstract models associated to applications and 2) the ability to reuse the same requirements for different or constantly changing service implementations.

We are to demonstrate that the composition engine with the aforementioned properties can successfully address the challenges described in the discussion of Section 2.2.

The second contribution is the state-of-the-art approach to dynamic process adaptation. In particular, we show that a service composition engine with above properties can be used as a core for advanced techniques for dynamic adaptation of service-based business processes. For that purpose, the following issues are to be addressed:

- **Adaptation-enabling Process and Service Definitions**. Conventional process and service definition languages have to be extended in order to have allow for processes with flexible and easily customizable structure. They must also facilitate automatic problem detection and adaptation goal derivation;
- **Adaptation strategies**. There might be multiple ways (adaptation strategies) to modify the original process in order to address run-time

problems. One of the central issues is to identify a compact set of such strategies that will be enough to cover the vast majority of adaptation cases. Moreover, there must be a reasoning mechanism that allows for automatic selection of an adaptation strategy(s) to be used in certain conditions;

- **Automated adaptation life cycle**. In order to enable completely automatic run-time adaptation we have to automate all steps in process adaptation life cycle, which in addition to process detection and strategy selection includes derivation of an adaptation goal, strategy implementation (in our case, using service composition) and solution integration and execution.

From the title of the thesis one can see that our ultimate goal is not only to provide theoretical framework for solving the problems above but also to come up with prototype tools that can demonstrate the applicability of our solutions and serve as a platform for further experiments and evaluation.

Chapter 3

Process Adaptation in Dynamic Environments

Among the key advantages service-oriented paradigm gives to software developers is the possibility to decrease the cost of software development and maintenance while preserving control over software life cycle and quality. One of the key enabling factors for these advantages is the capability of service-oriented applications to flexibly adapt to critical changes in the execution environment, i. e., to modify their behavior and to evolve in order to satisfy new requirements and to fit new situations. This is especially true for the modern SOA, where applications often operate under constantly changing conditions, both in terms of the context and of services, users and providers involved. In such setting, the same application shall operate differently for different contextual situations, deal with the fact that involved services are not known a priori, and be able to dynamically react to unexpected changes. As we already showed in Section 2.3 this task is not at all easy to solve.

In this chapter we introduce the reader to our comprehensive framework for adaptivity of service-based business processes. The framework exploits the concept of process fragments ([39]) as a way to model reusable process knowledge so that service-based applications are delivered in form of

compositions of such fragments, i.e., business processes. The adaptability of the processes is based on our tools for run-time and context-aware composition of fragments (it will be covered in Chapter 4).

The framework allows for business processes that are only partially specified at design time, and that are automatically refined (customized) at run time taking into account the specific execution context. This refinement exploits the available fragments, which are provided by the other actors and systems to describe the services and capabilities that are offered to the process in the specific context. The framework also supports run-time adaptation to unexpected or improbable context changes that may affect the execution of the application. This is achieved through a set of adaptation mechanisms that, if properly combined through adaptation strategies, automatically find solutions to bring the application to a state where the execution can be correctly resumed.

We consider our adaptation framework per se as a significant contribution in the respective research area. However, in the context of the thesis it also plays the role of the main motivator for our research in the area of dynamic service composition. And this is essentially why we decided to start the main part of the thesis from the process adaptation framework rather than from the composition approaches themselves. In our opinion, the adaptation framework presented below delivers a clear idea of what we mean by "dynamic execution environment" and why the ability to compose services/fragments completely automatically and at run time is so important in modern SOA. Moreover, it serves as a platform, where we can examine issues related to the management of composition life cycle in dynamic setting.

We start this chapter with the motivating example from the car logistics domain that is intensively used in many parts of the thesis, from theoretical definitions to the final implementation and evaluation. Then we give a de-

tailed overview of the adaptation approach with the focus on the key issues it addresses. Finally, we define the process modeling language exploited by our solution to model both processes and reusable process fragments.

3.1 Motivating Example

The scenario used throughout this dissertation is based on the operation of the sea port of Bremen, Germany [19], where nearly 2 million new vehicles are handled each year in order to deliver them from manufacturers to retailers. The delivery process of each car (see Figure 3.1) consists of a set of procedures that can be customized according to the car brand, model, retailer-specific requirements, etc. Cars arrive by ship and are unloaded and unpacked at a certain terminal. Once a car is unpacked, it has to be moved to one of the storage areas, depending on the car type (e. g., covered/guarded areas for luxury cars) and on the availability of parking spaces; different storage areas have different parking procedures that need to be followed. The car remains at the storage area until it is ordered by a retailer. Once a car stored is ordered, it continues its way towards the delivery. In particular, the car is treated at dedicated treatment areas (e. g., washing, painting, equipping, repairing) according to the details in the order. When a car is ready to be delivered it is moved to the assigned delivery gate, where it is loaded onto a truck, and eventually delivered to the retailer.

Our goal is to develop a system (the Car Logistic System or CLS) to support the management and operation of the port, where numerous actors (i. e., cars, ships, trucks, treatment areas, etc.) need to cooperate in a synergistic manner respecting their own procedures and business policies. The system needs to deal with the dynamicity of the scenario, both in terms of the variability of the actors' procedures (customizable processes),

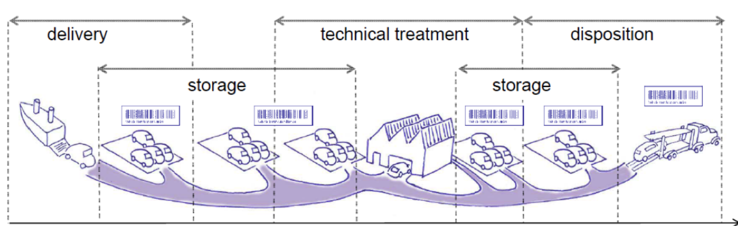


Figure 3.1: Process chain of the car logistics scenario

and of the exogenous context changes affecting its operation.

Customization means that different brands and models of cars should be treated in a similar but customizable way. Moreover new car models, having specific requirements and procedures, have to be able to be easily integrated in the system. Similarly, the system needs to flexibly deal with changes in the procedures of external actors such as ships and trucks. Finally, the system needs to promptly reflect changes in international regulations and laws.

Concerning *context dynamicity*, examples of environment conditions to be taken into account are the unavailability or malfunctioning of the different port facilities, accidental damages of cars and trucks, human errors (e.g., a car is parked in the wrong parking lot). These conditions, although related to specific entities in the domain, may affect the operation of other entities, as shown in the following examples:

- *Vehicle damage:* A car has been unloaded from a ship and must be parked in the storage area. A storage place is assigned to the car and it starts to move there. While moving, the vehicle gets damaged. The system should be able to handle damaged cars and, in case of serious damage, to free the booked parking lot in the storage area;
- *Unavailability of a storage area:* A vehicle is unloaded to the unloading area and using fragments of other partners "organizes" a process of storing itself at some storage area. While the vehicle is moving to

the assigned storage area, the latter is no longer available due to the lack of space or other kind of problems. The system should deal with the car storage, either finding and redirecting the car to a different storage area, or moving the car back to the unloading area to let it wait. Moreover, the system should deal with the same problem for all the cars possessing storage tickets for the unavailable storage area.

3.2 Approach Overview

In this section we present our approach to modeling adaptable and context-aware fragment-based systems that are able to meet all the challenges described in the motivating example. The proposed approach enables the adaptation of fragment-based systems, and is based on the exploitation of context information to continuously adapt the executing processes by appropriately composing available and reusable process fragments. While discussing our approach, we introduce the key ingredients needed for modeling and efficiently operating systems such as the CLS.

3.2.1 Application model

The system operation is modeled through a set of *entities* (e.g., ships, cars, trucks, etc.) (as depicted in Fig. 3.2), each specifying its behavior through a *business process*. Unlike traditional system specifications, where business processes are static descriptions of the expected run-time operation, our approach allows to define dynamic business processes that are refined at run time according to the current status of the system.

The underline idea is that entities can join the system dynamically, publish their functionalities through a set of process fragments that can be used by other entities to interoperate, discover fragments offered by the other entities, and use them to automatically refine their own business

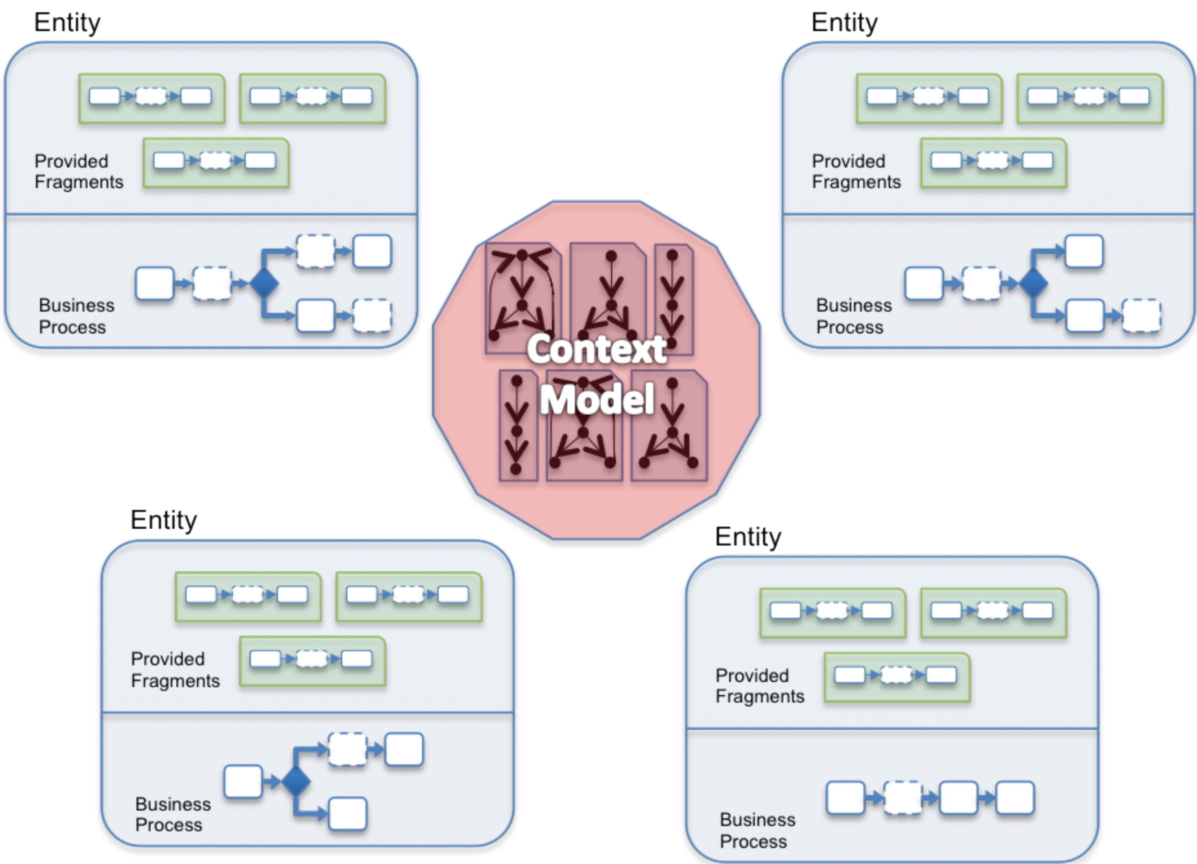


Figure 3.2: Artifacts of application model

processes. For instance, within the CLS, whenever a *ship* approaches the harbor, it discovers the fragments provided by the *landing manager* and by the *gates*. These fragments model the harbor-specific procedures and regulations that the ship should execute in order to land. Different fragments may be provided by different gates and for different ship types. Similarly, the ship will publish its own fragments implementing the procedures to be followed for the unloading of cars.

Another important feature of the proposed framework is the possibility of leaving the handling of extraordinary/improbable situations to run time instead of analyzing all the extraordinary situations at design time and embedding the corresponding recovery activities in the business process.

This kind of modeling extremely simplifies the specification of business processes that have to operate in dynamic environments, since the developer does not need to think about and specify all the possible alternatives (with respect to context changes, availability of functionalities, improbable events). It also efficiently copes with the fact that proper handling of extraordinary situation is not always doable at design time (e.g., in case run-time information such as a set of fragments currently available or the current contextual situation is needed to properly react to dynamic changes).

These dynamic features offered by the framework rely on a shared *context model*, describing the operational environment of the system. The context is defined through a set of *context properties*, each describing a particular aspect of the system domain (e.g., current location of a car, status of a car, availability of a storage area). A context property may evolve as an effect of the execution of a fragment activity, which corresponds to the "normal" behavior of the domain (e.g., current location of car may change from *unpacking area* to *storage area A* as a result of the execution of movement fragment), but also as a result of exogenous changes (e.g., car status changes from *ok* to *nok*). A *context configuration* is a snapshot of the context at a specific time, capturing the current status of all its context properties.

To better explain the idea behind the context, let us consider some of the context properties that may be defined in the scope of CLS. For instance, the *CarLocation* diagram (depicted in Fig. 3.3) captures how the car location can change over time. Initially, the car is on the ship. The car process aims to unload the car to the unpacking area and move it to the storage. The treatment location is where the car can be repaired. Similarly, the *CarStatus* diagram represents car operability status. An example of an exogenous change could be where the car status changes from *ok* to *nok*

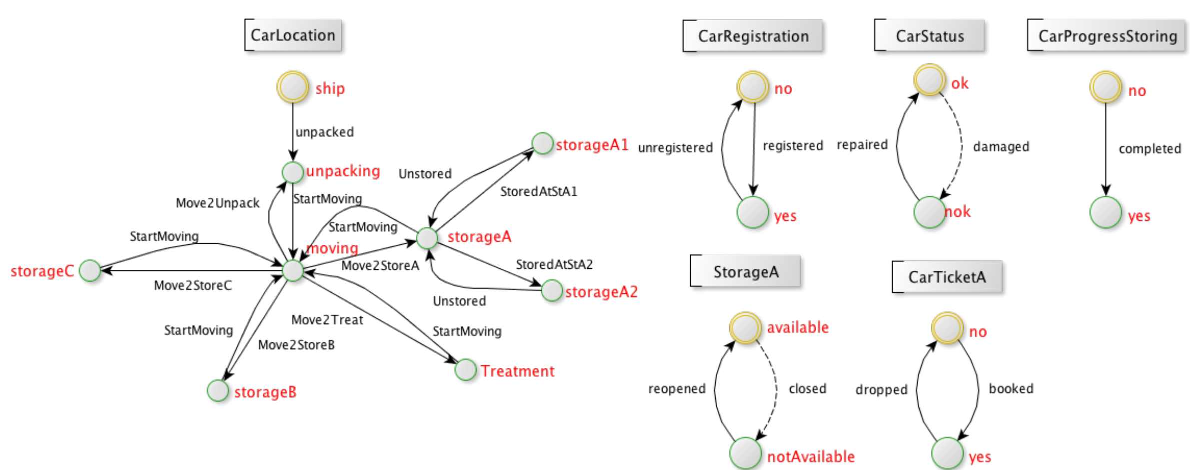


Figure 3.3: Examples of context properties in car logistics scenario

by the exogenous event *damaged*.

Business processes and fragments are modeled as *Adaptable Pervasive Flows* (APFs) [23, 53], an extension of traditional workflow languages (e.g., BPEL) which makes them suitable for adaptation and execution in dynamic pervasive environments. In addition to the classical workflow language constructs (e.g., input, output, data manipulation activities, complex control flow constructs), our edition of APFs adds the possibility to relate the process execution to the system context by annotating activities with *preconditions*, *effects* and *compensations*. *Preconditions* constrain the activity execution to specific context configurations, and in our framework are used to catch violations in the expected behavior and trigger run-time adaptation. *Effects* model the expected impact of the activity on the system context, and are used to automatically reason on the consequences of fragment/process execution. Finally, *compensations* indicate the goal over the context to be reached in case we want to compensate the effect of an activity already executed.

Consider for instance the precondition *P1: CarStatus=ok and CarRegistration=no* in the *Registration Reply* activity of Fig. 3.4 (label 1B). The

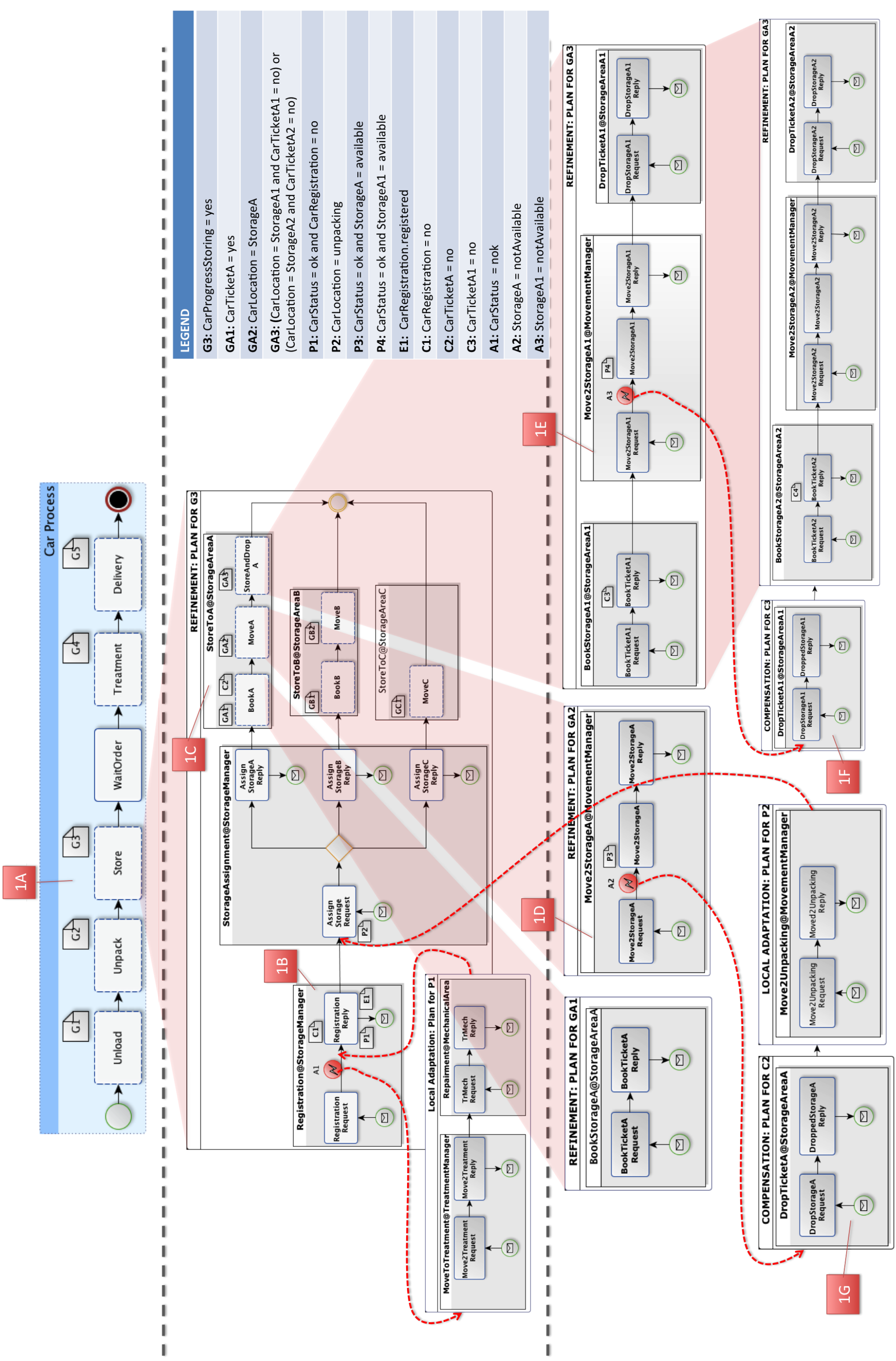


Figure 3.4: Adaptation mechanisms in car logistics scenario

Storage Manager, the provider of the fragment, specifies through this condition that the activity can be executed only if the car is not yet registered and is not damaged. The same activity, is annotated with the effect *E1: CarRegistration.registered*, meaning that the expected impact of this activity is to make the system context evolve to a configuration where property *CarRegistration* is in state *yes*. Finally, it is also annotated with the compensation goal that has to be fulfilled every time adaptation requires to rollback the process instance and it has already been successfully executed. It is the compensation goal *C1: CarRegistration=no*.

Finally, in order to have dynamically customizable processes, we extended the APFL language with constructs enabling the customization and adaptation of process fragments. In particular, we introduced the possibility of specifying *abstract activities* within fragments. An abstract activity is defined at design time in terms of an abstract context-based *goal* it is supposed to achieve. It is expressed as context configurations to be reached, and is automatically refined at run time to an executable process according to the goal to be reached. Being performed at run time, the procedure of refinement can benefit from the run-time information such as the set of available fragments and the current context configuration.

For instance, the abstract activity *Store* of the car process model in Fig. 3.4 (label 1A), aiming at storing the car in a storage area, is annotated with the goal *G3: CarProgressStoring=yes*. At run time, a specific fragment composition will be generated to achieve this goal taking into account the characteristics of the car, the status of the storage areas, and the available fragments for the car storing.

3.2.2 Adaptation Mechanisms

In this section we present different adaptation mechanisms that can be used to handle the dynamicity of context-aware pervasive systems. Our adapta-

tion framework can deal with two different adaptation needs: the need for refining an abstract activity within a process instance, and the violation of the context precondition of an activity that has to be executed. In the former case, the problem is resolved by providing a refinement process for an abstract activity. In the latter case, the aim of adaptation is to resolve the violation by bringing the system to a context state where the process execution can be resumed.

Refinement mechanism

The *refinement mechanism* is triggered whenever an abstract activity in a process instance needs to be refined. The aim of this mechanism is to automatically compose available process fragments taking into account the goal associated to the abstract activity and the current context configuration. The result of the refinement is an executable process that composes a set of fragments provided by other entities in the system and, if executed, fulfills the goal of the abstract activity. As we mentioned, the advantage of performing adaptation in general and refinement in particular at run time is twofold: available fragments are not always known at design time (e. g., a truck arriving at the delivery area may provide its own loading fragment), and the correct refinement may strongly depend on the current execution context (e.g., a storage area may be full and thus its fragments are not usable).

Consider, for instance, the abstract activity *Store* of the main car process in Fig. 3.4 (label 1A). During the execution the activity is automatically refined and composes five available fragments (i.e., Registration, StorageAssignment, StoreToA, StoreToB and StoreToC) provided by different entities (i.e, *Storage Manager*, *Storage Area A*, *Storage Area B*, and *Storage Area C*). The refinement obtained is injected in the car process instance that can continue its execution and achieve its goal.

Composed fragments may also contain abstract activities which requires further refinements during the process execution. The result of this incremental refinement is a multi-layer process execution model (see Fig. 3.4), where the top layer is the initial process of the entity and intermediate layers correspond to incremental refinements.

Local Adaptation Mechanism

Local adaptation aims at identifying a solution that lets the process engine to resume the execution of a process that faulted due to precondition violation from the activity where the violation occurred. To achieve this, a composition of fragments is generated with the goal to bring the system to the situation where the precondition is not violated anymore. After its execution, the execution of the main process can be resumed.

As an example, consider adaptation *A1* of Fig. 3.4 (label 1B). The car process is ready to execute the *Registration Reply* activity of the *Registration* fragment, however the car gets damaged and the precondition *P1* of the activity is not valid. The aim of local adaptation in this case is to repair the car (i.e., precondition *CarStatus=ok* must hold). It is achieved by composing two fragments that allow us to move the car to the treatment station (*MoveToTreatment*) and to repair it (*Repair*). After executing the local adaptation process, the car process instance can resume the execution of the original process.

Compensation Mechanism

The *compensation mechanism* can be used to dynamically compute a compensation process for a specific activity. The compensation process is a composition of fragments specifically selected for the current context and whose execution fulfills the compensation goal.

The advantage of specifying activity compensation as a goal on the

context, rather than explicitly declaring the activities to be executed (e.g., as it is done in WS-BPEL), are the same as with the adaptation in general: in this case it is possible to dynamically compute the compensation process taking into account the current status of the execution environment.

Consider for instance the compensation of the *BookA* activity of the *StoreToA* fragment provided by *Storage Area A* in Fig. 3.4 (lable 1C). The compensation goal *C2* associated to the activity requires that a context configuration where there are no places booked for the car in the storage area is reached. In our case the activity needs to be compensated after its completion and the generated compensation process requires that the ticket for the storage area is dropped.

3.2.3 Adaptation Strategies

When different adaptation mechanisms are combined and executed in a precise order, *adaptation strategies* are realized. They are able to deal with complex adaptation needs that cannot be addressed by applying adaptation mechanism in isolation. An example is the case where a violation of an activity precondition cannot be resolved with local adaptation (e. g., there is no way of making the storage area A1 available for adaptation need *A3* of Fig.3.4 (label 1E). Another example is the failure of an abstract activity refinement, caused by unavailability of fragments to be composed to fulfill the goal within a specific execution context.

Our framework provides different ways of combining adaptation mechanisms. A first possibility is a *one shot adaptation*, where the different adaptation mechanisms are combined for a single adaptation problem, and a comprehensive solution is searched for and, if found, executed. Another possibility is *incremental adaptation*, where each adaptation mechanism in the strategy is called and the resulting adaptation process is executed before applying the next adaptation mechanism. This interleaving of adap-

tation and execution makes it possible tailor each adaptation to the specific execution context, but has the main drawback of not knowing in advance whether the whole strategy can be completely executed (though, a solution to this problem may include execution simulation).

In the following we present some adaptation strategies that we have identified and that resulted to be very useful in our scenario. All the patterns can be implemented through one-shot or incremental adaptation, and, if needed, other patterns can be easily added to the framework.

The **rerefinement** strategy can be applied whenever a faulted activity belongs to the refinement of an abstract activity. The aim of this strategy is to compensate all the activities of the refinement labelled with a compensation goal and that have been already executed (through compensation mechanism) and to compute a new refinement (through refinement mechanism) that satisfies the goal of the abstract activity and takes into account the new environmental conditions. This strategy is used in the scenario of Fig. 3.4 to resolve the adaptation need *A3* (label 1E), where the storage area A1 becomes unavailable. In this specific execution context, the rerefinement of the abstract activity *StoreAndDropA* requires to compensate the activities *BookTicketA1Reply* of the *BookStorageA1* fragment of entity *Storage Area A1* by dropping the ticket and then to recompute a fragment composition that, taking into account the current storage availability, allows the car to be parked in Storage Area A2 (see Fig. 3.4 (label 1F)).

The *backward adaptation* strategy aims at bringing back the process instance to some specific point in the process from which, given the new context configuration, a different execution decision may be taken. The easiest way to exemplify this situation is where at some branching point of the process you (or a fragment under execution) decides which branch to enter. Afterwards, it may happen that the further execution in the branch

chosen is not possible, nor the aforementioned strategies can help to change the situation for better. The radical solution could be to compensate what has already been done in this branch and to jump back to the point where you took a decision on the branch (so called *decision point*) in hope that a different branch will be taken. This strategy requires the compensation of all the activities that need to be rollebacked (compensation mechanism), and for bringing the context to a state where the precondition of the activity next to the decision point is satisfied (local adaptation). In this case, the main process execution restarts form a decision point. One overhead of this strategy is that we have to manually designate the decision point, which become another component of fragment annotation. This strategy is used in our scenario to deal with adaptation *A2* (see Fig. 3.4 (label 1D)), where the storage area A is no longer available but a parking ticket has already been booked for the car. In this case, where neither local adaptation nor rerefinement would work, a successfull strategy could be to bring back the execution to an activity that can potentially make a different decision about the storage area assigned to the car (i.e., *AssignStorageRequest* in *StorageAssignment* fragment). To implement this strategy there is the need for compensating the *BookA* abstract activity by dropping the ticket (i.e., *DropTicketA* fragment) and of making the precondition *P2* valid (i.e., executing *Move2Unpacking* fragment, see Figure 3.4 (label 1G)).

We remark that other adaptation strategies (or even mechanisms) can be defined within our formal framework. Here we discussed only those we needed in the scenario (they were also the most intuitive). New strategies can be defined by composing the adaptation mechanisms in a different way or by combining sub-strategies. To give an example, a possible strategy could be to search for a local adaptation, and, in case no solution is found, try backward adaptation within the same fragment composition, then apply the rerefinement mechanism, then recursively apply backward

and rerefinement mechanism, moving up in the hierarchy of execution layers, till the upper layer, which is the process instance itself. A completely different strategy could be to search for alternative solutions in parallel and then choose the best solution according to a set of predefined metrics (e.g., number of activities to be performed, impact on the process structure, impact on the context configuration).

3.3 Adaptable Pervasive Flows and APFL

Adaptable Pervasive Flows (APFs) [53, 23] have been proposed as an extension of traditional workflow concept in order to make workflows flexible enough to be used in pervasive execution environments. One of the main requirements to APFs is the ability to dynamically (at run time) modify their structure in order to adapt to changes in the execution environment. To make it possible, certain changes have to be done both to the language for specifying APFs and to the execution engine that executes them.

In particular, in order to allow for the changes based on the status of the execution environment, the process specification has to contain some information that links process structure and its particular activities to the execution environment (e.g., context), so that the adaptation tools can automatically derive the changes to the process instance required by the current context. The structure of such process has to be flexible enough to integrate changes "on the fly". Correspondingly, the execution engine has to additionally be equipped with tools that monitor the status of the context and tools that allow for dynamic changes to the running process instances.

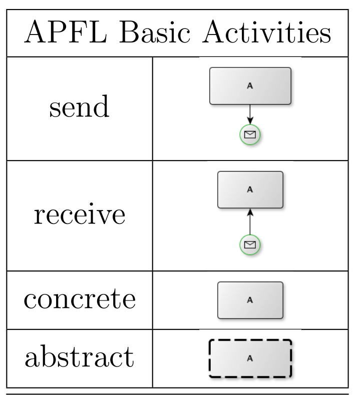
In our research on process adaptation we needed a process model that possessed the qualities mentioned above but stayed adhered to service-based systems. That is why we adopted and modified the Adaptable Per-

vasive Flow Language (AFPL [23]). The APFL has originally been introduced to model APFs and is based on WS-BPEL. It extends the latter in order to take into account the aspects related to pervasive applications. In particular, APFL distinguishes basic activities into two groups: *concrete activities* and *abstract activities*. Concrete activities include all activities for communicating with services (e.g., REPLY, RECEIVE) plus some other activities that go beyond service communication (e.g., internal data manipulation or human operation). An abstract activities are non-executable activities that abstractly define some tasks that have to be performed at certain point in the process. The idea is that the proper implementation of an abstract activity can be derived and assigned to it dynamically at run time. The implementation can be derived automatically using the abstract task specification attached to the abstract activity, and can take into account the most recent information about the execution conditions. As we already showed in Section 3.2, abstract activities, while simple in understanding, bring unprecedented level of flexibility to process structure. The set of structured activities included in APFL mostly repeats those used in WS-BPEL. The further details on the APFL can be found in [78].

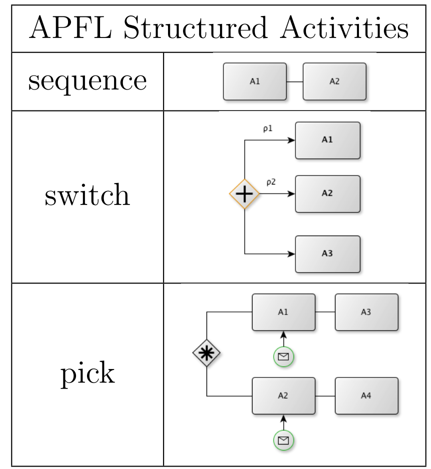
In this paper, we consider a simplified version of APFL, that is depicted in Table 3.1. Among the basic activities are SEND, RECEIVE for communicating with services, CONCRETE to model any kind of internal activity (e.g., internal processing, or human operations) and ABSTRACT for any abstract activity. The structured activities include SEQUENCE, SWITCH,WHILE and PICK, whose semantics is the same as in WS-BPEL. As we will show in the next section, we equip processes and process fragments with context-based annotations to enable context-based adaptation mechanisms. Moreover, since we do not consider the data aspect in our adaptatino-related research, we require that all conditions used in process specification (in SWITCH, WHILE) are expressed as context formulas. The

further details on this will be given in the next chapter.

Table 3.1: Basic and structured activities of APFL



APFL Basic Activities	
send	A
receive	A
concrete	A
abstract	A



APFL Structured Activities	
sequence	A1 A2
switch	p1 A1 p2 A2 A3
pick	A1 A3 A2 A4

Our edition of APFL language is used throughout the thesis to specify both executable processes attached to entities within the CLS and to specify process fragments that are used to advertise the functionality provided by entities. In fact, process and fragments can be compared to executable and abstract WS-BPEL specifications respectively.

3.4 Discussion

From the description of the scenario and adaptation approach it can eventually be observed that in extremely volatile environments the predefined solutions may often be inefficient and, what is more frustrating, even erroneous. Since dynamic changes in the environment can occur at any time, the general intuition is that the closer to the execution point a process is defined the more likely its execution will be successful. Indeed, a process defined immediately before the execution can take into account the most up-to-date information about the environment and, consequently, it becomes less probable that critical changes causing process failure will happen before/during the execution. The attempt to include predefined

"exception handlers" for all extraordinary cases does not work since 1) such cases may be too many and 2) not all extraordinary cases may be predicted (e. g., certain changes in fragment specification).

The conclusion is that an affordable solution can be provided only when processes are refined and, if needed, repaired at run time. However, such mechanisms can hardly be realized by "manual" composition and the automated techniques are needed. So we come to the need of an automated engine for fragment composition that could be integrated with the adaptation framework above. The complete automation of the composition process essentially implies that also the composition requirements have to be derived automatically, by examining the current situation in the system and understanding the adaptation needs. Such examination is likely to be performed at the level of the context and contextual annotations of fragments. As a result, the composition requirements will originally be expressed at the abstract level (e.g., to bring the "Car Status" property to state "ok"), which requires that composition engine is able to deal with such context-based requirements.

Our composition engine is presented in the next chapter. In Chapter 5 we show that it is powerful enough to support all types of adaptation mechanisms and strategies introduced by the adaptation framework. In there, we also discuss the details of how the composition engine integrates with the adaptation framework and, most importantly, what are the issues of composition execution in dynamic environment.

Chapter 4

Context-Aware Composition of Fragments

This chapter is devoted to the fragment composition approach inspired by the process adaptation framework introduced in Chapter 3. The central novel idea of the approach is the use of explicit context model as a way to conceptually describe operational semantics of fragments and to express composition requirements detached from service implementations. The chapter defines the formal model of composition and covers all phases of fragment composition, from representing their specifications with the elements of the model to building a planning domain to resolving the formal composition problem with a planning algorithm. We start with the approach overview (Section 4.1), then introduce the elements of the formal model (Section 4.2) and, using these elements, define the problem of fragment composition (Section 4.3). Afterwards we show how a formal composition problem can be converted into a planning problem (Section 4.4). Finally, the ad-hoc planning algorithm is presented and proved to be correct and complete (Section 4.5).

4.1 Overview

The overview of our fragment composition model is represented in Fig. 4.1. It is built around the explicit model of the execution context, which is a collection of *context properties*. Each context property models some aspect of the application domain that is relevant for a particular composition problem. For example, in our motivating case study context properties might be car location, car status, storage availability etc.

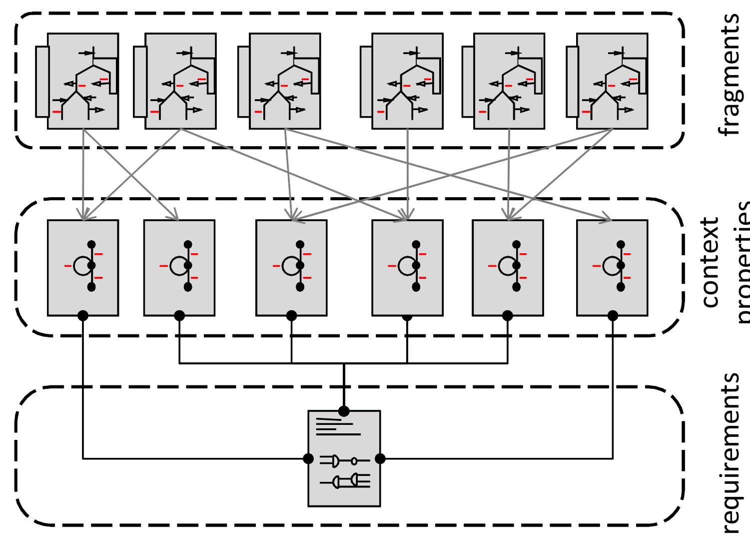


Figure 4.1: Fragment composition model

Each property may have complex behaviour (e. g., the car location and status may change over time). Context property behaviour is captured by its state diagram, which defines all possible property states and transitions between them. In fact, the transitions correspond to activities that can be performed over the context property (e.g., the car status may change as a result of repair) and to the external events affecting it (e.g., car can get damaged). We remark that our model of context properties is conceptually close to the notion of business process artifacts with behaviour ([89]). However, in our approach they are used to reason on how a certain objec-

tive can be achieved through fragment execution, rather than for process modeling purposes.

To link fragments defined in APFL (or, potentially, in any other similar language), we annotate fragment descriptions with context-related information. In this way we implicitly define mapping between the execution of fragment activities and the state of context properties. As we mentioned in Section 3.2, fragment activity may be annotated with *effect*, *precondition* and *goals* (only for abstract activities). The aforementioned fragment annotations are usually quite intuitive since they reflect the functional properties of fragments from the perspective of the application domain.

In order to define composition requirements on abstract level, we define them as reachability goals for context property states rather than fragment states (e.g., the car status have to be *ok*). We remark that the requirements language can be extended with much more sophisticated constructs (e.g., see [17]), but since our primarily objective in this work was the practical implementation and demonstration of concepts we reduced the requirements language to simple reachability of context states (for more complex composition requirements see Chapter 6).

The core idea of our fragment composition model is that, while fragment execution is closely related to the changes in context properties, the modeling of the latter does not depend on a particular fragment implementation. As such, by expressing composition requirements at the level of context properties on the one side, and by relating fragment execution to context properties on the other side, we create a composition framework in which composition requirements, though detached from fragment implementations, can always be automatically grounded on them.

In order to use abstract requirements with particular fragments, we have to restate them in terms of this fragments. Ans this is exactly what we mean by grounding. It is easy to understand that the same abstract

requirements will be grounded differently for different fragment implementations. However, using fragment annotations the procedure of grounding can be completely automated so that the requirements can be dynamically adjusted to certain concrete fragments.

It is worth to notice that in this way it becomes easy to modify a scenario to account for different fragment implementations: it is enough that new fragments are properly annotated, while it is not necessary to change context property models nor composition requirements.

As we will show in Section 4.4, once context properties and composition requirements are specified and component fragments are properly annotated, the whole set of these specifications can be converted into a planning problem which is then resolved using planning algorithms.

Another important point is that using our fragment composition engine, all the aforementioned adaptation mechanisms and thus strategies can be realized (for details see Chapter 5).

4.2 Composition Model Elements

In this section we formally define all elements of our composition framework. One of the key issues addressed is how real APFL fragment specifications correlate with the fragment model used in our approach. All elements are accompanied with examples from the CLS scenario.

4.2.1 Context

We model the context as a set of context properties. Context property behaviour is described by a state transition system that contains all possible states of the context property and transitions between them. Each transition is labeled with a context event. Formally:

Definition 1 (Context Property)**.** *Context property is a state transition*

system $p = \langle L, l^0, E, T \rangle$, *where:*

- L *is a set of context states and* $l^0 \in L$ *is the initial state;*
- E *is a set of context property events;*
- $T \subseteq L \times E \times L$ *is a transition relation.*

In our context model, context events are usually triggered by fragment execution (we sometimes call them controlled events). However, some events are not controlled by fragments and are somehow external (or *exogenous*) to the system. This difference is demonstrated in the following example.

Example 1 (Context Properties)**.** In our motivating example, one context property considered is the "Car Status" (Fig. 4.2), which is attached to any car entity operating within the scenario. It includes two states: *ok* corresponds to operable car and *nok* corresponds to non-operable car. An exogenous transition *(ok, damaged, nok)* models a situation where the car gets damaged (of course, there is no service that intentionally breaks the car, that is why the corresponding event is exogenous). On the contrary, a controlled transition *(nok, repaired, ok)* models a situation where the car is repaired.

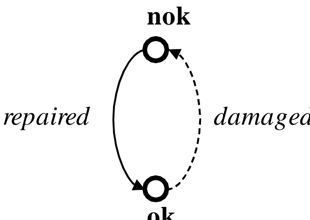


Figure 4.2: Context property for car status

Examples of other types of context properties within the car logistics scenario are given in Fig. 3.3).

Later in this chapter we will show that our composition approach ignores exogenous events at the composition phase. However, in the next chapter, we provide mechanisms for handling such events during the composition execution (i.e., outside the composition engine).

Since the overall context may be quite complex, it can be defined as a set of context properties (what is essentially demonstrated in Fig. 3.3). In this case we require that evolutions of context properties within the same context do not explicitly correlate, i. e., they feature mutually disjoint sets of context events. Formally, the *context* is defined as follows:

Definition 2 (Context). *A context is a set of context properties* $C = \{p_1, p_2, \ldots, p_n\}$ *such that* $p_i = \langle L_i, l_i^0, E_i, T_i \rangle$ *for all* $i \in [1, n]$ *and for any two constituent context properties* $p_i, p_j \in C$ *sets of events do not intersect (i.e.,* $E_i \cap E_j = \emptyset$*). In this case, the current state of the context is determined by current states of all its constituent context properties, so that the initial context state is* $l_C^0 = (l_1^0, l_2^0, \ldots, l_n^0)$ *and the set of all context states is* $L_C = \prod_{i=1}^{n} L_i$*. We additionally introduce a set of context events* $E_C = \bigcup_{i=1}^{n} E_i$*.*

In order to be able to succinctly specify groups of context states we use *context formulas* which are disjunctions of conjunctions over states of context properties belonging to some context:

Definition 3 (Context Formula). *Let* $C = \{p_1, p_2, \ldots, p_n\}$ *be a context such that* $p_k = \langle L_k, l_k^0, E_k, T_k \rangle$ *for all* $k \in [1, n]$*. A state formula for* C *is a propositional formula* $\bigvee_i \bigwedge_j l_{ij}$*, where* $l_{ij} \in \bigcup_{k=1}^{n} L_k$*.*

The space of all context formulas of context C is denoted as R_C. In order to define the satisfaction of context formulas we introduce the notion of context state projection:

Definition 4 (Context State Projection). *Let* $C = \{p_1, p_2, \ldots, p_n\}$ *be a context such that* $p_i = \langle L_i, l_i^0, E_i, T_i \rangle$ *for all* $i \in [1, n]$ *and let* $l =$

$(l_1, \ldots, l_j, \ldots, l_n) \in L_C$ be one of its states. Projection of state l onto context property p_j is defined as follows:

$$l \downarrow_{p_j} = l_j$$

The *satisfaction of the context formula* by context states is defined as follows:

Definition 5 (Context Formula Satisfaction)**.** *Let $C = \{p_1, p_2, \ldots, p_n\}$ be a context such that $p_i = \langle L_i, l_i^0, E_i, T_i \rangle$ for all $i \in [1, n]$. Let $\rho, \rho_1, \rho_2 \in R_C$ be context formulas over C. Context state $l \in L_C$ satisfies ρ (denoted $l \models \rho$), if and only if one of the following holds:*

- *$\rho = \top$;*
- *$\rho \in L_i$, $i \in [1, n]$ and $l \downarrow_{\Sigma_j} = \rho$;*
- *$\rho = \rho_1 \vee \rho_2$, such that $s \models \rho_1$ or $s \models \rho_2$;*
- *$\rho = \rho_1 \wedge \rho_2$, such that $s \models \rho_1$ and $s \models \rho_2$.*

A context formula can be associated with a set of states it satisfies. As it will become clear in future sections, we adhere to formulas in form of disjunction of conjunctions in order to be able to properly handle abstract activities within our formal model. Nevertheless, we remark that with this restriction we still have enough flexibility to identify any subset of context states with a single formula (indeed, any context diagram state can be identified with a conjunction, while any set of states is a disjunction of corresponding conjunctions). The exception is an empty set of states, which, however, can hardly be useful in our model. We also remark that each conjunction may have no more than one state per context property, otherwise it would identify an empty set (a single context property cannot be in two states at a time).

For the future use, we define the *applicability of context event* on the context as follows:

Definition 6 (Context Event Applicability)**.** *Let $C = \{p_1, p_2, \ldots, p_n\}$ be a context, such that $p_i = \langle L_i, l_i^0, E_i, T_i \rangle$ for all $i \in [1, n]$ and its set of context states is L_C, and its set of context events is E_C. Event $e \in E_C$ is applicable on state $l \in L_C$ (denoted $App_C(e, l)$) if $\exists i \in [1, n] : \exists (l \downarrow_{p_i}, e, l') \in T_i$.*

Informally, the event is applicable on the context state if there exists a constituent context property such that this event is applicable on its state corresponding to the current context state.

4.2.2 Annotated Fragments

In our framework we use a unified model for both fragments and processes (which often represent the composition of fragments), and uniformly use the term of *fragment* for both of them. We model fragments as state transition systems where transitions are labelled with two different types of actions : controllable and uncontrollable. Controllable actions are used to model process activities that do not depend on external actors (e. g., SEND or CONCRETE). Uncontrollable actions model activities whose execution depends on external actors (e. g., RECEIVE or PICK). Finally, the both types of actions are used to model ABSTRACT activities (the details are discussed later in this section). The distinction between controllable and uncontrollable actions is crucial for proper handling of the asynchronicity of fragments behaviour in fragment composition. The fragment STS is formally defined as follows:

Definition 7 (Fragment)**.** *A fragment is a deterministic state transition system $f = \langle \mathcal{S}, s^0, \mathcal{I}, \mathcal{O}, \mathcal{R} \rangle$, where*

- *$\mathcal{S}$ is the set of states and $s^0 \subseteq \mathcal{S}$ is the initial state;*
- *$\mathcal{I}$ and $\mathcal{O}$ are sets of controllable and uncontrollable actions such that $\mathcal{I} \cap \mathcal{O} = \emptyset$;*

- $\mathcal{R} \subseteq \mathcal{S} \times \{\mathcal{I} \cup \mathcal{O}\} \times \mathcal{S}$ *is a transition relation.*

Example 2 (Repair Fragment)**.** In connection with the "Car Status" context property in Example 1, a simple variant of the "Car Repair" fragment is given in Fig. 4.3). It is provided by a treatment facility and is used for repairing cars. It is a simple request-response fragment that, upon the request *repairRequest*, replies with *repairResponse* indicating that treatments is successful. Here and later in the text we prepend '!' and '?' to the names of controllable and uncontrollable fragment actions respectively.

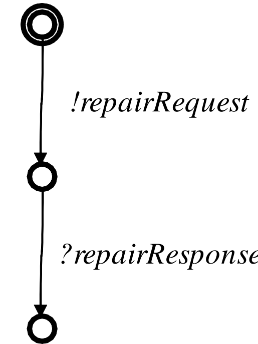


Figure 4.3: Fragment model of "Car Repair" fragment

To link a fragment to some context C, we introduce context annotations in fragment specifications. In particular:

- any SEND, RECEIVE or CONCRETE activity can be annotated with a *precondition* specifying a set of context states in which activity execution is allowed;

- any SEND, RECEIVE or CONCRETE activity can be annotated with an *effect* specifying a set of context events that are triggered with the execution of the activity;

- any ABSTRACT activity must be annotated with a *goal* specifying a set of context states in which a task related to this activity is considered to be completed.

Since propositional formulas can be used to capture any set of context states, *fragment annotation* can be formalized as follows:

Definition 8 (Fragment Annotation). *Let $f = \langle \mathcal{S}, s^0, \mathcal{I}, \mathcal{O}, \mathcal{R} \rangle$ be a fragment and let C be a context. An annotation of fragment f over context C is a tuple $\omega_f = \langle \mathcal{P}, \mathcal{E}, \mathcal{G} \rangle$, where:*

- *$\mathcal{P} : \{\mathcal{I} \cup \mathcal{O}\} \rightarrow R_C$ is the precondition labeling function;*
- *$\mathcal{E} : \{\mathcal{I} \cup \mathcal{O}\} \rightarrow E_C^*$ is the effect labeling function. Any action effect $\mathcal{E}(a)$ may contain no more than one event per context property, i. e., for any context property $p = \langle L, l^0, E, T \rangle \in C$ the following holds: $\nexists e_1, e_2 \in \mathcal{E}(a) : e_1, e_2 \in E$. Moreover, if $\mathcal{E}(a) \neq \emptyset$ then $\mathcal{G}(a) = \emptyset$ (i.e., an action can be annotated either with a goal or with an effect);*
- *$\mathcal{G} : \{\mathcal{I} \cup \mathcal{O}\} \rightarrow R_C$ is the goal labeling function, such that $\mathcal{G}(a) \neq \emptyset$ only if $\mathcal{E}(a) = \emptyset$ (i.e., an action can be annotated either with a goal or with an effect).*

We remark that these annotations are used both in APFL specifications and in formal fragments defined in Def. 7. In the latter case, the union of a fragment and its annotation form an *annotated fragment*:

Definition 9 (Annotated Fragment). *Let C be a context, let f be a fragment and let ω be its annotation over C. An annotated fragment is a tuple $f^+ = \langle f, \omega \rangle$.*

Example 3 (Repair Fragment). Joining together Examples 1 and 2, in Fig. 4.4 we show how the "Car Repair" fragment can be annotated. The *repairRequest* action is annotated with a precondition $CarStatus = nok$ meaning that its execution is allowed only in context states where the state of the "Car Status" context property is *nok*. Conceptually, it means that "Car Repair" fragment can be only applied to non-operable cars, which is

a sort of business policy. Similarly, the effect of the *repairResponse* action is associated with even *CarStatus.repaired* indicating that the expected result of the "Car Repair" fragments is that the car status changes from *nok* to *ok*. We remark that this is just a very simple example that gives the flavour of how our formal framework works. In general, fragments and their annotations may be way more complex.

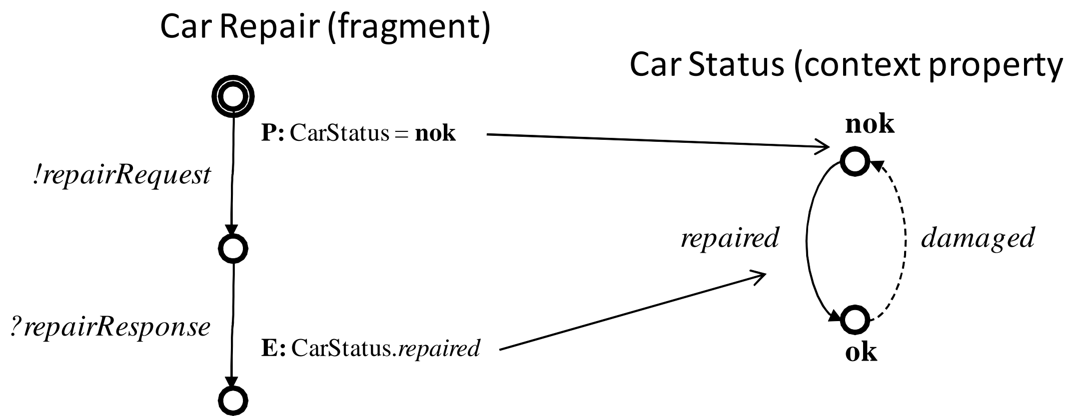


Figure 4.4: Fragment annotation of "Car Repair" fragment

Annotated APFL process as STS. In the following we present the synopsis of our APFL language with annotations and the details of how an annotated APFL process can be transformed into an annotated STS as defined in Def. 8. Since both fragments and processes are defined in the same language, the translations below are valid for both of them. Our translation supports all APFL basic and structured activities introduced in Table 3.1. In Table 4.1 the translation for basic activities is shown. Specifically, SEND and CONCRETE are represented with a single controllable transition, while RECEIVE is a single uncontrollable transition.

ABSTRACT activities are way more complex since at the moment of creating a new process they are not refined to a concrete process and the only thing we know about them is their abstract goals. We treat an abstract

Table 4.1: Translation of basic APFL activities into STSs

APFL Basic Activity	STS	Annotation
receive	s —?A→ t	$\mathcal{P}(A) = P$ $\mathcal{E}(A) = E$ $\mathcal{C}(A) = C$
send	s —!A→ t	$\mathcal{P}(A) = P$ $\mathcal{E}(A) = E$ $\mathcal{C}(A) = C$
concrete	s —!A→ t	$\mathcal{P}(A) = P$ $\mathcal{E}(A) = E$ $\mathcal{C}(A) = C$
abstract G1 = g1 and....and gi and... and gn	s —?A→ t1; t1 —!a1, !a2, !ai, !an→ t2	$\mathcal{G}(a_i) = g_i$

activity as a "black box" that performs a task as defined by its goal. In this regard, an abstract activity combines the properties of controllable and uncontrollable actions. On the one hand, the initiation of an abstract activity is controllable (within a process we can decide if to execute it and when). On the other hand, it is not possible to predict a priori the terminal context configuration. In STS, such behaviour can be modeled as a controllable action followed by a number of uncontrollable actions corresponding to all possible terminal context states. We actually reduce the number of terminal states to the number of conjunctive clauses in the

goal formula. Later in this section we will make this transformation rule clearer.

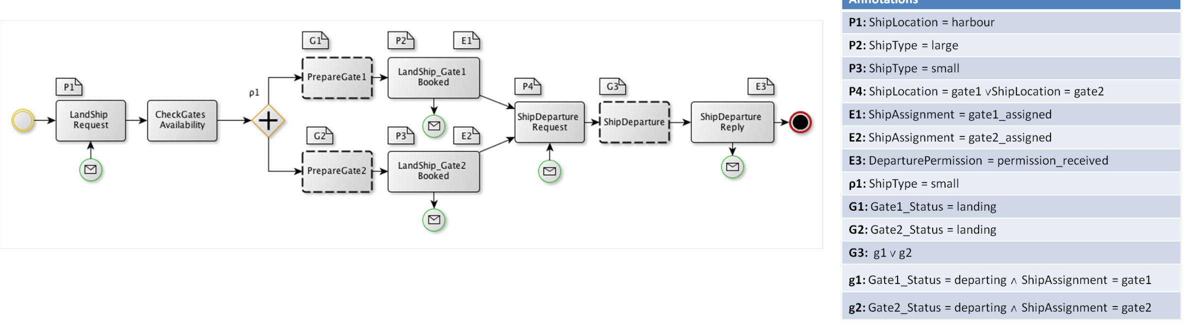
Table 4.2: Translation of structured APFL activities into STSs

APFL Structured Activity	STS	Annotation
sequence		
switch		$\mathcal{P}(a1) = \rho_1$ $\mathcal{P}(a2) = \rho_2$ $\mathcal{P}(a3) = \neg\rho_1 \vee \neg\rho_2$
pick		

Based on the above translations of basic activities, a structured APFL

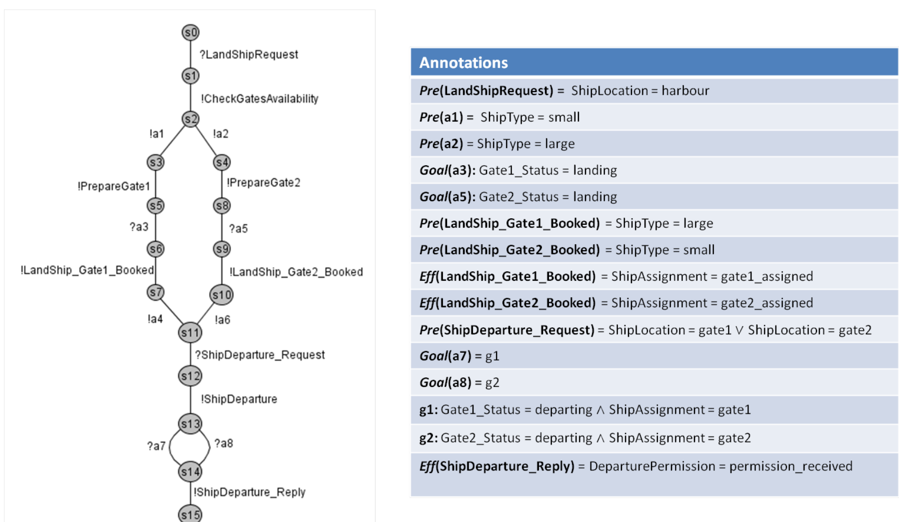
activity (SEQUENCE, SWITCH, WHILE and PICK) is translated by recursive translation of its subactivities and their further linkage into a more complex STS as shown in Table 4.2. Since and APFL process at higher level of abstraction always consists of one activity (probably, structured), such recursive translation procedure can be applied to complex processes.

Example 4. In the CLS scenario, the Landing Manager is an entity that is responsible for landing ships to the gates. As any other entity within the CLS, the landing manager has an executable process attached to it that regulates its operation. In Figure 4.5 we show the APFL process that regulates the operation of the Landing Manager. From the process, it can be seen, that ship handling consists in landing it to the gate and then providing its departure. Some of the activities are abstract (*PrepareGate1*, *PrepareGate2* and *ShipDeparture*) and will be refined at run time. The annotation details are presented in the accompanying table. The result of the translation of the main process of the Landing Manager into annotated model is shown in Figure 4.6. The transformation of fragments is done in the same way as the transformation of processes.



Annotations
P1: ShipLocation = harbour
P2: ShipType = large
P3: ShipType = small
P4: ShipLocation = gate1 ∨ShipLocation = gate2
E1: ShipAssignment = gate1_assigned
E2: ShipAssignment = gate2_assigned
E3: DeparturePermission = permission_received
ρ1: ShipType = small
G1: Gate1_Status = landing
G2: Gate2_Status = landing
G3: g1 ∨ g2
g1: Gate1_Status = departing ∧ ShipAssignment = gate1
g2: Gate2_Status = departing ∧ ShipAssignment = gate2

Figure 4.5: Annotated APFL process of Landing Manager



Annotations
***Pre*(LandShipRequest)** = ShipLocation = harbour
***Pre*(a1)** = ShipType = small
***Pre*(a2)** = ShipType = large
***Goal*(a3):** Gate1_Status = landing
***Goal*(a5):** Gate2_Status = landing
***Pre*(LandShip_Gate1_Booked)** = ShipType = large
***Pre*(LandShip_Gate2_Booked)** = ShipType = small
***Eff*(LandShip_Gate1_Booked)** = ShipAssignment = gate1_assigned
***Eff*(LandShip_Gate2_Booked)** = ShipAssignment = gate2_assigned
***Pre*(ShipDeparture_Request)** = ShipLocation = gate1 ∨ ShipLocation = gate2
***Goal*(a7)** = g1
***Goal*(a8)** = g2
g1: Gate1_Status = departing ∧ ShipAssignment = gate1
g2: Gate2_Status = departing ∧ ShipAssignment = gate2
***Eff*(ShipDeparture_Reply)** = DeparturePermission = permission_received

Figure 4.6: Landing Manager process as STS

4.2.3 STS to APFL

In order to be able to produce executable processes, we have to provide the rules for backward translation of state transition systems into APFL processes. Indeed, in our formal model, the composition is obtained in form of a state transition system and, in order to be further executed has to be converted into an APFL process. To guarantee that such conversion is possible, we have to impose a few additional restrictions on the structure of STSs encoding compositions. In particular, a solution STS cannot contain multiple controllable actions starting from the same state. Indeed, such constructions are not allowed in APFL since the process engine would not be able to figure out which of the actions has to be executed next. Similarly, the STS cannot contain an uncontrollable and controllable action from the same state, since in this case the process engine would not be able to figure out if it has to execute a controllable action or to "listen" to an uncontrollable one. Such conflicting situations are referred to in the literature as "internal" nondeterminism. At the same time, having a

number of uncontrollable actions from one state is quite natural since it reflects the nondeterministic behaviour of external partner, where all cases have to be taken into account. Such situation is easily processed by the process engine (e.g., PICK activity of APFL models the situation where multiple uncontrollable actions can unpredictably fire). These second type of situations is also known as ”external” nondeterminism. Summing it up, “external” nondeterminism is acceptable and “internal” nondeterminism is not acceptable in an STS that encodes an executable process. We also require that runnable process is deterministic. Formally, we introduce *runnable STS* as follows:

Definition 10 (Runnable Process). *A process* $f = \langle \mathcal{S}, s^0, \mathcal{I}, \mathcal{O}, \mathcal{R} \rangle$ *is runnable if for each* $s \in \mathcal{S}$*, if* $(s, a, s') \in \mathcal{R}$ *and* $a \in \mathcal{I}$ *then no other transition is available from* s*.*

The only situation where multiple controllable actions from the same state could be unambiguously resolved is when they featured preconditions corresponding to mutually disjoint sets of context states. These would model a SWITCH structure in the final process. However, since in our model context configuration at each point is completely predictable (context evolves only as a result of action executions), at each such point of choice we can unambiguously figure out which choice will be made by the process engine at run time. Consequently, including all controllable actions from the same state in this case would be somewhat redundant.

The translation of a runnable process (STS) into an APFL process is quite straightforward. Since we prevent loops in the final STS (it will be better explained in the section to come), we simply revisit all states of the runnable STS and convert the respective transitions into APFL activities using the information in specifications of component fragments and by inverting the transformation rules for APFL-to-STS conversion. For example, an uncontrollable transitions become *receive* (or PICK if muti-

ple uncontrollable transitions originate from the same state). Similarly, controllable transitions become *send* or *concrete* or ABSTRACT.

4.2.4 Context-Aware System and Annotation Semantics

A context and a set of fragments annotated over it form a *context-aware system*:

Definition 11 (Context-Aware System). *Let C be a context and let $F^+ = \{\langle f_1, \omega_1\rangle, \langle f_2, \omega_2\rangle, \ldots, \langle f_n, \omega_n\rangle\}$ be a set of fragments annotated over C. Context-aware system is a tuple $\Psi = \langle F^+, C\rangle$.*

By means of fragment annotations, the relation between the context evolution and fragment execution is created. This relation is twofold. First, the ability of the activity to be executed is constrained to certain context configurations. Second, the activity execution may trigger the context evolution. To formalize these two relations, we introduce the notions of *action executability* and *action impact*.

An action can be executed from a context configuration if 1) the action precondition holds in this configuration and 2) all context events belonging to the action effect are applicable on this configuration. Formally:

Definition 12 (Action Executability). *Let $\Psi = \langle F^+, C\rangle$ be a context-aware system with a set of context states L_C and a set of context events E_C. Action a belonging to some fragment $f^+ \in F^+$ annotated with precondition $\mathcal{P}(a) = \rho$ and effect $\mathcal{E}(a) = \{e_1, e_2, \ldots, e_n\}$ is executable from context state $l \in L_C$ if and only if*

1. *$l \models \rho$;*

2. *for all $i \in [1, n]$ event $e_i \in \mathcal{E}(a)$ is applicable on state l, i.e., $App_C(e_i, l)$;*

The set of all context states in which a is executable is denoted as $Exec_\Psi(a)$.

The impact indicates how the current context state changes after action execution. The impact of an action is determined by the action effect or by the action goal (only one of them can belong to action annotation). Actions that are not annotated with effect or goal are *impactless* and do not change the context configuration.

For an action annotated with effect, all events belonging to the effect are considered to be triggered as a result of action execution:

Definition 13 (Action Impact (Effect)). *Let $\Psi = \langle F^+, C \rangle$ be a context-aware system with a set of context states L_C and a set of context events E_C. The impact of action a belonging to some fragment $f^+ \in F^+$ and annotated with non-empty effect ($\mathcal{E}(a) \neq \emptyset$) when executed from context configuration $l \in L_C$ (denoted $Imp_\Psi(a, l)$) is a context configuration $l' \in L_C$ such that for every context property $p_i = \langle L_i, l_i^0, E_i, T_i \rangle \in C$, if $\exists e \in \mathcal{E}(a) : (l \downarrow_{p_i}, e, l'_i) \in T_i$ then $l' \downarrow_{p_i} = l'_i$, otherwise $l' \downarrow_{p_i} = l \downarrow_{p_i}$.*

In other words, if an action has a non-empty effect then all events belonging to the effect fire as a result of its execution (all the constituent context properties evolve correspondingly). Due to the fact that action annotation can have no more than one event per context property (see Def. 8) and context properties are deterministic STSs, action impact is always deterministic. Although in our model we never consider action impact of a in states where a is not executable, Def. 13 does not prohibit such calculations explicitly.

In order to define a goal-based impact we introduce the notion of minimal satisfaction:

Definition 14 (Minimal Satisfaction). *Let $C = \{p_1, p_2, \dots, p_n\}$ be a context with a set of context states L_C and let $\rho \in R_C$ be a context formula that is a conjunctive clause of context property states. State $l' = (l'_1, \dots, l'_j, \dots, l'_n) \in L_C$ is a minimal satisfaction of conjunctive clause ρ*

for state l (written $Min(l,\rho)$) if $l' \models \rho$ and for all $j \in [1,n]$ the following holds: $(l'_j \neq l_j) \rightarrow ((l'_1, \ldots, l'_{j-1}, l_j, l'_{j+1}, \ldots, l'_n) \not\models \rho)$.

In fact, a minimal satisfaction for state l and context formula ρ is a context state l' that differs from l only in those components, that must be changed in order to satisfy ρ. All the other components remain the same as in l. It is trivial to prove that for any context state l and any conjunctive clause ρ there exists exactly one context state $Min(l,\rho)$.

For an action annotated with a goal (which is, as it can be seen from Table 4.2, always a conjunctive clause), the impact is derived from the assumption that an abstract action never produces *side-effects*. In other words, it satisfies its goal with minimal changes to the context, which are defined with minimal satisfaction. Taking into account that in fragments goals are always conjunctive clauses, we define the goal-based impact as follows:

Definition 15 (Action Impact (Goal)). *Let $\Psi = \langle F^+, C \rangle$ be a context-aware system with a set of context states L_C and a set of context events E_C. The impact of action a belonging to some fragment $f^+ \in F^+$ and annotated with non-empty goal ($\mathcal{G}(a) \neq \emptyset$) when executed from context configuration $l \in L_C$ (denoted $Imp_\Psi(a,l)$) is a context configuration $l' \in L_C$ such that $l' = Min(l, \mathcal{G}(a))$.*

Finally, the actions that have neither effect nor goal are called *impactless* and do not change the state of the context when executed:

Definition 16 (Action Impact (Empty)). *Let $\Psi = \langle F^+, C \rangle$ be a context-aware system with a set of context states L_C. The impact of action a belonging to some fragment $f^+ \in F^+$ and annotated with neither effect nor goal ($\mathcal{G}(a) = \emptyset \wedge \mathcal{E}(a) = \emptyset$) when executed from context configuration $l \in L_C$ (denoted $Imp_\Psi(a,l)$) is a context configuration $l' = l$.*

It can be easily observed that impact is a deterministic function, i.e., for each pair a, l there is always a single state $l' = Imp_{\Psi}(a, l)$.

4.2.5 Composition Requirements

In our fragment composition engine, we specify composition requirements as a set of goal context states that have to be reached as a result of composition execution. As such, for a context-aware system $\Psi = \langle F^{+}, C\rangle$, composition requirements are expressed as a context formula $\rho \in R_C$

4.3 Problem of Context-Aware Fragment Composition

Within fragment orchestration, the component fragments are executed in parallel and evolve independently. We assume that fragments within a single context-aware system have uncorrelated actions, i.e., such fragments have mutually disjoint sets of actions. In order to encode all possible parallel executions of fragments for some context-aware system we introduce the notion of *execution domain*, which is a parallel product of fragments:

Definition 17 (Execution Domain). *Let $f_1 = \langle \mathcal{S}_1, s_1^0, \mathcal{I}_1, \mathcal{O}_1, \mathcal{R}_1\rangle$ and $f_2 = \langle \mathcal{S}_2, s_2^0, \mathcal{I}_2, \mathcal{O}_2, \mathcal{R}_2\rangle$ be two observable state transition systems such that $(\mathcal{I}_1 \cup \mathcal{O}_1) \cap (\mathcal{I}_2 \cup \mathcal{O}_2) = \emptyset$. An execution domain Σ_F for fragments $F = \{f_1, f_2\}$ is an asynchronous product of two fragments:*

$$\Sigma_F = \langle \mathcal{S}_1 \times \mathcal{S}_2, (s_1^0, s_2^0), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{R}_F\rangle$$

where:

$$((s_1, s_2), a, (s_1', s_2)) \in \mathcal{R}_F, \ \textit{if}\,(s_1, a, s_1') \in \mathcal{R}_1$$
$$((s_1, s_2), a, (s_1, s_2')) \in \mathcal{R}_F, \ \textit{if}\,(s_2, a, s_2') \in \mathcal{R}_2$$

When we consider a context-aware system $\Psi = \langle F^+, C \rangle$ such that $F^+ = \{\langle f_1, \omega_1 \rangle, \langle f_2, \omega_2 \rangle, \ldots, \langle f_n, \omega_n \rangle\}$ the execution domain for Ψ is an asynchronous product of all its fragments $f_1, \ldots, f_n$.

In the future we will intensively use some STS-related terms that are given below regarding some execution domain $\Sigma_F = \langle \mathcal{S}_F, s^0_F, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_F \rangle$. *STS run* is a sequence $\pi = (s_1, a_1, s_2, a_2, \ldots, a_{n-1}, s_n)$ such that $s_1 = s^0_F$ and $\forall i \in [1, n] : s_i \in \mathcal{S}_F$ and $\forall i \in [1, n-1] : a_i \in (\mathcal{I}_F \cup \mathcal{O}_F)$. Moreover, $\forall i \in [1, n-1] : (s_i, a_i, s_{i+1}) \in \mathcal{R}_F$. *Final states* of an STS (denoted $Finals(\Sigma_F)$) are the states that have no outgoing transitions. A run that terminates in a final state is called a *complete run*.

Intuitively, every run of a process that correctly (with respect to the action order in fragments) orchestrates a set of fragments F has to be a run of the respective execution domain Σ_F. And so the idea of fragment composition consists in finding a set of executions of execution domain that satisfy certain composition requirements and constraints. In the following we will talk about these requirements and constraints in detail.

In the future, in order to show how the current context state changes in time, we will use the notion of *context evolution*. Formally, context evolution of context C is any sequence of context configurations belonging to L_C.

First of all, certain constraints on the runs are imposed by the context model and semantics of fragment annotations. In particular, we require that we consider only those runs that, in the presence of context evolving according to action impact (Definitions 13,15,16), never violate action preconditions. We call this runs *context-aware* and define them as follows.

Definition 18 (Context-Aware Run)**.** *Let $\Psi = \langle F^+, C \rangle$ be a context-aware system with initial context state l^0_C and let Σ_F be its execution domain. A run of Σ_F*

$$\pi = (s_1, a_1, s_2, a_2, \ldots, a_{n-1}, s_n)$$

is context-aware in Ψ *if there exists a context evolution of* C

$$\pi_C = (l_1, l_2, \ldots, l_n)$$

such that

- $l_1 = l_C^0$;
- $Imp_\Psi(a_i, l_i) = l_{i+1}$ *for all* $i \in [1, n-1]$;
- $l_i \in Exec_\Psi(a_i)$ *for all* $i \in [1, n-1]$.

In the definition we exploit the fact that, since execution domain Σ_F shares actions with component fragments F, the respective fragment annotations of the context-aware system $\Psi = \langle F^+, C \rangle$ remain valid in Σ_F. The context evolution π_C is called *associated context evolution* for the execution domain run π. It is obvious that since impact is deterministic, for each context-aware run there exists only one associated context evolution.

Since our composition goals are expressed as context formulas, the run that achieves the goal is the one whose associated context evolution terminates in a context state satisfying the formula. Formally:

Definition 19 (Satisfying Run)**.** *Let* $\Psi = \langle F^+, C \rangle$ *be a context-aware system, let* Σ_F *be its execution domain and let* $\rho \in R_C$ *be a context formula for* C*. A run of* Σ_F

$$\pi = (s_1, a_1, s_2, a_2, \ldots, a_{n-1}, s_n)$$

is satisfying for ρ *if it is context-aware for* Ψ *and its associated context evolution*

$$\pi_C = (l_1, l_2, \ldots, l_n)$$

is such that $l_n \models \rho$.

In order to be able to deal with the execution domain in the presence of context, we introduce the notion of *context-aware execution domain.* The

context-aware execution domain is an STS that is a sort of synchronous product of the execution domain and the context based on the notions of action impact and action executability. The idea is that the context-aware execution domain reflects how the context evolves with the execution of fragment actions and explicitly prohibits violating executions of fragment actions:

Definition 20 (Context-Aware Execution Domain)**.** *Let $\Psi = \langle F^+, C\rangle$ be a context-aware system, let $\Sigma_F = \langle \mathcal{S}_F, s^0_F, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_F\rangle$ be its execution domain and let C be its context with a set of context states L_C. A context-aware execution domain is an STS $\Sigma_{CF} = \langle \mathcal{S}_{CF}, s^0_{CF}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_{CF}\rangle$ such that:*

$$\Sigma_{CF} = \langle \mathcal{S}_F \times L_C, \{s^0_S, l^0_C\}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_{CF}\rangle$$

where:

$$((s,l),a,(s',l')) \in \mathcal{R}_{CF}, \;\; \textit{if } (s,a,s') \in \mathcal{R}_F, \;\; \textit{and } l \in Exec_\Psi(a) \;\; \textit{and } l' = Imp_\Psi(a,l);$$

Taking into account that Σ_F is a deterministic STS and impact is a deterministic function we conclude that Σ_{CF} is a deterministic STS.

Before we proceed with the other details, we have to remark that the context-aware execution domain as it is defined above, eliminates from consideration all exogenous events, i.e., those events that are not associated with some fragment activity. From the further definitions it will become clear, that the resulting fragment composition neglects the possibility of exogenous events. However, in Section 3.4 we explain how exogenous events, though ignored by the composition, can be properly handled at the phase of composition execution.

From the definition above and the definition of context-aware run (Def. 18)

it can be easily observed that for every context-aware run

$$\pi_F = (s_1, a_1, s_2, a_2, \ldots, a_{n-1}, s_n)$$

of execution domain Σ_F there exists a run

$$\pi_{CF} = ((s_1, l_1), a_1, (s_2, l_2), a_2, \ldots, a_{n-1}, (s_n, l_n))$$

of context-aware execution domain Σ_{CF} such that

$$\pi_C = (l_1, l_2, \ldots, l_n)$$

is the associated context evolution for π_F. The same is true in the opposite direction: for every run of π_{CF} of Σ_{CF} there exists an equivalent run π_F of Σ_F. So we can conclude that Σ_{CF} encodes all the context-aware runs of Σ_F.

The notion of satisfying run can also be easily propagated to context-aware execution domain. A run

$$\pi_{CF} = ((s_1, l_1), a_1, (s_2, l_2), a_2, \ldots, a_{n-1}, (s_n, l_n))$$

of context-aware execution domain Σ_{CF} is satisfying for context formula ρ if $l_n \models \rho$.

Since we are interested only in context-aware and satisfying runs, we are essentially interested in the runs of the context-aware execution domain that satisfy the goal formula. In order to encode a set of runs of the execution domain, which is the prototype of the solution to the composition problem, we introduce the notion of *solution executor*:

Definition 21 (Solution Executor)**.** *Let $\Psi = \langle F^+, C\rangle$ be a context-aware system and let $\Sigma_{CF} = \langle \mathcal{S}_{CF}, s^0_{CF}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_{CF}\rangle$ be its context-aware execution domain. A solution executor for Σ_{CF} and context formula ρ is an STS $\Sigma_E = \langle \mathcal{S}_E, s^0_E, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_E\rangle$ such that:*

- *$\mathcal{S}_E \subseteq \mathcal{S}_{CF}$, $s^0_E = s^0_{CF}$, $\mathcal{R}_E \subseteq \mathcal{R}_{CF}$, i.e., Σ_E is a subgraph of Σ_{CF};*

- $\forall s \in Finals(\Sigma_E) : s \models \rho$, *i.e., all complete runs of* Σ_E *are satisfying for* ρ.

The fact that we search for a solution in form of subgraph of the context-aware execution domain imposes some restrictions on the solutions that can be found by the approach. However, as we will discuss later on in this section, these restrictions are reasonable and do not affect the practical applicability of the approach.

There are some additional restrictions that we have to deliberately impose on the solution executor in order to guarantee that it executes the fragments within a context-aware system consistently. In the following we explain what we mean by "consistency" and give the appropriate formal definition. In the discussion we will use the fact that every state s of a solution executor can be always associated with a pair (s_F, l) where s_F is a state of respective execution domain and l is a state of context (this fact is a direct consequence of Definitions 20 and 21).

First, since the solution executor is supposed to encode a solution to a composition problem and, as such it is supposed to be further translated into APFL, it has to be a runnable process as defined in Definition 10.

Second, while executing a number of fragments in parallel, the executor has to take into account the asynchronous behaviour of the fragments and treat controllable and uncontrollable actions differently. In particular, we have to bear in mind that uncontrollable actions in fragments are unpredictable (we cannot predict which of uncontrollable actions available from the current state of the fragment will fire next) and uncontrollable (we cannot prevent the firing of an uncontrollable transition if it is available from the current state of the fragment). Consequently, in order to avoid inconsistent behaviour we require that every time the solution executor brings its context-aware execution domain to a state from which uncontrollable actions are available the next actions of the executor must be all uncontrol-

lable actions that are available from the current state of the context-aware execution domain and are executable (Def. 12) from the current context state. We remark that any other behaviour of the executor may result in unpredictable behaviour and dead-locks of the whole system. Indeed, if the executor does not "listen" to some uncontrollable action available, the corresponding message will be lost if this action fires. Moreover, if the executor tries to execute some controllable action while an uncontrollable action is available from the current state, the latter can fire in the meanwhile, which will again result in message loss.

Third, we want the executor to reach its objectives in efficient way, that is without traversing more than once the same state of the execution domain in the same context. This requirement actually implies that all executions of the executor have to be finite and cannot contain loops.

We join all the restrictions above in the definition of *consistent executor*:

Definition 22 (Consistent Executor). *Let $\Psi = \langle F^+, C\rangle$ be a context-aware system, let $\Sigma_{CF} = \langle \mathcal{S}_{CF}, s^0_{CF}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_{CF}\rangle$ be its context-aware execution domain and let $\Sigma_E = \langle \mathcal{S}_E, s^0_E, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_E\rangle$ be a solution executor for Σ_{CF} and some context formula ρ. Solution executor Σ_E is consistent if:*

1. *for all $s \in \mathcal{S}_E$, if exists transition $(s, a', s') \in \mathcal{R}_E : a \in \mathcal{I}_F$, then it is the only transition from this state ($\nexists(s, a'', s'') \in \mathcal{R}_E$);*

2. *if $s \in \mathcal{S}_E : s \not\models \rho$ then $\forall(s, a, s') \in \mathcal{R}_{CF} : (a \in \mathcal{O}_F) \rightarrow ((s, a, s') \in \mathcal{R}_E))$, i.e., if the executor traverses some state of the context-aware execution domain where uncontrollable actions are available, it has to include from this state all respective uncontrollable transitions and, as such, to account for all possible outputs of the execution domain;*

3. *for every complete run π_E of Σ_E any state $s \in \mathcal{S}_E$ is traversed no more than once, i.e., Σ_E does not contain infinite runs.*

A few important observations can be done in the definition above:

- a consistent executor is a runnable process (condition 1);
- a consistent executor correctly tackles fragments asynchronicity: condition 1 prevents controllable and uncontrollable action originating from the same state and condition 2 guarantees that all uncontrollable actions are always accounted;
- due to the fact that all runs are finite (condition 3) a consistent executor is a DAG structure that does not contain cycles.

It is worth to notice that in condition 2 we require that all the uncontrollable actions are accounted only for non-goal states. We assume that once a goal state is reached the composition terminates despite the existence of uncontrollable transitions available from this state. The different approach is considered in Chapter 7, where the composition is continuous and takes into account the fact that once a goal state is reached, the system can still be forced to leave it unintentionally through uncontrollable transitions. In this case the further coordination has to be provided in order to bring the system back to one of its goal states.

The fact that the solution is searched for in form of subgraph of the context-aware execution domain actually imposes the restriction that from the same state of the context-aware execution domain, the consistent executor must always execute the same controllable action (otherwise the executor will not be runnable). We remark that in extreme cases the same configuration of the context-aware execution domain can be reached though different paths in the domain and, having different execution history one may take different decision on a controllable action to execute. However, in our approach we assume that the current run-time situation is only determined by the state of the execution domain and the state of the context (it is basically the context who stores all important information about the

previous execution) and does not explicitly take into account the previous execution history (i.e., does not have memory other than the current context). That is why we find this restriction reasonable: there is no sense to make different decisions in the same state of the context-aware execution domain.

Eventually, the problem of context-aware fragment composition can be formulated as a problem of finding a consistent executor satisfying composition requirements in form of goal formula:

Definition 23 (Problem of Context-Aware Fragment Composition)**.** *Let $\Psi = \langle F^+, C\rangle$ be a context-aware system and let $\rho \in R_C$ be a context formula over C expressing composition requirements. The problem of context-aware fragment composition for Ψ and ρ consists in finding a solution executor Σ_E for Ψ and ρ that is consistent.*

4.4 Composition Problem as Planning Problem

In this section we explain how a fragment composition problem as defined in Def. 23 can be transformed into a planning problem and can consequently be resolved by a planning algorithm for asynchronous and nondeterministic domains. Specifically, we present a multi-step procedure for building a planning domain from the elements of the composition problem and formulate a planning problem. We prove the correctness and completeness of our approach by showing that 1) any solution to a planning problem is a consistent executor for a given composition problem and 2)if a solution to a planning problem is not found, a solution to the composition problem does not exist.

The overview of our fragment composition approach is given in Fig. 4.7. The composition engine accepts as input a context C represented by context properties $p_1, p_2, \ldots, p_m$, a set of fragments $F^+ = \{f_1^+, f_2^+, \ldots, f_n^+\}$

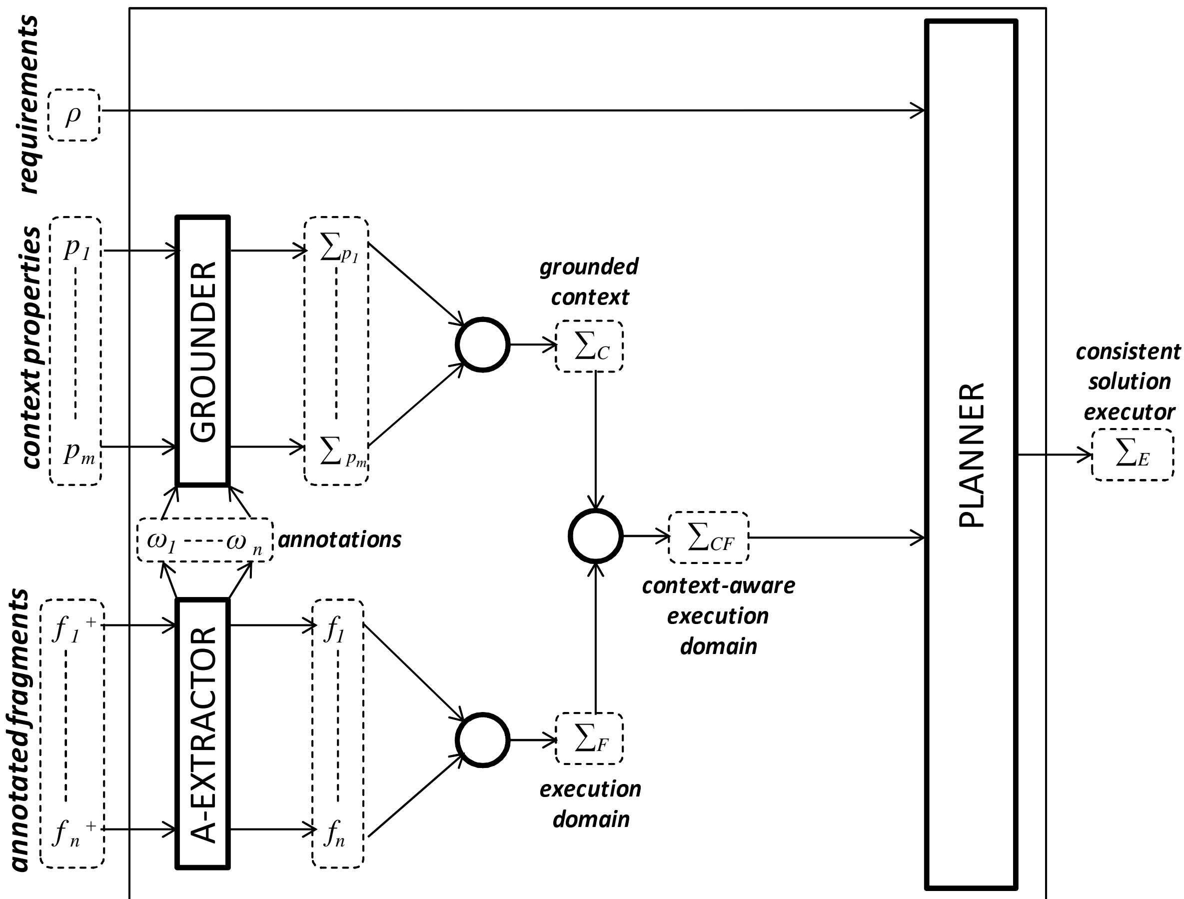


Figure 4.7: Fragment composition approach

annotated over C (together C and F^+ form a context-aware system $\Psi = \langle F^+, C \rangle$) and composition requirements ρ expressed as a context formula over C. The output is an executor Σ_E that is a solution executor for Ψ and ρ and is consistent. As we showed in Section 4.2, fragment models can be directly derived from APFL process specifications and a runnable process (including Σ_E) can be translated into an APFL process.

The very general idea of the approach consists in building a planning domain Σ_{CF}, that together with a goal ρ form a planing problem.

The *execution domain* Σ_F is built as asynchronous product of fragments F^+ (see Def. 17). Using fragment annotations extracted by A-

EXTRACTOR, context properties are grounded on fragment actions by GROUNDER so that the grounded context properties $\Sigma_{p_1}, \Sigma_{p_2}, \ldots, \Sigma_{p_m}$ are produced. In brief, the procedure of grounding is where event-labeled transitions in context properties are replaced with guarded action-labeled transitions basing on fragment action effects and preconditions. Additionally, transitions corresponding to goal-based actions and impactless actions are added. When derived in this way, a grounded context property reflects how the execution of fragment actions affects its state. It also accounts for action executability. The *grounded context* Σ_C is obtained as a synchronous product of the grounded context properties. As it will be proved later on, its runs encode all the evolutions of the context that are enabled by the current system of annotated fragments F^+.

In turn, the synchronous product of the execution domain Σ_F and grounded context Σ_C is a *context-aware execution domain* Σ_{CF} whose runs are all possible context-aware runs of the execution domain Σ_F. Moreover, Σ_{CF} reflects contextual impact of any its run. We show that it coincide with the notion of context-aware execution domain presented in Def. 20. The goal formula ρ can be directly applied to the context-aware execution domain. We show that using a specific planning algorithm, the consistent solution executor for a composition problem of Ψ and ρ can be directly derived from a composition problem of Σ_{CF} and ρ. Moreover, if such plan is not found, then the composition problem does not have a solution.

4.4.1 Grounded Context

The procedure of context grounding consists in replacing event-labeled transitions in context properties with action-labeled transition reflecting the impact of these actions on the property state. Additionally, we use transition guards to reflect the executability of actions. As a result the *grounded context property* features the same set of states as the orginial

context property, but has different transition relation:

Definition 24 (Grounded Context Property)**.** *Let* $\Psi = \langle F^+, C\rangle$ *be a context-aware system and let* R_C *be a space of context formulas of context* C*. A grounded context property for context property* p *is a tuple* $\Sigma_p = \langle L, l^0, \mathcal{A}_F, T\rangle$*, where:*

- L *is a set of states and* $l^0 \in L$ *is the initial state;*
- $\mathcal{A}_F$ *is a set of all fragment actions of fragments* F^+*;*
- $T \subseteq L \times R_C \times \mathcal{A}_F \times L$ *is a guarded transition relation.*

The procedure of grounding consists in defining a grounded context property Σ_p on top of a context property p. While the sets of states in p and Σ_p are the same, in Σ_p the event-based transitions of p are replaced with action-based transitions as indicated by annotations. For each transition of p labelled with event e and for each action a whose effect contains e, we define a transition in Σ_p with the same initial and final state and labelled with a. For each goal-labeled action a_{abs}, if an action goal (which is a conjunctive clause) requires that this property has to be in a particular state l (i.e., a proposition corresponding to l appears in the conjunctive clause expressing the goal of a_{abs}), for every state in Σ_p we add a transition that starts in this state, terminates in l and is labelled with a_{abs}. Finally, for each action a_{less} that has no impact on the property we define a transition that start and finishes in this state and is labelled with a_{less}. As such, we reflect the impact of all actions with respect to context property p. In order to take into account action preconditions, for each transition we introduce the guard, which is a precondition formula of its labelling action. A transition guard must be interpreted as a condition on the state of the whole context for which the transition is enabled. Formally:

Definition 25 (Grounding). *Let $\Psi = \langle F^+, C\rangle$ be a context-aware system. A grounding of a context property $p = \langle L, l^0, E, T\rangle \in C$ is a grounded context property $\Sigma_p = \langle L, l^0, \mathcal{A}_F, T^g\rangle$ such that for every action $a \in \mathcal{A}_F$:*

1. *if $\exists e \in \mathcal{E}(a) : e \in E$ then for every transition $(l, e, l') \in T$ there exists transition $(l, \mathcal{P}(a), a, l') \in T^g$;*
2. *if state $l_g \in L$ appears in conjunctive clause $\mathcal{G}(a)$ then for every state $l \in T$ there exists transition $(l, \mathcal{P}(a), a, l_g) \in T^g$;*
3. *if $\mathcal{E}(a) = \emptyset \wedge \mathcal{G}(a) = \emptyset$ or $(\mathcal{E}(a) \neq \emptyset) \wedge (\mathcal{E}(a) \cap E = \emptyset)$ or $(\mathcal{G}(a) \neq \emptyset) \wedge (\nexists l_g \in L : l_g \in \mathcal{G}(a))$ then for every state $l \in L$ there exists a transition $(l, \mathcal{P}(a), a, l') \in T^g$;*
4. *no other states and transitions belong to Σ_p.*

Since action effect contains no more than one event per context property, and since a goal conjunctive clause cannot contain more than one state per context property, the grounded context property is a deterministic STS (only one transition with the same label is possible from each state).

In order to reflect the impact and executabilty of fragment actions with respect to the whole context we introduce the notion of *grounded context*, which is a synchronous product of all constituent grounded context properties. We remark that the guards in the synchronous product can be removed. Indeed, for any guarded transition we can unambiguously figure out if the initial state of a transition satisfies the guard (a guard is a context formula and a state is a context state, so the Def. 5 for context satisfaction can be applied). Consequently, if the initial state satisfies the guard it is always “unlocked” and we can replace it with the unguarded transition with the same properties, and if the initial state does not satisfy the guard it is always “locked” and can be removed from the transition relation. Formally:

Definition 26 (Grounded Context)**.** *Let $\Psi = \langle F^+, C\rangle$ be a context-aware system with context $C = \{p_1, p_2, \dots, p_n\}$ and let $\Sigma_{p_1}, \Sigma_{p_2}, \dots, \Sigma_{p_n}$ be the respective grounded context properties such that $\Sigma_{p_i} = \langle L_i, l_i^0, \mathcal{A}_F, T_i\rangle$ for all $i \in [1, n]$. Grounded context for Ψ is an STS $\Sigma_C = \langle L_C, l_C^0, \mathcal{A}_F, T_C\rangle$ which is defined as follows:*

$$\Sigma_C = \langle L_1 \times \dots \times L_n, \{l_1^0, \dots l_n^0\}, \mathcal{A}_F, T_C\rangle$$

where:

$$((l_1, \dots, l_n), a, (l'_1, \dots l'_n)) \in T_C, \;\; \text{if } (l_i, \mathcal{P}(a), a, l'_i) \in T_i \text{ for all } i \in [1, n] \text{ and } (l_1, \dots, l_n) \models \mathcal{P}(a)$$

It can be easily shown that since grounded context properties are deterministic STSs, the grounded context is also a deterministic STS. A set of states in grounded context coincides with the set of states in the corresponding context (Def. 2).

Conceptually, grounded context Σ_C for context-aware system $\Psi = \langle F^+, C\rangle$ reflects all evolutions of context C that can be caused by the execution of fragments F^+ according to their annotations. In the following lemma we show that every transition $(l, a, l') \in T_C$ in the grounded context is such that $l' = Imp_\Psi(a, l)$ and $l \in Exec_\Psi(a)$. Moreover, for each pair of states $l, l' \in L_C$ and action $a \in \mathcal{A}_F$, such that $l' = Imp_\Psi(a, l)$ and $l \in Exec_\Psi(a)$, the corresponding transition exists in the grounded context ($(l, a, l') \in T_C$).

Lemma 1 (Properties of Grounded Context)**.** *Let $\Psi = \langle F^+, C\rangle$ be a context-aware system and let $\Sigma_C = \langle L_C, l_C^0, \mathcal{A}_F, T_C\rangle$ be its grounded context as defined in Definition 26. Then $(l, a, l') \in T_C$ if and only if $l \in Exec_\Psi(a)$ and $Imp_\Psi(a, l) = l'$.*

Proof. In fact, the proof directly follows from the way we constructed the grounded context and the definitions of action executability (Def. 12) and

action impact (Defs. 13, 15, 16). In the proof we presume that $C = \{p_1, \ldots, p_n\}$ and that $p_i = \langle L_i, l^0, E_i, T_i \rangle, i \in [1, n]$ and their respective grounded properties are $\Sigma_{p_i} = \langle L_i, l_i^0, \mathcal{A}_F, T_i^g \rangle, i \in [1, n]$. In order to prove the lemma we have to prove the two statements.

1. $\forall (l, a, l') \in T_C : (Imp_\Psi(a, l) = l') \wedge (l \in Exec_\Psi(a))$. Let $(l, a, l') \in T_C$ and let $l = (l_1, \ldots, l_n)$ and $l' = (l'_1, \ldots, l'_n)$.

From Def. 26 of the grounded context it follows that $(l_i, \mathcal{P}(a), a, l'_i) \in T_i^g, i \in [1, n]$. From Def. 24 we can conclude that:

(a) if $\mathcal{E}(a) \neq \emptyset$ then

$$(\exists e \in \mathcal{E}(a) : e \in E_i) \rightarrow ((l_i, e, l'_i) \in T_i), i \in [1, n]$$

and

$$(\nexists e \in \mathcal{E}(a) : e \in E_i) \rightarrow (l'_i = l_i), i \in [1, n].$$

From Def. 13, it follows that $Imp_\Psi(a, l) = l'$;

(b) if $\mathcal{G}(a) \neq \emptyset$ then from Def. 25, 26 it follows that $l' = Min(l, \mathcal{G}(a))$. From Def. 15, it follows that $Imp_\Psi(a, l) = l'$;

(c) if $\mathcal{G}(a) = \emptyset \wedge \mathcal{E}(a) = \emptyset$ then $l' = l$. From Def. 16, it follows that $Imp_\Psi(a, l) = l'$.

This means that in general $Imp_\Psi(a, l) = l'$. From statement (a) and from Def. 26 it also follows that if $\mathcal{E}(a) \neq \emptyset$ then all events of $\mathcal{E}(a)$ are applicable on l (Def. 6). Taking into account that according to Def. 26 $l \models \mathcal{P}(a)$ we conclude that $l \in Exec_\Psi(a)$ (Def. 12).

2.$\forall a \in \mathcal{A}_F, \forall l, l' \in L_C : ((Imp_\Psi(a, l) = l') \wedge (l \in Exec_\Psi(a))) \rightarrow ((l, a, l') \in T_C)$. Let $l = (l_1, \ldots, l_n)$ and $l' = (l'_1, \ldots, l'_n)$. From the fact that $Imp_\Psi(a, l) = l'$ it follows that

(a) if $\mathcal{E}(a) \neq \emptyset$ then

$$(\exists e \in \mathcal{E}(a) : e \in E_i) \rightarrow ((l_i, e, l'_i) \in T_i), i \in [1, n]$$

and

$$(\nexists e \in \mathcal{E}(a) : e \in E_i) \rightarrow (l'_i = l_i), i \in [1, n].$$

At the same time, from Def. 25 it follows that $(l_i, \mathcal{P}(a), a, l'_i) \in T^g_i$, $i \in [1, n]$. Consequently, taking into account that $l \in Exec_\Psi(a)$ and considering Def. 26 we can conclude that $(l, a, l') \in T_C$;

(b) if $\mathcal{G}(a) \neq \emptyset$ then $l' = Min(\mathcal{G}(a), l)$. At the same time, from Definition 25 it follows that $(l_i, \mathcal{P}(a), a, l^g_i) \in T^g_i, i \in [1, n]$ such that $(l^g_1, l^g_2, \ldots, l^g_n) = Min(l, \mathcal{G}(a))$. Taking into account that $l \in Exec_\Psi(a)$ and considering Def. 26 we can conclude that $(l, a, l') \in T_C$;

(c) if $\mathcal{G}(a) = \emptyset \wedge \mathcal{E}(a) = \emptyset$ then $l' = l$. At the same time, from Def. 25 it follows that $(l_i, \mathcal{P}(a), a, l'_i) \in T^g_i, i \in [1, n]$. Taking into account that $l \in Exec_\Psi(a)$ and considering Def. 26 we can conclude that $(l, a, l') \in T_C$.

□

4.4.2 Context-Aware Execution Domain From Grounding

In Def. 20 we already introduced context-aware execution domain. The alternative way of obtaining a context-aware execution domain is through synchronous product of grounded context and execution domain. This method is more practical from the implementation perspective and gives additional understanding of the nature of the context-aware execution domain.

Definition 27 (Context-Aware Execution Domain (From Grounding)). *Let $\Psi = \langle F^+, C \rangle$ be a context-aware system, let $\Sigma_F = \langle \mathcal{S}_F, s^0_F, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_F \rangle$ be its execution domain and let $\Sigma_C = \langle L_C, l^0_C, \mathcal{A}_F, T_C \rangle$ be its grounded context (from the definitions of execution domain and grounded context it follows that $\mathcal{A}_F = \mathcal{I}_F \cup \mathcal{O}_F$). The context-aware execution domain is a*

fragment-like STS $\Sigma_{CF} = \langle \mathcal{S}_{CF}, s^0_{CF}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_{CF} \rangle$ *that is defined as follows:*

$$\Sigma_{CF} = \langle \mathcal{S}_F \times L_C, \{s^0_S, l^0_C\}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_{CF} \rangle$$

where:

$$((s,l),a,(s',l')) \in \mathcal{R}_{CF}, \ \textit{if } (s,a,s') \in \mathcal{R}_F, \ \textit{and } (l,a,l') \in T_C$$

From Lemma 1 it becomes obvious that Defs. 20 and 27 actually define the same STS. As a consequence, the consistent solution executor can be searched for using the context-aware execution domain obtained though the procedures of this section.

4.5 Algorithm

Following the formal model of the previous section, we solve the problem of fragment composition by 1) building a context-aware execution domain Σ_{CF} and 2) searching for its consistent solution executor for goal ρ.

The construction of the context-aware execution domain is pretty straightforward and relies on the definitions of grounding (Def. 25) and context-aware execution domain (Def. 27). Once the resulting STS $\Sigma_{CF} = \langle \mathcal{S}_{CF}, s^0_{CF}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_{CF} \rangle$ is obtained it becomes a planning domain $D = \Sigma_{CF}$. For our convenience in this section we will omit the indices and denote the domain as follows: $D = \langle \mathcal{S}, s^0, \mathcal{I}, \mathcal{O}, \mathcal{R} \rangle$. The initial state of D becomes the initial state of the planning problem $I = s^0$, and the goal states are all states of the domain that satisfy ρ, that is $G = \{s \in \mathcal{S} : s \models \rho\}$. As such, we obtain a conventional planning problem $\{D, I, G\}$.

Once a planning problem is obtained, a consistent solution executor is derived by the algorithm for strong planning in asynchronous domain presented in [18]. In the following, we recap the description of this algorithm

and proof of its correctness and completeness. Although the most part of the content of this section can be found in [18], we find it necessary to be explicitly present it here since it is crucial for understanding the proofs of Chapter 7.

```
1  function plan(I,G)
2    OldSA := Fail
3    SA := ∅
4    while (OldSA ≠ SA ∧ I ∉ (G ∪ StatesOf(SA)))
5      Pr := StrongPreImage(G ∪ StatesOf(SA))
6      NewSA := PruneStates(Pr, G ∪ StatesOf(SA))
7      OldSA := SA
8      SA := SA ∪ NewSA
9    done
10   if (I ∈ (G ∪ StatesOf(SA)))
11     return SA
12   else
13     return Fail
14   fi
```

Figure 4.8: Search algorithm

The routine for searching consistent solution executor is presented in Fig. 4.8. In it we assume that the domain D is globally available, while we explicitly pass to it initial states I and goal state G. The algorithm is a fix-point iteration that incrementally constructs a state-action table SA, that indicates which action has to be executed in certain state of D in order to reach a goal state. As such, SA encodes all transitions of the domain that can potentially be presented in the consistent solution executor. SA is initially empty and grows at each iteration by adding state-actions which unconditionally lead to the states that are already covered by SA or goal states (i.e., states $\textsc{StatesOf}(SA) \cup G$). The termination of the algorithm

is caused by either the situation when 1) no new states are included in the next iteration or 2) the current state-action table already contains all initial states I, which actually means that the solution for the initial states is already found.

The algorithm is defined such that it explicitly deals with the constraints imposed by consistent solution executor (Def. 22). This logic is essentially realized by the key primitives StrongPreImage and PruneStates.

StrongPreImage is the basis of the backward search. For a subset S of states of Σ_{CF}, StrongPreImage returns a set of state-action pairs $\{\langle s, a\rangle\}$ that encode all transitions of Σ_{CF} that immediately lead to S. It takes into account that uncontrollable actions can be neither controlled nor predicted. So the function guarantees that ones a state-action $\langle s, a\rangle$ is included in the table, states of S can always be reached from s despite of nondeterminism. The primitive is defined as follows:

$$\begin{aligned}\textsc{StrongPreImage}(S) = \; & \{\langle s, a\rangle : (a \in \mathcal{I}) \wedge (\exists (s, a, s') \in \mathcal{R} : s' \in S) \wedge \\ & \qquad (\nexists (s, a', s'') \in \mathcal{R}) : a \in \mathcal{O})\} \bigcup \\ & \{(s, a) : (a \in \mathcal{O}) \wedge (\exists (s, a, s') \in \mathcal{R} : s' \in S) \wedge \\ & \quad \forall (s, a', s'') \in \mathcal{R} : (a' \in \mathcal{O}) \rightarrow (s'' \in S))\}.\end{aligned}$$

In order to properly reflect the requirements imposed by the definition of consistent solution executor (Def. 22), controllable and uncontrollable actions are treated differently. For example, when we include controllable state-action, not only do we check that it leads to the states that are already in the state-action table but also make sure that uncontrollable actions are not available from the same state. Similarly, the way we treat uncontrollable actions guarantees, that none of the uncontrollable actions originating from the same state are disregarded. Consequently, the strong pre-imaging function significantly contributes to the satisfaction of condi-

tion 1 (executor is runnable) of Def. 22 of consistent solution executor. We remark that this planning algorithm is significantly different from the conventional strong planning algorithms (e.g., [32]) that treat all the actions of the planning domain uniformly.

PruneStates function is responsible for removing from the current pre-image all the state for which the solution is already available (i.e., those that are already included in the state-actino table). It is defined as follows:

$$\textsc{PruneStates}(\gamma, S) = \{\langle s, a\rangle \in \gamma : s \notin S\}.$$

We remark that the purpose of the pruning goes beyond avoiding the duplication of the same state-actions in the resulting table. The pruning ensures that for each state no more than one controllable state-action is included which closely relates to conditions 1 of consistent solution executor. It also guarantees that the state-action table does not contain loops (conditions 3). Another property of the pruning that has nothing to do with the definition of consistent solution executor is that only the shortest solution from any state appears in the state-action table.

The resulting *state-action table* (a collection of state-action pairs) shows how the resulting executable process should behave in different states (i.e., which actions and where must be executed). Uncontrollable state-actions indicate which uncontrollable actions have to be expected in the respective state. Similarly, controllable state-actions indicate which controllable action has to be executed from the respective state. The consistent solution executor can be directly derived from the state-action table.

In the following we show that the main routine always terminates and is correct and complete. The correctness of the algorithm means that the STS derived from the state-action table built by the algorithm is always a consistent solution executor for the given problem. The completeness of the algorithm means that whenever a state-action is not found (Fail is

returned), the consistent solution executor for the given problem does not exist.

In order to prove the termination we show that at every iteration the size of the state-action table monotonically grows and the number of states is finite.

Theorem 1 (Termination). *Let $D = \langle \mathcal{S}, s^0, \mathcal{I}, \mathcal{O}, \mathcal{R} \rangle$ be a context-aware execution domain, let $I = s^0$ be its initial state and let $G \subseteq \mathcal{S}$ be a set of goal states. The execution of* PLAN(I,G) *on D terminates.*

Proof. We observe that $\mathcal{S}$ contains finite number of states. From the definition of the algorithm it can be seen that the within the main cycle (lines 4-9) the state-action table SA monotonically grows up (line 8) Therefore we can conclude that after at most $|\mathcal{S}|$ iterations no new state-actions can be added to SA. Consequently, the program exits the loop due to violation of looping condition $OldSA \neq SA$, and so PLAN(I,G) always terminates. □

In order to prove the correctness and completeness we actually prove that at every iteration of the main cycle the states that are included in state-action table are those for which a consistent solution executor already exists. In other words, if any state within state-action table was the initial state, the consistent solution executor for it would be (a part of) the current state-action table.

The STS associated with state-action table SA is defined as a set of states and transitions that can be traversed from the initial state of D using state-actions of SA:

Definition 28 (STS associated with state-action table).
Let $SA = \{\langle s, a \rangle : s \in \mathcal{S}, a \in \mathcal{I} \cup \mathcal{O}\}$ be a state-action table over some STS $D = \langle \mathcal{S}, s^0, \mathcal{I}, \mathcal{O}, \mathcal{R} \rangle$, let I be set of initial states so that $I = s^0$ if $\exists \langle s, a \rangle \in SA : s = s^0$ and $I = \emptyset$ otherwise, and let $G \subseteq \mathcal{S}$ be a set of goal states.

The STS associated with SA, I, G on STS D, denoted $D \triangleleft (SA, I, G) = \langle \mathcal{S}_{SA}, I, \mathcal{I}, \mathcal{O}, \mathcal{R}_{SA} \rangle$*, is defined as follows:*

1. $I \subseteq \mathcal{S}_{SA}$;
2. *if* $s \in \mathcal{S}_{SA}$ *and there exists a sequence* $\langle s_0, a_0 \rangle, ..., \langle s_n, a_n \rangle$ *such that* $s_0 = s$, $\langle s_i, a_i \rangle \in SA, i \in [1, n] : (s_i, a_i, s_{i+1}) \in \mathcal{R}$*, then* $s_n \in \mathcal{S}_{SA}$;
3. $\mathcal{S}_{SA}$ *contains no other states;*
4. *if* $s \in \mathcal{S}_{SA}$ *and* $(s, a, s') \in \mathcal{R}$*, then* $(s, a, s') \in \mathcal{R}_{SA}$.

Lemma 2 (Invariant property of the state-action table)**.**
After the i-th iteration of the main loop of routine PLAN(I,G)*, set of states* $G \cup StatesOf(SA)$ *contains all the states for which a consistent solution executor of depth up to i exists. In particular, if* $s \in G \cup StatesOf(SA)$*, then* $D \triangleleft (SA, I, G)$ *is a consistent solution executor for s.*

Proof. To prove the lemma we use induction on the number i of iterations of the main loop of routine PLAN(I,G)(lines 4-9).

Basis ($i = 0$**)**. Since SA is initialized to empty set, the state set $G \cup StatesOf(SA)$ contains only goal states G and for them a solution exists that requires 0 steps to achieve the goal (i.e. for which a consistent solution executor of depth 0 exists).

Induction step. Assuming that the theorem holds for the i-th iteration, we prove that it also holds for the $i+1$-th iteration. According to the definition of StrongPreImage (called at line 5), a state-action $\langle s, a \rangle$ will be included in $Pr_{i+1} =$ StrongPreImage($G \cup$ StatesOf(SA)) only if states $G \cup$StatesOf(SA) are achievable from s through some controllable transition (in this case it is a transition labeled with action a) or through all possible outgoing uncontrollable transitions of s (one of them is labeled with a). As a result, we can conclude that the goal is achievable from

StatesOf(Pr_{i+1}) in at most $i+1$ steps. Indeed, from StatesOf(Pr_{i+1}) there exists a possibility to strongly reach states $G \cup$ StatesOf(SA) in one step, while according to the inductive hypothesis for every state of $G \cup$ StatesOf(SA) the state transition system $D \triangleleft (SA, I, G)$ already contains a consistent solution executor of depth up to i. Given the inductive hypothesis and the correspondence between the definition of StrongPreImage and the conditions of the consistent solution executor in Definition 22, at the $i+1$-th iteration there will be extracted exactly the state-actions corresponding to a consistent solution executor of depth $i+1$). As a result, only state-action pairs in Pr_{i+1} for which the goal is reachable in no more than $i+1$ steps will be store to $NewSA$ (line 6). Such state-action pairs are added to SA (line 8), which consequently will hold all of the state-action pairs from which the goal is reachable in up to $i+1$ steps. □

Theorem 2 (Correctness and Completeness).
If plan(I,G) *returns state-action table SA, then $D \triangleleft (SA, I, G)$ is a consistent solution executor to the respective composition problem. If* plan(I,G) *returns* Fail*, then no consistent solution executor to the respective composition problem exists.*

Proof. The proof directly follows from Lemma 2 and Definition 22 of consistent solution executor. The consistent solution executor guarantees that a goal state will be unconditionally reached from the initial state in a finite number of steps. It essentially means that at some iteration of the algorithm it will be included in the state-action table (i.e., $I \in G \cup$ StatesOf(SA)) and SA, encoding consistent solution executor will be returned. On the contrary, if a consistent solution executor does not exist, Fail will be eventually returned. □

The further information on algorithm implementation and evaluation

will be given in Chapter 8.

4.6 Discussion

The ability to handle abstract composition requirements defined in detachment from details of fragment implementations is the main novel feature of our fragment composition approach presented in this chapter. Due to the explicit context model being one of the central elements of our formalism, we call our approach context-aware.

One of the key advantages of using abstract requirements that comes directly from our adaptation framework presented in Chapter 3, is the ability to integrate our engine with the adaptation framework, which automatically derives composition requirements in terms of context. However, there is one more advantage that can be considered separately from the process adaptation problem and that has already been articulated in Section 4.1. When composition requirements are expressed over context, they are essentially detached from the fragments, but any fragment that is properly annotated over the same context model can be used in combination with them . As we will show, this point can be used, for example, in user-centric systems, where requirements are expressed by the technical expert (designer) while the choice of fragments/services is made by the end user. The further composition can run without any intervention from technicians, since in our formal framework the predefined abstract requirements will successfully be grounded on the fragment/service implementation chosen at run time.

We find our composition model quite intuitive to be effectively mastered and used by service/fragment designers (especially, in the presence of graphic modeling tools that seem to be easy to implement). At the same time, our context model is expected to allow for many advanced

composition features bringing the automated service composition to a new level. In particular, in Chapter 7 we describe 1) a simple yet expressive context-based language for control-flow requirements that goes beyond the reachability of context states, 2) a prototypal approach to integrating data-flow requirements into the engin. In [60] it can also be found a prototypal approach to automatic derivation of composition interface/protocol.

Chapter 5

Composition in Dynamic Environment

In this chapter we show how the composition engine presented in Chapter 4 can be exploited by the adaptation framework presented in Chapter 3 to realize dynamic adaptation. In Section 5.1 we discuss the operation of the integrated system. For that purpose, we define the notion of system configuration completely describing the status of the system and show all possible ways the system configuration can evolve. Section 5.2 is devoted to the issues of executing composite processes in the dynamic environment. In particular, we analyze the most important dynamic factors featured by the execution environment of the CLS and discuss the ways we can handle them from the perspective of running instances of composite processes. The execution of composed processes in dynamic environments is a more general topic that can also be considered out of the scope of our adaptation framework.

5.1 System Operation

In this section we mostly reuse the definitions given in Chapter 4, however some additional elements are needed. In particular, some adaptation mech-

anisms and strategies require extra annotations to be added to fragments. As a result, the standard effect, precondition and goal annotations, which preserve their meaning and semantics, are accomplished with compensations and decision points. *Compensation* associates a fragment action with a set of context states where the effect of this action is considered to be negated (i.e., compensated). For example, if the fragment action is devoted to booking a place at a storage area (context property "storage ticket" goes to state "booked"), its compensation may naturally be to have the same context property in state "not booked". *Decision points* are states of fragments where critical decision within a fragment are supposed to be made (e.g., branching point in a fragment). Such points can be considered as a target for backward adaptation (rollback), where we would like to revert the process in hope a different decision will be made afterwards. The examples of both annotations can be found in Section 3.2. The new extensions do not affect the way the composition is performed and are not reflected in the composition framework. They are used exclusively for adaptation purposes. Formally, the updated definition of fragment annotation is as follows:

Definition 29 (Fragment Annotation (Extended))**.** *Let* $f = \langle \mathcal{S}, s^0, \mathcal{I}, \mathcal{O}, \mathcal{R} \rangle$ *be a fragment and let* C *be a context. An annotation of fragment* f *over context* C *is a tuple* $\omega_f = \langle \mathcal{P}, \mathcal{E}, \mathcal{G}, \mathcal{C}, \mathcal{B} \rangle$*, where:*

- $\mathcal{P} : \{\mathcal{I} \cup \mathcal{O}\} \rightarrow R_C$ *is the precondition labeling function;*
- $\mathcal{E} : \{\mathcal{I} \cup \mathcal{O}\} \rightarrow E^*_C$ *is the effect labeling function. Any action effect* $\mathcal{E}(a)$ *may contain no more than one event per context property, i. e., for any context property* $p = \langle L, l^0, E, T \rangle \in C$ *the following holds:* $\nexists e_1, e_2 \in \mathcal{E}(a) : e_1, e_2 \in E$*. Moreover, if* $\mathcal{E}(a) \neq \emptyset$ *then* $\mathcal{G}(a) = \emptyset$ *(i.e., an action can be annotated either with a goal or with an effect);*
- $\mathcal{G} : \{\mathcal{I} \cup \mathcal{O}\} \rightarrow R_C$ *is the goal labeling function, such that* $\mathcal{G}(a) \neq \emptyset$

only if $\mathcal{E}(a) = \emptyset$ *(i.e., an action can be annotated either with a goal or with an effect);*

- $\mathcal{C} : \{\mathcal{I} \cup \mathcal{O}\} \rightarrow R_C$ *is the compensation labeling function;*
- $\mathcal{B} \subseteq \mathcal{S}$ *is a set of decision points.*

The following definition captures the status of the execution of an adaptable process. As illustrated in Fig. 3.4, an adaptable process is usually a *hierarchical structure*, where on top of a core process other processes related to adaptation tasks may exist. Complex adaptation and "adaptation of adaptation" may require multiple levels in such hierarchy. Each element of the hierarchy is a triple process-state-history describing the configuration of an elementary process. In the triple, *process* is a fragment as defined in Def. 7 describing a process model, *state* is the current state of the process instance and *history* is a partial run of the process describing its execution history. In order to intuitively introduce adaptation mechanisms and strategies, we model process configuration of an adaptable process as a stack of triples. The bottom (first) triple refers to the core process and all the others refers to adaptation processes. The top (last) triple in the stack is the one that is currently under execution. Triples can be pushed to the stack when process adaptation is performed and can be popped from the stack when, e.g., the process instance of the top triple terminates. Formally:

Definition 30. *(Process Configuration) A process configuration is a non-empty stack of triples* $\phi = (w_1, s_1, h_1), (w_2, s_2, h_2), \ldots, (w_n, s_n, h_n)$*, where:*

- $w_i = \langle \mathcal{S}_i, s_i^0, \mathcal{I}_i, \mathcal{O}_i, \mathcal{R}_i \rangle$ *is a process fragments;*
- $s_i \in \mathcal{S}_i$ *is a current state in the corresponding process fragments;*

- *h_i is a partial run of w_i terminating in s_i and representing the previous execution history of w_i.*

The configuration of the whole system is defined by the current configuration of the context properties, by the configuration of the processes in the system, and by the set of available fragments.

Definition 31. *(System Configuration) A system configuration is a tuple $\Theta = \langle C, F^+, l_C^0, \Phi \rangle$, where:*

- *C is the context;*
- *F^+ is the set of fragments available in the system;*
- *l_C^0 is the current state of context C;*
- *$\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ are configurations of all processes running within the system.*

5.1.1 General Adaptation Problem

The core of our adaptation framework is the fragment composition engine presented in Chapter 4. Our framework is designed such that all the adaptation mechanism can be realized through fragment composition. As a result, the formal definition of a *general adaptation problem* is structurally very close to the definition of the problem of context-aware fragment composition:

Definition 32 (General Adaptation Problem)**.** *A general adaptation problem is a tuple $\xi = \langle F^+, C, l_C^0, \rho \rangle$, where:*

- *F^+ is a set of fragments annotated over context C;*
- *l_C^0 is the current state of context C;*
- *$\rho \in R_C$ is a context formula describing the set of goal configurations.*

The resolution of a general adaptation problem $\xi = \langle F^+, C, l_C^0, \rho\rangle$ consists in building a context-aware fragment composition for $\Psi = \langle F^+, C\rangle$ and ρ (see Def. 23) considering l_C^0 as the initial context state.

All the adaptation mechanisms and strategies used in our adaptation framework (see Section 3.2) essentially consist in deriving certain general adaptation problems from the current system configuration, resolving them using the composition engine and executing the compositions obtained (we call them *adaptation processes*). *Various adaptation mechanisms and strategies differ only in how the corresponding general adaptation problems are derived.*

5.1.2 Evolution of System Configuration

To describe the dynamic aspects of system operation we show how the system configuration can evolve in time. We distinguish five different kinds of evolution, that cover both "normal" evolution and various unexpected (extraordinary) situations that may require process adaptations. These five types are: 1) exogenous events, 2) entity in/out, 3) fragment in/out, 4) process execution, 5) adaptation. The first three types of evolution correspond to dynamic changes in the environment that affect the way existing processes are executed and adapted. The last two types correspond to the "normal" execution of adaptable processes, which include the execution itself, and the adaptation.

In the following we take a closer look at all of them. We assume that the system configuration is $\Theta = \langle C, F^+, l_C^0, \Phi\rangle$ before elementary evolution and $\Theta' = \langle C', F^{+'}, l_C^{0'}, \Phi'\rangle$ after it.

Exogenous Events

Exogenous events correspond to changes of the current state of the context that are not triggered by process execution. As such, within the related

type of system evolution only the current context state changes: $l_C^{0'} \neq l_C^0$.

Entity In/Out

In our adaptation framework, a set of active entities constantly evolves, with new entities entering the scenario and existing entities exiting it. Since an entity has a complex structure comprising process(es) attached to it, a set of fragments advertised to other entities, and a set of context properties describing the entity, once an entity enters or exits the scenario the system configuration may dramatically change. What is important is that these changes may affect the operation of other entities within the scenario.

An entity exiting the scenario may potentially cause troubles, mostly if other entities are dependent on its fragments, (e.g., if an activity refinement of some adaptable process engages a fragment belonging to the exiting entity). On the contrary, the entrance of a new entity may create new opportunities for the other entities by introducing new facilities in form of its fragments.

In general the changes in the set of entities operating within the scenario affects the set of available fragments F^+ (some new fragments are added or some existing fragments are removed), the global context model C (some new context-properties are added or some existing context properties are removed) and the set of configurations of running processes (some adaptable processes may be instantiated or unexpectedly terminated). The current states of context properties that do not belong to the entity entering/existing the scenario remain unchanged. Summarizing it, we can state that for this type of system evolution all the element of the system configuration may change: $l_C^{0'} \neq l_C^0$, $C' \neq C$, $F^{+'} \neq F^+$ and $\Phi' \neq \Phi$.

Fragment In/Out

Even if an entity does not enter/exit the scenario it can dynamically change the set of fragments advertised in order to temporarily enable/disable some types of collaboration (e. g., a treatment station may vary the set of treatment facilities available to cars by adding/removing the respective fragments to/from its set of available fragments). In this case, $F^{+'} \neq F^{+}$.

Process Execution

Let $\phi = (w_1, s_1, h_1), (w_2, s_2, h_2), \ldots, (w_n, s_n, h_n)$ be some process configuration such that $\phi \in \Phi$. As we discussed, the top triple is the one under execution. Consequently, the final configuration of the same fragment is supposed to be $\phi' = (w_1, s_1, h_1), (w_2, s_2, h_2), \ldots, (w_n, s'_n, h'_n)$, where the current state and the history of the top triple is updated. Taking into account that w_n is a fragment, the execution of w_n is possible if there exists an activity a_n available from current state s_n such that 1) the activity precondition holds in the current context $l^0_C \models \mathcal{P}(a)$ 2) a_n belongs to some fragment among F^+ (it is still available) and 3) a_n is not abstract (otherwise, the refinement is needed). We remark that a special case of execution is when a top triple terminates and is popped from the stack. In this case the final configuration is $\phi' = (w_1, s_1, h_1), (w_2, s_2, h_2), \ldots, (w_{n-1}, s_{n-1}, h_{n-1})$, i.e., the terminated triple is removed.

Since process execution changes the related process configuration ϕ for ϕ', a container set Φ also changes ($\Phi' \neq \Phi$). It is worth to mention that within Φ' more than one process may have its configuration updated (e.g., when the activity executed is a communication between two processes). Moreover, Φ' may have increased or decreased number of process configurations (in case activity execution instantiates a partner process and in case the bottom (core) triple of some process terminates respectively).

Concerning the other elements of the system configuration, process execution may also change the current context state according to the annotation semantics (see Section 4.2.2), so that $l_C^{0'} \neq l_C^0$. At the same time, the context model and the set of fragments remain unchanged.

Adaptation

Let $\phi \in \Phi$ be a process configuration of a process to be updated. The adaptation of $\phi = (w_1, s_1, h_1), (w_2, s_2, h_2), \ldots, (w_n, s_n, h_n)$ affects exclusively ϕ (i.e., $\phi' \neq \phi$) and does not affect the other elements of the system configuration (i.e., $l_C^{0'} = l_C^0$, $C' = C$, $F^{+'} = F^+$), nor does it affect other process configurations within Φ.

As a rule, adaptation results in new triple(s) added to the top of the stack (e.g., refinement of an abstract activity in n-th triple will add the refinement process as $n+1$-th triple). In addition to that, some changes are usually done to the triple whose execution caused adaptation (e.g., in case of refinement the top triple is changed as if the abstract activity was already executed, so that after the refinement process terminates, the execution of the process that contains the refined abstract activity will resume from the state after the respective abstract activity).

The way ϕ changes in result of adaptation heavily depends on the adaptation strategies and mechanisms applied. In the following, we describe these changes for each of the adaptation mechanisms mentioned in Section 3.2. The solution to the general adaptation problem $\xi = \langle F^+, C, l_C^0, \rho \rangle$ is denoted as $w = Adapt(\xi)$.

Refinement mechanism. The refinement mechanism is applied when the next activity a to be executed in the current triple is abstract. Let $\mathcal{G}(a)$ be the goal of a. In this case, the problem of finding activity refinement is formulated as a general adaptation problem $\xi_{abs} = \langle F^+, C, l_C^0, \mathcal{G}(a) \rangle$. In other words, the goal of the refinement process is to achieve one of the

context states that satisfy the goal formula of a from the current context state. Let $w_{abs} = Adapt(\xi_{abs})$ by an adaptation process form problem ξ_{abs}. The changes to process configuration will be the following:

$$\phi' = (w_1, s_1, h_1), \ldots, (w_n, s'_n, h'_n), (w_{abs}, s_{abs}, h_{abs})$$

, where triple (w_n, s'_n, h'_n) reflects the result of execution of abstract activity a in (w_n, s_n, h_n) and triple $(w_{abs}, s_{abs}, h_{abs})$ relates to the refinement process with s_{abs} being the initial state of process w_{abs} and h_{abs} being an empty execution history containing only state s_{abs}. It can be seen that after w_{abs} terminates the top triple will be removed from the configuration and the execution will proceed from the point of w_n immediately after the abstract action a, which will simulate the fact that a has been executed.

Local adaptation mechanism. The local adaptation is used when the precondition of the activity is violated. It is the simplest possible solution, aiming at bringing the context to a state from which the execution of the process can be resumed. Let next action be a with precondition $\mathcal{P}(a)$. The problem of local adaptation is formulated as a general adaptation problem $\xi_{local} = \langle F^+, C, l_C^0, \mathcal{P}(a) \rangle$. The goal of the refinement process is to achieve one of the context states that satisfy the precondition of a. Let $w_{local} = Adapt(\xi_{local})$ be an adaptation process for problem ξ_{local}. The final process configuration will be the following:

$$\phi' = (w_1, s_1, h_1), \ldots, (w_n, s_n, h_n), (w_{local}, s_{local}, h_{local})$$

, where triple (w_n, s_n, h_n) remains unchanged (the precondition is violated and the process gets stuck) and triple $(w_{local}, s_{local}, h_{local})$ relates to the local adaptation process with s_{local} being the initial state of process w_{local} and h_{local} being an empty execution history containing only state s_{local}. After w_{local} terminates the top triple will be removed from the configuration and the execution of w_n will proceed with action a, whose precondition in the "corrected" context already holds.

Compensation mechanism. The compensation mechanism is applied when it is necessary to compensate the effects of some actions executed before. Let the current triple be (w_n, s_n, h_n) and let the respective history be $h_n = (s_1^n, a_1^n, s_2^n, a_2^n, \dots, a_{m-1}^n, s_m^n)$ such that $s_n = s_m^n$. The compensation of an action a_{m-1}^n annotated with compensation goal $\mathcal{C}(a_{m-1}^n)$ can be formulated as a general adaptation problem $\xi_{m-1} = \langle F^+, C, l_C^0, \mathcal{C}(a_{m-1}^n)\rangle$ with the resulting compensation process $w_{m-1} = Adapt(\xi_{m-1})$. So the compensation process will bring the context to the state where the effect of action a_{m-1}^n is considered to be compensated. The final configuration will be:

$$\phi = (w_1, s_1, h_1), \dots, (w_n, s_n', h_n'), (w_{m-1}, s_{m-1}, h_{m-1}).$$

The new triple added on top of the n-th triple is related to the compensation of action a_{m-1}^n. Process w_n is partially "rollbacked" so that its current state is $s_n' = s_{m-1}^n$ and its history is $h_n' = (s_1^n, a_1^n, \dots, a_{m-2}^n, s_{m-1}^n)$. The further execution of the adaptable process will first compensate a_{m-1}^n, and then will come back to the execution of the n-th triple from the point to which it was rollbacked. It is clear that "pure" compensation does not make a lot of sense (it is rarely reasonable to compensate actions and then re-execute them), however compensation mechanism is extremely useful when combined with other mechanisms into an adaptation strategy.

We remark that often a part of a process has to be compensated. In this case, different implementations are possible. One of them is to use complex planning goals (e.g., [67]) in order to produce a "one-shot" compensation process that compensates all chain of activities. A simpler solution is to perform incremental compensation, where actions are compensated one-by-one.

As we already mentioned in Section 3.2, adaptation mechanisms can be further combined into strategies, where a few mechanisms are applied (in

sequence or in parallel) in order to resolve a single adaptation problem. In general, combining a few mechanisms is quite intuitive, though many alternative implementations of the same strategy may often be proposed. Here we give only one example of an adaptation strategy, which is the *rerefinement* strategy.

The idea of rerefinement is to compensate the execution of the current refinement of some abstract action and to refine it anew in order to produce a more up-to-date refinement. These strategy is applicable in situations where the execution of the current refinement cannot be completed, e.g., due to "bad" context or due to unavailability of some fragments exploited by it. In these case, the new refinement takes into account the most recent changes in the system configuration and produces a solution that is compatible with them.

Let $\phi = (w_1, s_1, h_1), \ldots, (w_{n-1}, s_{n-1}, h_{n-1}), (w_n, s_n, h_n)$ be the current process configuration. Here, triple (w_n, s_n, h_n) represents the refinement of some abstract action a_{abs} with goal $\mathcal{G}(a_{abs})$ of process w_{n-1}. We assume that at this point the execution of w_n got stuck and we decide to apply re-refinement. In order to perform it, we, first, compensate all actions of h_n and then produce a new refinement of the same action a_{abs} in new conditions. The compensation consists in generating an adaptation process for a general adaptation problem $\xi_i^{CMP} = \langle F^+, C, l_i^0, \mathcal{C}(a_i^n)\rangle$, executing it and repeating the same steps for all activities of h_n in backward order. After that, we come to the point when w_n is compensated and the problem of new refinement is formulated as a general adaptation problem $\xi'_{abs} = \langle F^{+'}, C', l_C^{0'}, \mathcal{G}(a_{abs})\rangle$. Its solution is process $w'_n = Adapt(\xi'_{abs})$. We remark that although w'_n is produced to achieve the same goal as w_n, the two processes are not generally the same since all other elements of their original general adaptation problems may be different. So the resulting process configuration is $\phi = (w_1, s_1, h_1), \ldots, (w_{n-1}, s_{n-1}, h_{n-1}), (w'_n, s'_n, h'_n)$, where

triple (w'_n, s'_n, h'_n) represents new refinement of a_{abs}.

5.1.3 Managing Adaptation Strategies

As we showed in the previous section, the adaptation of business processes is often an extremely complex procedure that has to take into account multiple factors. In our adaptation framework, the most basic elements of the adaptation logic are adaptation mechanisms. We showed that all of them can be expressed as a general adaptation problem, which is resolved through fragment composition. In turn, adaptation mechanisms can further be combined into adaptation strategies, which can be even more diverse and complex. We notice that even the same adaptation strategy may have multiple implementations that work differently in different situations.

An important open issue here is how to manage different strategies defined within the framework and how to decide on which strategy should be applied in a particular situation and how this has to be done. Ideally, the management of adaptation strategies has to take into account multiple factors such as 1) the application domain, 2) the type of problem triggering the adaptation, 2) the current system configuration, 4) non-functional properties of fragments and processes 5) the previous adaptation history of the process etc. The adaptation management can be way more complex then just deciding on which strategy to use. For instance, the multiple strategies may be chosen and ordered according to their expected efficiency in the current case. Once the most preferable strategy fails, the ones with lower priority may be actualized. Alternatively, simultaneous strategy simulations may be performed in order to empirically choose the one that performs better for the given adaptation case. We expect that other approaches can be proposed here.

Our first steps in the direction of the management of adaptation strategies showed that our adaptation framework is a promising platform for

implementing various strategies and experimenting with them. Although we admit that adaptation strategy management goes beyond the scope of the dissertation, we still have to come up with some basic solution in order to be able to implement the demonstrator. The idea here is to try to solve problems as "locally" as possible. For example if a problem in some refinement happens, we try to resolve it within this refinement, without "jumping" to a higher level of abstraction, where the associated abstract activity is located. In other words, the process repair should be done to affect as smaller portion of the whole large adaptable process as possible. In our implementation we consider two types of inconsistencies: unrefined abstract activities and precondition violation. The problem of unrefined abstract activity is always resolved by simple refinement (for now, we avoid situation where refinement cannot be found). Assuming that the process currently under execution is w, for the precondition violation the following set of rules is proposed:

1. Try to apply local adaptation;

2. If 1 does not work and w is itself a local adaptation for some activity a_{parent} of process w_{parent}, skip the execution of w and try to resume the execution of w_{parent};

3. If 1 does not work and w is itself a refinement for some activity a_{abs} of process w_{parent}, compensate w and resume the execution of w_{parent}.

It can be observed, that using these rules the adaptation is attempted to be performed in the very limited scope first (local adaptation is essentially adaptation in the scope of particular problematic activity). Then, if it is not possible, we widen the scope of adaptation and try to resolve a problem more globally, for example, in the scope of the parent process (which is one level higher in the hierarchy of process configuration), and

so on. In case of rule 2, if w is itself and local adaptation process, we skip it and try to resume the execution of the parent process. It is very likely that we will still have the precondition violation for a_{parent} (indeed the problem resolution implemented by w has not been completed) and so the resolution of precondition violation of a_{parent} will be restarted anew and the same rules will be applied. In case of rule 3, once the local adaptation is not possible and w is a refinement for some abstract activity a_{abs} of process w_{parent}, we compensate w and jump back to the level of the w_{parent} to try to re-refine a_{abs}.

5.2 Fragment Composition in Dynamic Environment

It can be easily observed that the fragment composition algorithm of Chapter 4 exploited by our adaptation framework actually operates under a number of strict assumptions that let it neglect the dynamic factors of the execution environments (e.g., exogenous events, fragments unavailability etc). As a result, the correctness of the produced compositions is guaranteed only in static environment where these assumptions hold. Consequently, taking into account that dynamic changes in the execution environment are quite frequent (e.g., entities constantly enter and exit the scenario) and the composition is a process whose execution may last for long, we cannot avoid the discussion of how the dynamic factors affect the ability of the compositions to achieve their original goals.

We identify three basic assumptions that are explicitly made by the fragment composition algorithm and related to the dynamicity of the execution environment:

1. *Controllable Context.* The context can evolve only as a result of process execution. No exogenous events are possible;
2. *Static Fragment Set.* The set of fragments remains unchanged;

3. *Pure Refinement.* The abstract activities in the fragments can always be refined such that they achieve activity goal in minimalistic way (without side effects).

We remark that the violation of these assumptions does not necessarily lead (and often does not lead) to the process failure. In big application domains, a process depends only on a small portion of the execution environment and if the related elements are not affected (or if they are not affected in critical way) the process execution can still be successful. For example, the process for parking a car at the storage area has nothing to do with treatment facilities and so if a treatment facility becomes unavailable it does not in any way affect the parking process. However, in some cases assumption violation results in problems that can be resolved only through process adaptation.

There are two conceptually different ways to increase composition robustness against dynamic factors: (i) to change the composition approach so that it considers dynamic factors while planning for a solution and (ii) to design the adaptation framework such that once critical changes happen at run time they are automatically addressed using the run-time adaptation.

From our point of view, the first solution cannot successfully solve the problems related to dynamicity. For example, the problem of dynamic set of available fragments cannot be addressed by this approach in principle. Indeed, we cannot make predictions at composition time about which fragments will disappear and especially about which new fragments will emerge in the system when a real problem occurs. The same is true with the other assumptions. Another important point is that the attempt to take into account all possible exogenous events at all steps of execution will dramatically increase the size of the domain and slow down the composition procedure. What is more frustrating is that the reactions to exogenous events will still be vulnerable against dynamic factors such as volatile set

of available fragments. Then the question arises: does it make sense to calculate in advance solutions to dozens of different improbable problematic situation if we know that 1) most of these solutions will never be demanded and 2) they can be easily broken by other dynamic factors and will not work when they are needed. It is also worth to mention that the attempt to deal with system dynamicity at design time does not follow the flavour of "dynamic adaptation", where a problem is addressed as soon as it happens and the run-time information is intensively used to produce a better solution.

This is why in our adaptation framework we follow the second approach. We treat all processes running within a system (including the adaptation processes themselves) as potential target for adaptation and constantly track their execution. For example, if a problem occurs in a running process w, the adaptation engine automatically produces an adaptation process w_x that is supposed to address the problem. While running w_x, new dynamic changes may happen and so the adaptation process may get stuck itself. In this case, the adaptation engine may try to repair the adaptation process ("adaptation of adaptation") using the same adaptation mechanisms and strategies as for the core process w and so on. Alternatively, as we discussed in the previous section, more sophisticated adaptation management may take into account the previous adaptation history of a process. For example, if process w_x realizes local adaptation for process w and at some point w_x gets stuck due to precondition violation, knowing that w_x is already a local adaptation process, the adaptation manager may decide to skip it and to produce a new adaptation process w'_x with the same goal but for updated environmental conditions. In general, we presume that with advanced management of adaptation strategies it is possible to achieve extreme robustness against dynamic factors even using "static" fragment composition.

Our adaptation framework never reacts directly to dynamic changes in the execution environment and always continues execution normally. Only when the process execution cannot proceed due to known adaptation triggers (i.e., precondition violation, fragment not available, unrefined abstract activity), the adaptation is applied. As such, we assume that all critical changes will sooner or later trigger adaptation and will be properly addresses by the respective adaptation process. We also admit that certain level of proactiveness could be of much help here (though we do not use it in our demonstrator and consider it as potential direction for future research). In the rest of this section we have a short discussion of the three assumptions introduced above and show the problems that can arise once the assumptions are violated. We also briefly outline possible proactiveness enhancements to the adaptation manager that can improve efficiency in this case.

Controllable Context. In the lack of exogenous events, the context evolves precisely as it is defined by the contextual impact (see Defs. 13, 15, 16). Automatically composed processes are guaranteed to be correct only if the context is controllable. Once exogenous event happens, the process may get stuck due to precondition violation, which can be dealt with in normal way. Sometimes it makes sense to perform process simulation as a reaction to exogenous events. In this case, the harmful effect of these events can be detected a priori and more efficient adaptive actions may be performed. For example, if a car is about to start driving towards the storage area, and the storage area becomes full, it seems to be unwise to let the car drive to the storage area even though we know a priori it will not be able to get stored there. By simulating the process we can actually identify the problem before moving the car to the area and so more efficient adaptation may be proposed. Alternative solution (which is actually used in the demonstrator) is to provide proactive fragment annotations (e. g.,

to specify that driving to the storage area can be performed only when the latter is not full: the only reason to drive to the storage are is for store the car over there).

Static Fragment Set. The problem of fragment unavailability, as a rule, has more serious consequences for a process using unavailable fragment. This problem can hardly be addressed at design time. At run time, the fragment unavailability usually results in process compensation and replanning (clearly enough local adaptation does not resolve the problem). The proactiveness can drastically improve the adaptation efficiency and can be easily provided by checking all the activities in the process every time some fragment disappears.

Pure Refinement. In fragment composition, abstract activities are always treated as "black boxes" that act according to their goals. As a result, it is never checked at composition time if the abstract activities within a composition are refinable and whether they produce side effects. The proactiveness can be achieved here, for instance, by checking the refinability of abstract actions after the composition is produce but before its execution starts. In this way we can identify potential problems and use recomposition to avoid them. Concerning the side effects, they can be treated in the same way as exogenous events.

We remark that many additional aspects related to the static fragment composition and consequent execution in a dynamic environment may arise in real setting. For example, the inability to refine an abstract activity may be the direct consequence of unavailability of some fragments. The proactive checking of refinability should then be additionally performed every time the set of available fragments changes. One more example is where we have two sequential activities a_1 and a_2 with different preconditions. Once the precondition of a_1 is violated, a local adaptation process w_1 is

produced and executed. However, in addition to satisfying the precondition of a_1 it, as a side effect, may violate the precondition of a_2, and so a more intelligent adaptation may be needed in this case.

Similarly to the management of adaptation strategies, we have to admit that the efficient treatment of dynamic factors in automated systems is by itself a large research area that cannot be completely covered in this dissertation. However, we find this discussion important to understand the advantages of our adaptation framework and to come up with some basic ideas for our implementation of the adaptation engine.

In the conclusion, we would like to emphasize, that we consider the simplifying assumptions in fragment composition algorithm to be rather an advantage than a disadvantage of our approach. In fact, these assumptions are a way to simplify the specification of core processes and to postpone the resolution of potential execution problems to the very last moment, when they really occur and when we have the most up-to-date information about the environment and thus can produce better solutions. Finally, this makes it possible to create a truly run-time adaptation engine that provides unprecedented level of flexibility and robustness in dynamic environments.

Chapter 6

Context-Aware Composition of Services

In this chapter we present the context-aware model for service composition in dynamic environments which strongly relies on the fragment composition model of Chapter 4. We start from the motivating example that is substantially different from the one presented in Section 3.1. After that we discuss the difference between the notion of service and the notion of fragment and show that under certain reasonable assumptions the composition of services is very close to the composition of fragments. Consequently, we show that the context-aware composition of services can actually be performed following the approach of Chapter 4.

We remark that in this section we consider simple context-aware service composition, where composition goals are reachability goals for context states (the same as in Chapter 4). For the time being, we also neglect data-flow requirements. In Chapter 7 we will discuss advanced control- and data-flow requirements in context-aware service composition, based on the motivating example and the background material of this chapter.

6.1 Motivating case study

In order to illustrate the advantages of our approach and to exemplify its concepts we use a well-known motivating example from the travel domain (later referred to as Virtual Travel Agency or VTA scenario). Two independent service providers that have no direct communication with each other provide sets of services for managing flight tickets and hotel reservations respectively. In particular, with the services of the flight company flight tickets can booked and canceled, while with the services of the hotel reservation company hotel reservations can be made, modified and canceled. In addition to that, there is a third-party service that tracks flights and sends notifications about flight delays and cancellations. In order to help the user conveniently manage a flight ticket and an associated hotel reservation as a single travel package a service composition is needed. The composition has to support the complete life cycle of the package, from its acquisition, to possible modifications and cancellations.

The details of the service protocols are given in Figures 6.1, 6.2, 6.3, 6.4, 6.5 in form of abstract BPEL processes. Flight Booking Service (Fig. 6.1) accepts a flight ticket request and checks for the ticket availability. If tickets are available it asks for the user's confirmation of booking. Flight Cancellation Service (Fig. 6.3) is a simple request-response service. Hotel Management Service (Fig. 6.4) implements all three operations associated with hotel reservations. Hotel reservation and modification resemble the flight booking procedure and hotel cancellation resembles flight cancellation. Finally, there is a Flight Notification Service (Fig. 6.2) that can send notifications of two types: "flight is delayed" and "flight is cancelled".

The interaction with the user is modeled as another service (Fig. 6.5), which essentially determines the communication protocol of the future service composition.

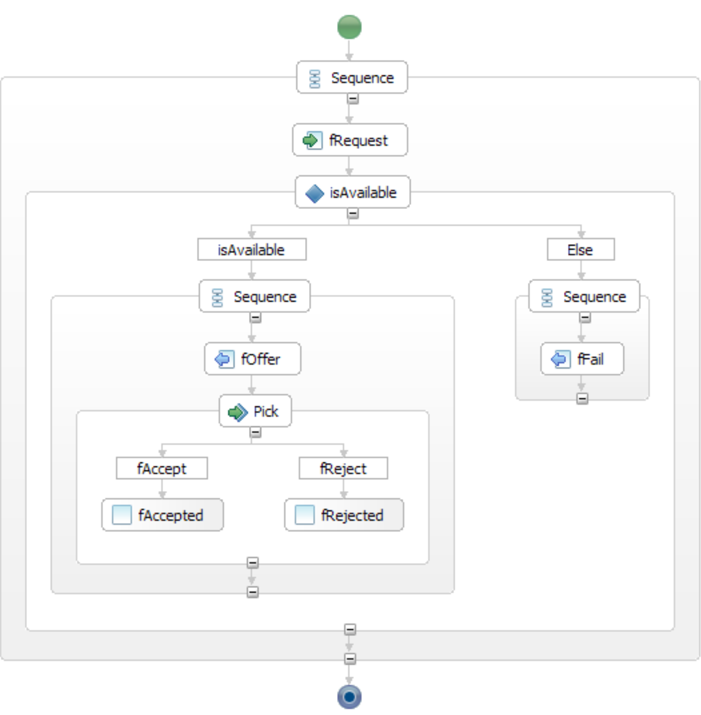


Figure 6.1: Flight Booking Service

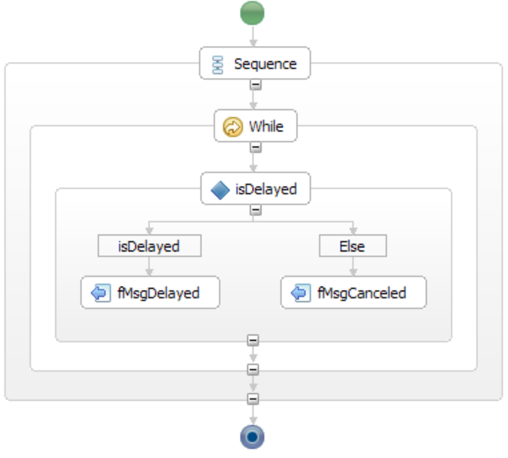


Figure 6.2: Flight Notification Service

The composition requirements can be expressed in natural language as follows:

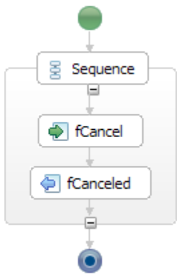


Figure 6.3: Flight Cancellation Service

1. The initial objective is to purchase a travel package requested by the user. A flight and hotel must be booked transactionally (incomplete packages of only hotel or flight are impossible) and aligned in time and location so that hotel location coincides with flight destination and hotel check-in date corresponds to the flight arrival date (to simplify the resulting model we consider a one-way ticket only);

2. Upon the delay of the purchased flight, the hotel reservation has to be aligned to the new arrival date. If it is not possible, the whole package has to be cancelled, which means the transactional cancellation of both the flight and the hotel reservation;

3. Upon the flight cancellation, the hotel reservation has to be cancelled as well.

We remark that more complex and close-to-reality variants of these scenarios can be modelled with our framework. However, even this simple variant features a number of important issues that have never been addressed altogether by the existing automated composition techniques.

First of all, the services within the scenario are stateful components featuring asynchronous interaction, partial observability and nondeterminism.

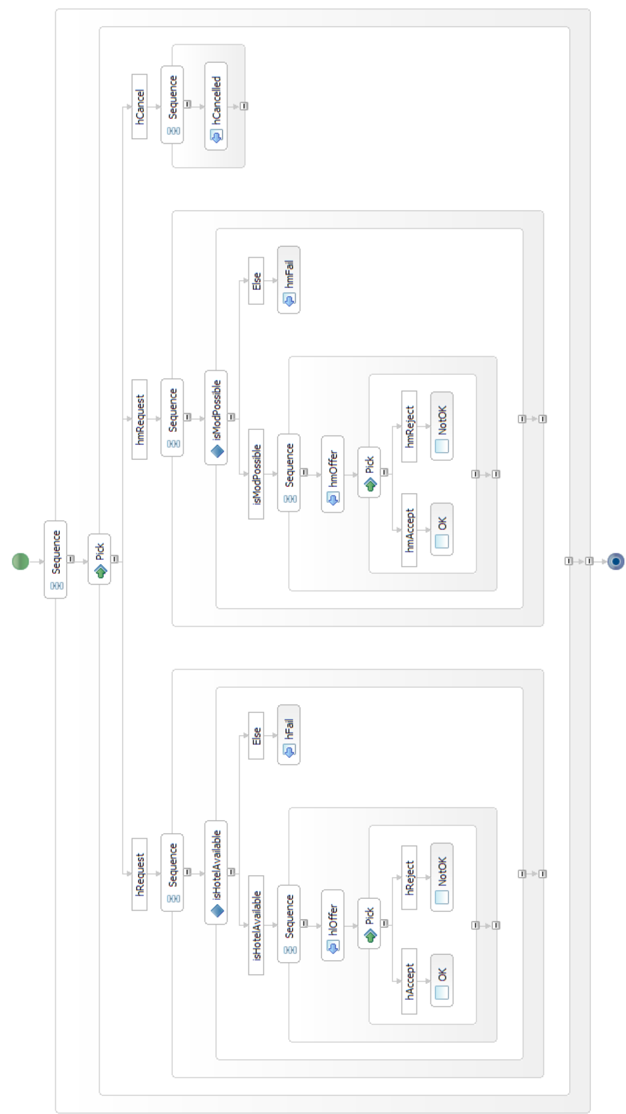


Figure 6.4: Hotel Management Service

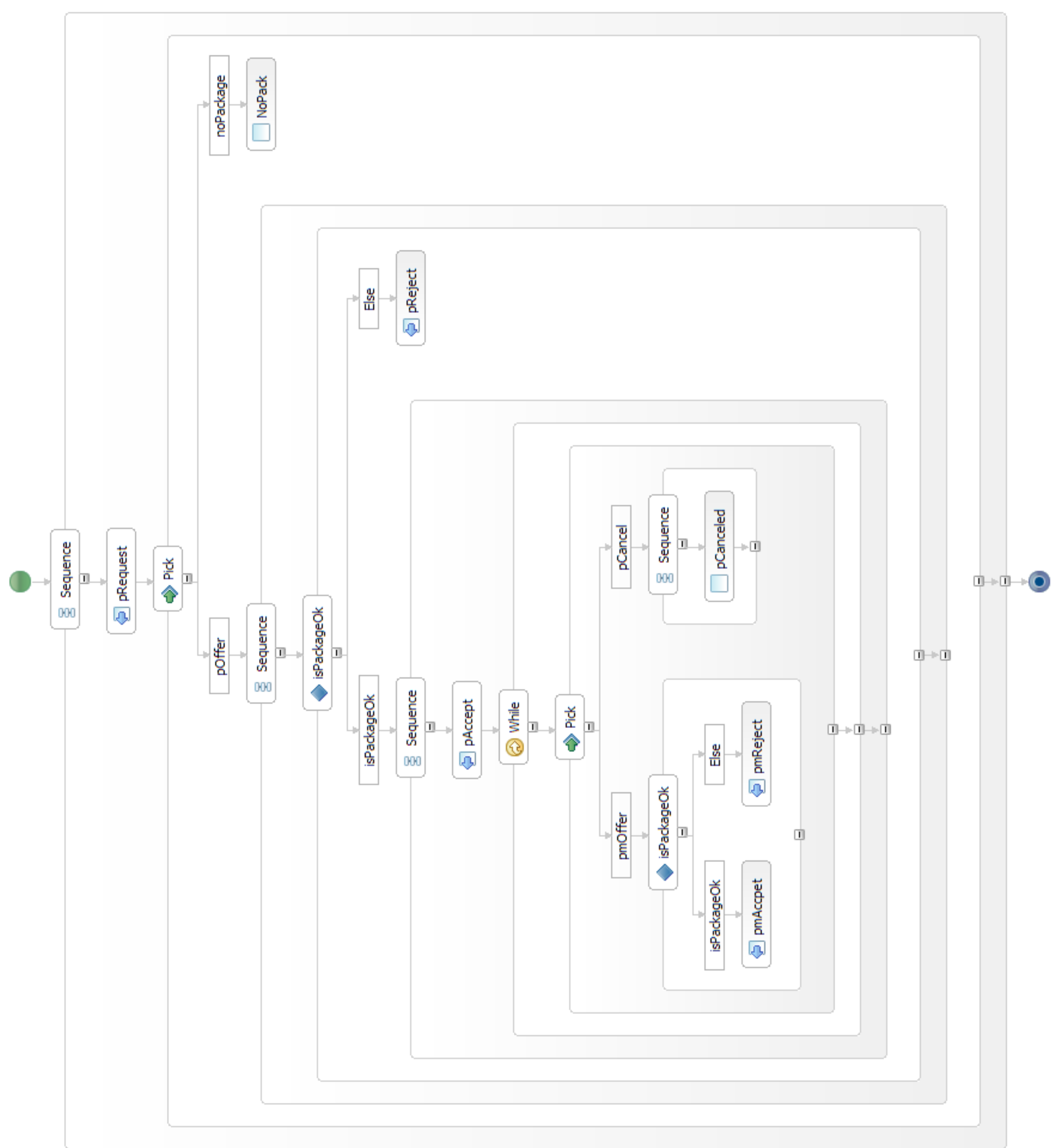


Figure 6.5: User Protocol

Second, the composition goal cannot be expressed in terms of a single protocol state to be reached. While requirement (1) can be encoded as state reachability, requirements (2) and (3) are reactive rules that express desirable reaction to critical events. Moreover, requirements (1) and (2) include prioritized alternatives (e.g., to modify a hotel, if not possible - to cancel the whole package). Another important aspect is that these complex control-flow requirements go along with data-flow requirements that also have to be encoded and taken into account by the composition.

Finally, the selected application domain motivates the reuse of the composition requirements. Indeed, it is quite likely that at some point one provider of flight ticket services will be replaced with another one (e.g., in user-centric setting the situation may become even more complex if we want to let the user select the providers to be used within the composition). To avoid additional expenses on application support, the requirements have to be reusable.

6.2 Composition Model Overview

The overview of our composition model is represented in Fig. 6.6. Structurally, it is very close to the model for fragment composition discussed in Chapter 4. However, in addition to dealing with services rather than fragments, our service composition approach concerns some advanced composition topics that have not been covered in the approach for fragment composition. In particular, we consider the problems of rich control- and data-flow requirements and show how the context-aware service composition can be used in user-centric applications.

The explicit context model is the same as in fragment composition, and is modeled as a set of context properties. In the VTA scenario, context properties might be the Hotel Reservation, the Flight Ticket and the

Travel Package. Similarly to the model of fragment composition, services are related to context model through service annotations and composition requirements are expressed over the context model rather than over service states. The core idea is that while service execution is closely related to the changes in status and data of context properties, the modeling of the latter does not depend on a particular service implementation. As such, by expressing control- and data-flow composition requirements at the level of context properties on the one side, and by relating services to context properties on the other side, we create a composition framework in which composition requirements, though detached from service implementations, can always be automatically grounded on them.

Context property behaviour is captured by its state diagram, which defines all possible property states and transitions between them. In fact, the transitions correspond to activities that can be performed over the context property (e.g., the flight ticket can be booked or cancelled) and to the external events affecting it (e.g., flight delay). Context property data

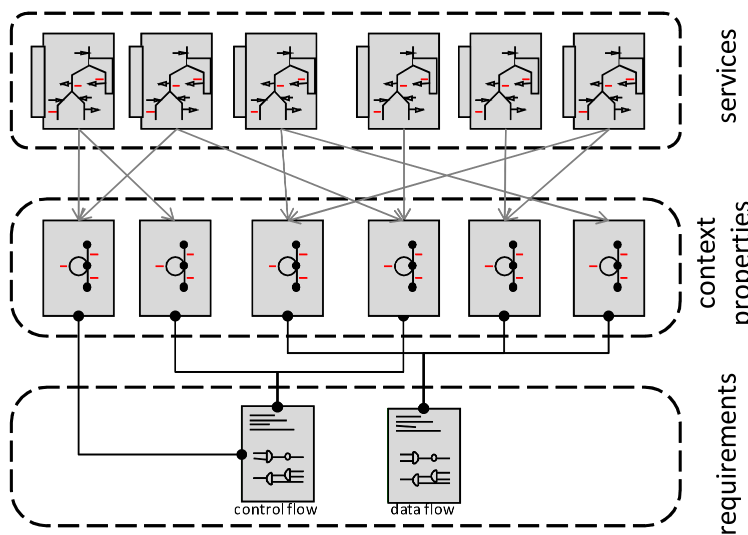


Figure 6.6: Service composition model

is a collection of data fields similar to parts in WSDL messages.

To link context properties and services described in WSDL and Abstract BPEL (or, potentially, in any other similar language), we annotate service descriptions with context-related information. In this way we implicitly define mapping between the execution of service operations and the changes in context property status and data. From the control flow perspective, every service operation may be annotated with 1) *contextual effect* indicating context property evolution enforced by operation execution and 2) *contextual precondition* indicating in which contextual conditions operation execution is allowed. Concerning the data flow, it is explicitly specified how the parts of input and output messages relates to the data fields of context properties (similarly to the DataNet approach of [79]).

The aforementioned service annotations are usually quite intuitive since they reflect the functional properties of services from the perspective of the application domain. It is worth to notice that in this way it becomes easy to modify a scenario to account for different service implementations: it is enough that services are properly annotated, while it is not necessary to change context property models nor requirements on their control- and data-flow.

In order to define requirements on conceptual level, we define them at the level of context properties rather than service specifications. Namely, in control-flow requirements we express tasks in terms of reachability of states in context property state diagrams (e.g., to reach state "created" in the Flight Ticket) or in terms of context property transitions (events) to be triggered (e.g., to trigger transition related to the modification of Hhotel Reservation). We also allow for coordination requirements, where a task has to be performed as a reaction to some contextual situation (e.g., to cancel hotel reservation in case of flight cancellation). Within a task, recovery goals can also be specified (e.g., to modified a hotel reservation, if

not possible to cancel the whole package). Conceptual data-flow requirements show how data fields of one context property relate to data fields of another context property (e.g., flight arrival date is equal to hotel check-in date). This information is later used to direct data flows within the composition. The extension of DataNet approach of [79] is used here.

6.3 Context Model

We completely reuse the context model and all the respective definitions presented in Chapter 4. Here we simply give some examples related to the motivating case study of this section.

Example 5 (Context Properties)**.** In our motivating example we distinguish three context properties: Flight Ticket, Hotel Reservation and Travel Package. Their respective state diagrams are show in Fig. 6.7. Flight Ticket and Hotel Reservation model real entities that the composition operates while Travel Package models the virtual concept of predefined travel solution as it is perceived by the user.

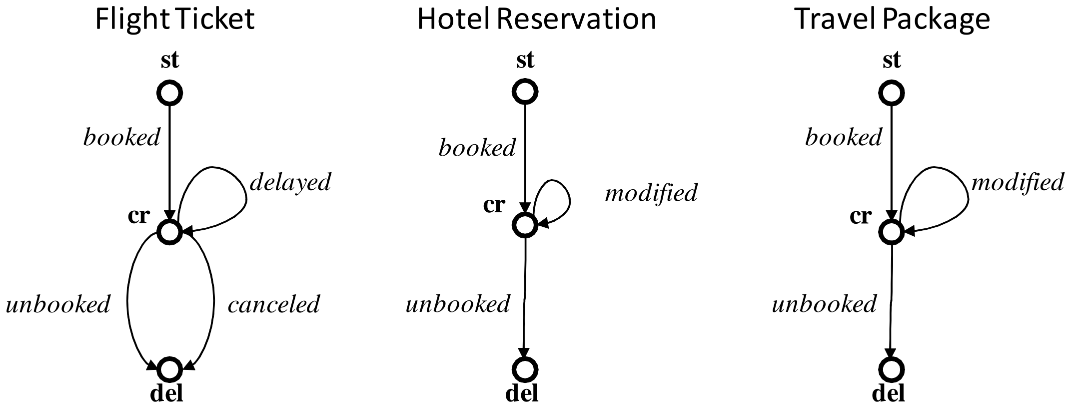


Figure 6.7: Context properties in virtual travel agency scenario

Let us consider one of the properties in detail. Flight Ticket contains three states. State *st* is the initial state where the flight ticket is not yet

instantiated, state *cr* corresponds to the situation where the ticket is already booked and state *del* is where the ticket is not valid anymore. The transitions model all possible activities and events related to the ticket. For example, transition labeled with event *created* corresponds to ticket booking, events *delayed* and *canceled* correspond to flight delay and cancellation. It can be seen that there are two transitions leading from state *cr* to state *del*. Although they correspond to the same changes to the state, *unbooked* corresponds to intentional cancellation of the ticket by the customer while the *canceled* corresponds to its cancellation by the airline. We model them as two different events in order to be able to distinguish them in requirements (e.g., if we want to react only to the flight cancellation by the airline while ignoring the same action performed by the customer).

In order to be able to resolve state and event of different objects with the same name we agree to denote event ev of property p as $ev^e(p)$ and state st of property p as $st^s(p)$. In the future we use the names of $flight$, $hotel$ and $package$ to denote the context properties of Flight Ticket, Hotel Reservation and Travel Package respectively.

Example 6 (Context Formula)**.** In our service composition model we intensively use context formulas, both in service annotations and in goal specifications. For example, a context formula expressing the situation in which both parts of a travel package (i.e., flight ticket and hotel reservation) are instantiated can be expressed as formula $cr^s(flight) \wedge cr^s(hotel)$, and can be used as a goal reflecting the need to book both parts of the package.

6.4 Services and Service Orchestration

6.4.1 Service Model

We assume that services are stateful and their description includes both service interface and protocol. Following [18], we initially model service protocol with a state transition system with three types of actions. *Input* actions model input messages received by a service, *output* actions model output messages sent by a service, and *internal* actions(also called τ-actions) model internal operations such as auxiliary control-flow transitions and data-related manipulations that are both hidden from the outside world. Output and internal actions can be considered as controllable since the service can decide when and which of them to execute. Similarly, input actions are not controllable since the decision on their execution is made by a service client. This distinction is important to produce correct composition of asynchronous services. Formally:

Definition 33 (Service)**.** *Service is a tuple* $s = \langle V, v^0, I, O, R \rangle$*, where*

- V *is the set of states and* $v^0 \in V$ *is the initial state;*
- I *is a set of input actions,* O *is a set of output actions such that* $I \cap O = \emptyset$*;*
- $R \subseteq V \times (I \cup O \cup \{\tau\}) \times V$ *is a transition relation with three types of actions.*

6.4.2 WS-BPEL Services as STS

Translation of Abstract BPEL specifications to STS

In order to translate Abstract BPEL specifications to state transition systems we use the translation rules similar to those described in [18, 76]. We remark that our translation is restricted to the most common constructions

of the WS-BPEL language, namely basic activities invoke, receive, receive, assign, empty and structured activities sequence, switch, while, pick (without timeouts). However, we believe that the considered subset provides enough expressiveness to be enough in the most of application domains.

As we already mentioned, we do not focus on the data flow in our approach, which significantly simplifies the transformation compared to those used in [18, 76] (we actually ignore all data-related information, including conditions in such constructs as switch and while). The key transformation principles are quite intuitive. The basic observable actions in Abstract BPEL specifications (i.e., those related to message exchange) appear as input/output transitions in a service STS. Similarly, unobservable actions become τ-transitions in a service STS. The structure activities imply the recursive transformation of its constituent elements and the transformation of the whole process consists in recursive transforming its individual constructs and rejoining them into a single STS.

The visual rules for the translation of basic activities are given in Table 6.1. The observable actions are represented with receive, reply and invoke, altogether covering both synchronous and asynchronous interaction. The reply and receive activity are used for asynchronous interactions in both directions. In order to keep the sets of input and output actions in an STS disjoint, reply and receive for the same operation use different action names. The invoke activity can be of two types (as defined is WS-BPEL): one-way operation invocation and synchronous operation invocation. request-response invocation. The two basic activities assign (of all types) and empty are transformed into internal activities since they are not observable to a service partner and are a part of internal service logic.

The mapping of the structured activities can be found in Table 6.2. The sequence activity is a linearly ordered list of activities. The switch activity is used to model internal conditional branching in the protocol that is hidden

WS-BPEL activity	State Transition System
`receive` `operation="op"`	?op
`reply` `operation="op"`	!rep_op

`invoke`
`operation="op"`
`inputVariable="x"`
	!op

`invoke`
`operation="op"`
`inputVariable="x1"`
`outputVariable="x2"`
	!op
	?rep_op

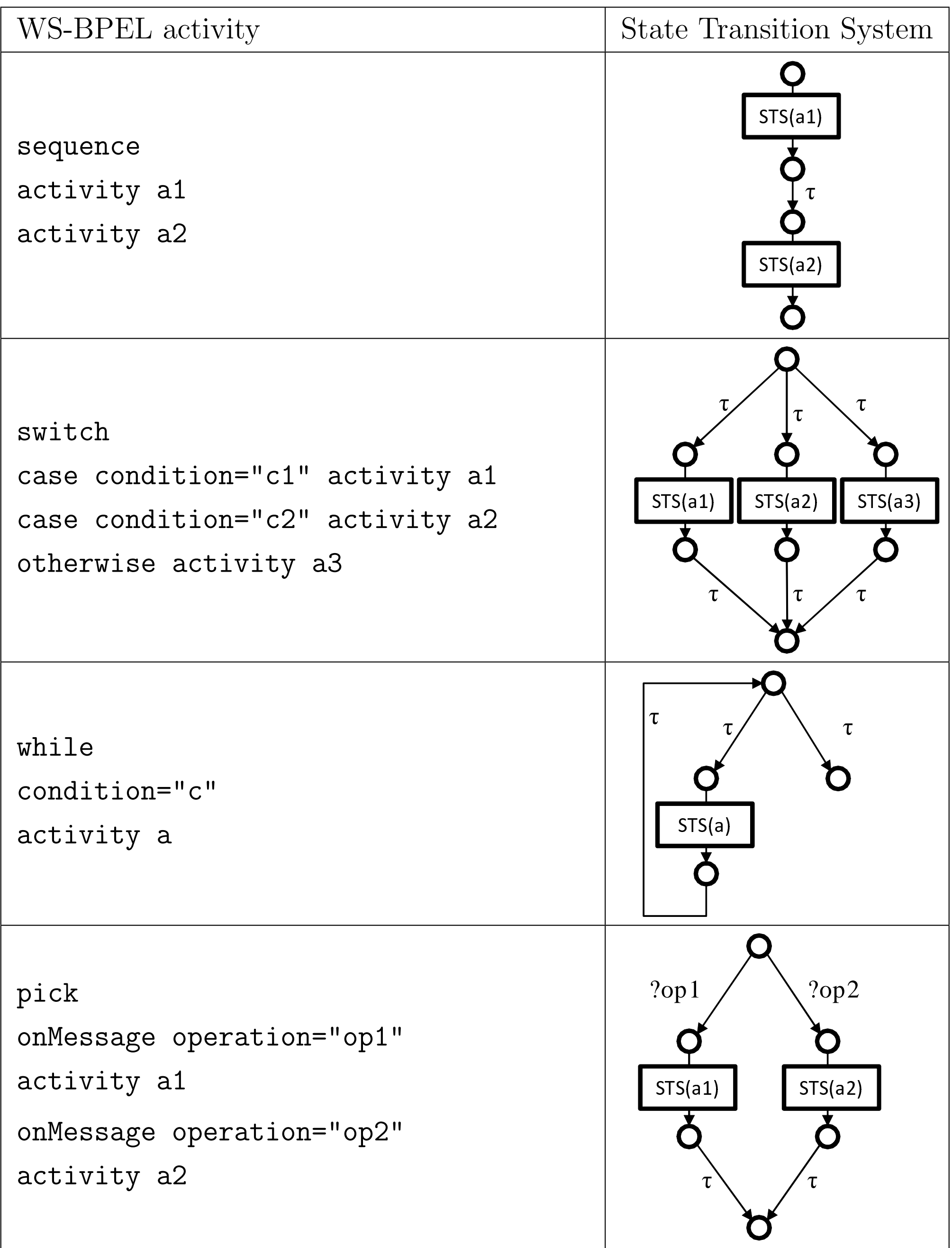
`assign`
`copy from variable="x1" part="p1"`
`to variable="x2" part="p2"`
	τ

`empty`	τ

Table 6.1: Translation of basic WS-BPEL activities into STSs

from the outside world. The while activity supports loops. Its behaviour is similar to that of switch. The pick activity models another type of branching that is controlled by the client by means of input messages to a service.

Example 7. As an example, in Fig. 6.8 we give the result of the transformation of the Flight Booking Service (Fig. 6.1).



WS-BPEL activity	State Transition System

`sequence`
`activity a1`
`activity a2`
	STS(a1) τ STS(a2)

`switch`
`case condition="c1" activity a1`
`case condition="c2" activity a2`
`otherwise activity a3`
	τ τ τ STS(a1) STS(a2) STS(a3) τ τ τ

`while`
`condition="c"`
`activity a`
	τ τ τ STS(a)

`pick`
`onMessage operation="op1"`
`activity a1`
`onMessage operation="op2"`
`activity a2`
	?op1 ?op2 STS(a1) STS(a2) τ τ

Table 6.2: Translation of structured WS-BPEL activities into STSs

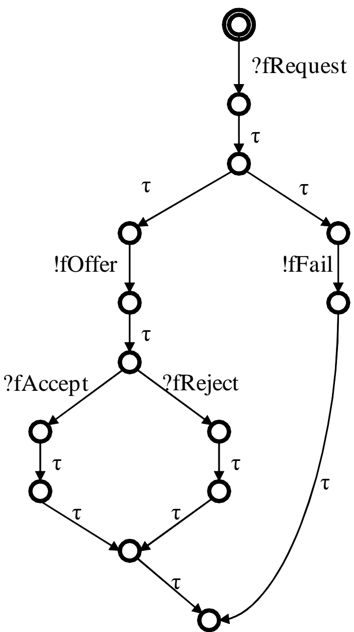


Figure 6.8: Flight Booking Service as STS

Translation of STS into executable WS-BPEL

In our formal model, the composition is obtained as a state transition system and has to be translated back to an executable process specification. This imposes additional restrictions on the model of such STS compared to that of a service. In particular, a solution STS cannot contain multiple controllable actions (in case of orchestrator, they are τ-actions and output actions) starting from the same state. Indeed, in such situation, the engine would not be able to figure out which of them has to be executed next. Such conflicting situations are referred to in the literature as "internal" nondeterminism. At the same time, having a number of uncontrollable (input) actions from one state is quite natural since it reflects the nondeterministic behaviour of external service, where all cases have to be taken into account. Such situation is easily processed by a process engine (e.g., "pick" activity in BPEL waits for a number of possible inputs).

These second type of situations is also known as "external" nondeterminism. Summing it up, "external" nondeterminism is fine and "internal" nondeterminism is not fine in an STS that encodes executable processes. We also require that runnable STS is deterministic. To define it formally, we introduce runnable STS as follows:

Definition 34 (Runnable STS (Process))**.** *A runnable STS is a deterministic STS $\langle V, v^0, I, O, R \rangle$, such that:*

- *if $(v, a, v') \in R$ and $a \in O$, then no other transition from v belongs to R;*
- *if $(v, \tau, v') \in R$ then no other transition from v belongs to R.*

The back translation of a runnable STS is quite straightforward. We simply revisit all states of the tree and convert the respective transitions into BPEL activities using the information in original specifications of components and mapping between action labels and names in the specifications that can be created during the procedure of direct translation. In particular, all output transitions become *invoke* and *reply* activities (depending on whether the corresponding operations are synchronous or asynchronous in a component protocol). Similarly, input transitions become either *receive*, or *pick* (depending on whether it is only one input transitions from this state or they are many).

6.4.3 Observable Behaviour

In order to discuss observable behaviour of services, we introduce a concept of τ-closure(v) in service s as a set of states that are reachable from v through τ-transitions:

Definition 35 (τ-closure)**.** *Let $s = \langle V, v^0, I, O, R \rangle$ be a service and $v \in V$ be one of its states. A τ-closure of v is a set of states that are reachable*

from v through τ-transitions, i. e., τ-closure(v)= $\{v_x : \exists v_0, v_1, \ldots, v_n : v_0 = v, v_x = v_n, \forall i \in [0, n-1] : (v_i, \tau, v_{i+1}) \in R\}$.

In our formal model we make an assumption that all the internal logic of services is hidden from external partners (which is consistent with basic principles of SOA), and the partners can perceive the internal state of a service only via its observable behaviour (i.e., input and output messages). So our approach aims to perform service composition on the level of observable behaviours rather than on the level of internal protocols. As a result, all the τ-actions within a protocol become of no use to us since we cannot track their execution. Considering Def. 33, we can conclude that the observable behaviour of a service is an STS such that at each step of its evolution we can precisely understand what are the next observable operations to be performed. In fact, it must be an STS with only input and output actions.

Definition 36 (Observable STS (Observable Service Behaviour))**.** *Observable STS is a tuple $s = \langle V, v^0, I, O, R\rangle$, where*

- *V is the set of states and $v^0 \in V$ is the initial state;*
- *I is a set of input actions, O is a set of output actions such that $I \cap O = \emptyset$;*
- *$R \subseteq V \times (I \cup O) \times V$ is a transition relation with no τ-actions.*

The problem here is how service protocol as defined in Def. 33 correlates with service observable behaviour.

The fact of partial observability is a problem for service orchestration, because when a service transits via τ-actions, an external partner (e.g., an orchestrator) cannot figure out the moment when a service is ready to execute next observable action (i.e., receive or send messages). To fix service behaviour in such situations, we assume that service s is ready

for observable action a in state v if there exists a state v' accessible from v via τ-transitions (i.e., $v' \in \tau$-closure(v)) such that a transition labeled with a is available from v'. In other words, if action a is applicable on a state belonging to τ-closure(v), the external partner, being not able to distinguish between states within the same τ-closure, should treat it as if a was applicable on v. We can define the procedure for deriving an observable service behaviour from its protocol as follows:

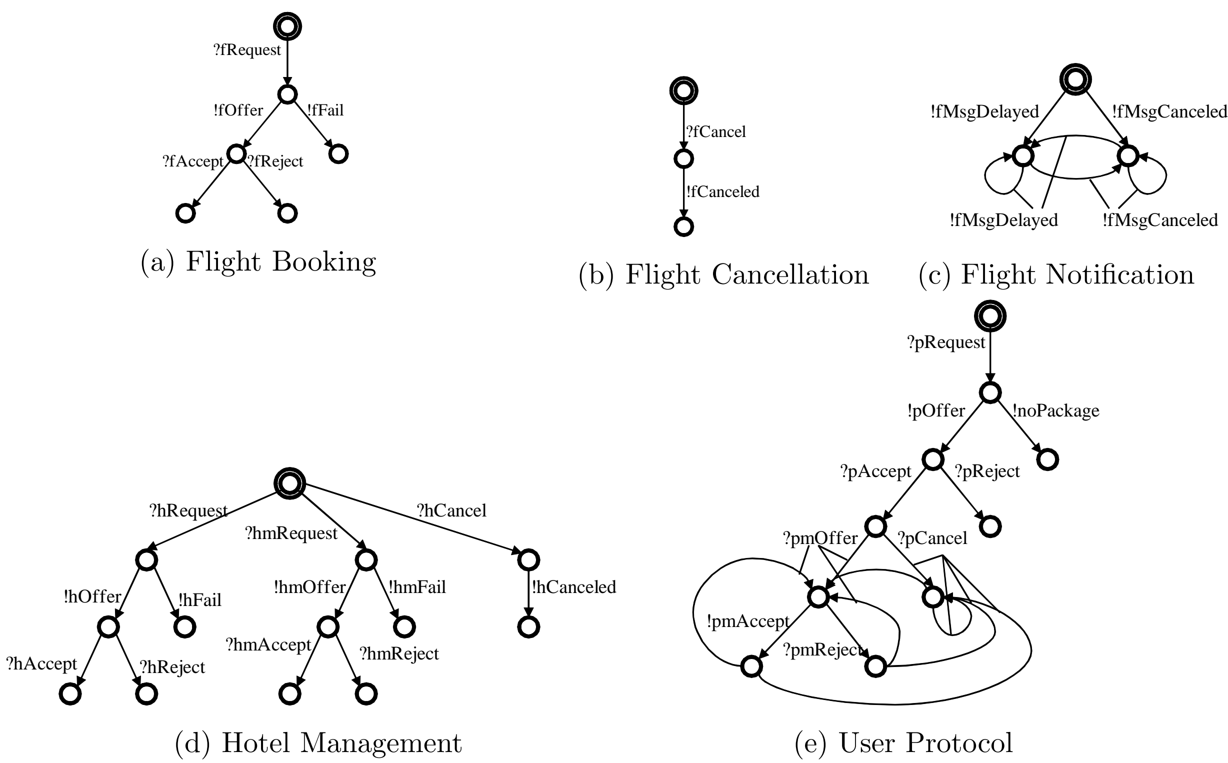
Definition 37 (Service Protocol as Observable Behaviour)**.**
Let $s = \langle V, v^0, I, O, R\rangle$ be a service. The observable behaviour of s is an STS $s' = \langle V', V'^0, I, O, R'\rangle$ such that:

- $V' = \{v' \in V : (\exists(v, a, v') \in R : a \neq \tau) \vee (v' = v^0) \vee (v' \in Finals(s)\}$;
- *for every transition $(v, a, v') \in R : a \neq \tau$ and for every state $v_x \in V'$: $v \in \tau$-closure(v_x), we define a transition $(v_x, a, v') \in R'$;*
- *no other states and transitions belong to s'.*

Keeping it simple, the observable behaviour is obtained by eliminating all τ-transitions and connecting non-τ transitions directly rather then via a chain of τ-transitions. We remark that in such way the observable behaviour can be directly derived from service protocol specification.

Example 8. The observable behaviour of the services from the Virtual Travel Agency scenario are shown in Fig. 6.9. Although the parts of the protocols that correspond to WHILE-loops may look complex, they still correctly reflect the observable behaviour of a service in this way.

Undecidable behaviour. We remark that such process languages as Abstract BPEL allow for service protocols that have undecidable observable behaviour. For example, if a service protocol contains a SWITCH activity with two branches starting with RECEIVE and REPLY activities



(a) Flight Booking (b) Flight Cancellation (c) Flight Notification

(d) Hotel Management (e) User Protocol

Figure 6.9: Observable behavior of services from virtual travel agency scenario

respectively, an external partner that cannot evaluate the conditional expression of the SWITCH block, cannot decide whether to send a message to a service or to receive a message from it. In our formalism, services with undecidable behavior can be defined as follows:

Definition 38 (Service With Undecidable Observable Behaviour)**.** *Service $s = \langle V, v^0, I, O, R\rangle$ has undecidable observable behaviour if there exists state $v \in V$ such that:*

1. $(\exists (v_1, a_1, v_1'), (v_2, a_2, v_2') \in R : v_1, v_2 \in \tau\text{-}closure(v)) \wedge (a_1 \in I) \wedge (a_2 \in O)$;
2. $(\exists (v_1, a_1, v_1'), (v_2, a_2, v_2') \in R : v_1, v_2 \in \tau\text{-}closure(v)) \wedge (a_1, a_2 \in I) \wedge (v_1 \neq v_2)$;
3. $(\exists (v_1, a_1, v_1'), (v_2, a_1, v_2') \in R : v_1, v_2 \in \tau\text{-}closure(v))$.

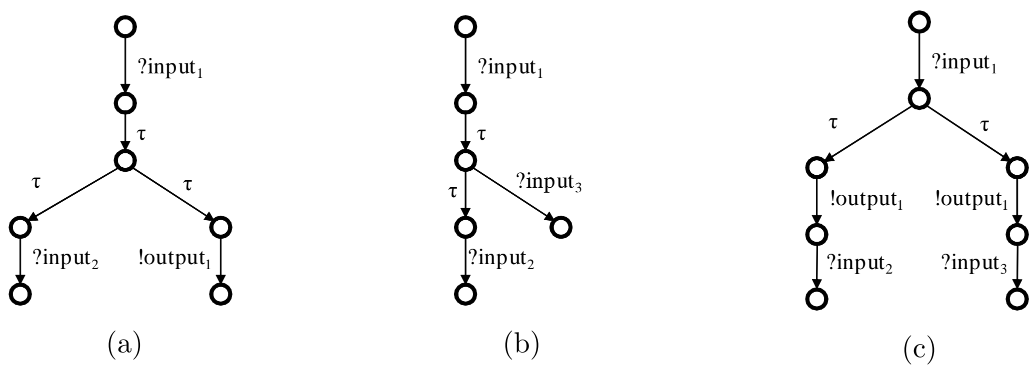


Figure 6.10: Services with undecidable observable behaviour

Example 9. In Fig. 6.10 the three cases of undecidable behaviour defined in Def. 38 are exemplified. Case 1 (Fig.6.10a) describes a situation where a partner cannot know from observable behaviour if to send a message ($?input_2$) or to expect a message from a service ($!output_1$). Case 2 (Fig.6.10b) is where a service may wait for different input message ($?inpit_2$ or $?input_3$) but not for all of them altogether. So, the partner cannot know which message must be sent. Case 3 (Fig.6.10c) describes behaviours with nondeterministic terminal state (after observable executions $?input_1$, $!output_1$ the service behaviour is unpredictable).

In other words, a service with decidable behaviour at each step either sends a known set of output messages, or expects a known set of input messages. Moreover, the observable action has to unambiguously indicate the next state of the observable behaviour. In our model, we assume that all the services we work with have decidable behaviours. Elimination of Case 3 also guarantees that an observable behaviour in our system is always a deterministic STS.

6.4.4 Services as Fragments

While comparing service protocols and service-based fragments of Chapter 4, it can actually be concluded that they have similar nature. The

major difference between them is that service protocols reflect the interaction between the two partners from the perspective of a service, while fragments reflect the same aspect but from the perspective of a client (e.g., orchestrator).

In general, fragments are more flexible since they let the designer to specify not only the interaction model but also some internal operations that have to be performed "inside" the client (e.g., this internal operations are CONCRETE activities in APFL). Still, the fragments can be naturally used by service providers to express how the respective functionality of their services has to be consumed by clients. As such, fragments can be considered as a competitor to abstract service protocols described with languages such as Abstract BPEL.

Considering observable behaviour of a service (Def. 36), it can be concluded that the observable behaviour of a client compatible with this service is essentially dictated by the observable behaviour of a service. By the compatibility in this case we understand the situations, where all interactions between the two parties are synchronized, i.e., whenever one partner is supposed to send a message, another partner has to be ready to accept it. It can be easily shown that such behaviour of a partner can actually be derived from an observable protocol by inverting input and output actions in observable protocol. In fact, in such way we obtain a process fragment reflecting the correct behaviour of an orchestrator when communicating to the corresponding service. Intuitively, if at some point a service is going to send one or a couple (in case of external nondeterminism) of output messages, the client has to be ready to accept any of them. Similarly, when a service is ready to receive one of a set of messages, one of these messages can be sent by the client. This semantics directly corresponds to the semantics of process fragments defined in Def. 7. As such, the following correspondence of an observable STS and a fragment can be installed:

Definition 39 (Service as Fragment). *Let $srv = \langle V, v^0, I, O, R\rangle$ be an observable behavior of some service. A fragment (as in Def. 7) corresponding to srv is an STS $f = \langle \mathcal{S}, s^0, \mathcal{I}, \mathcal{O}, \mathcal{R}\rangle$ such that:*

- *$\mathcal{S} = V$ and $s^0 = v^0$;*
- *$\mathcal{I} = O$ and $\mathcal{O} = I$;*
- *$\mathcal{R} = R$.*

A fragment corresponding to some service is called its *complementary* fragment.

Considering a set of services to be orchestrated we can eventually conclude that the problem of service composition consists in properly composing the fragments corresponding to the observable behaviour of such services. More precisely, having a set of services $s_1, \ldots, s_n$, we obtain their observable behaviours $s'_1, \ldots, s'_n$ and finally obtain their complementary fragments $f_1, \ldots, f_n$. The further composition theory generally coincides with the theory of Chapter 4 on context-aware fragment composition with minor differences that will be discussed later on. In several words, through asynchronous product we obtain an execution domain as in Def. 17 that encodes all possible correct executions of fragments. Since fragments are completely synchronized with the observable behaviour of services, the paths in the execution domain can also be interpreted as a parallel evolution of services $s_1, \ldots, s_n$. Since in our model the operational semantics of fragments and services coincides, the asynchronisity of services is essentially determined by asynchronisity of respective fragments. Consequently, the notion of consistent orchestrator for services and the way we derive it generally coincides with those of Chapter 4.

Belief-level system. In our approach to service composition we state that services are completely described by their observable behaviour and, as such, the problem of service composition can be eventually reduced to the

problem of fragment composition. This idea is based on the assumptions that 1) internal operation of services can never be observed by external clients and that 2) all services have decidable observable behaviour as defined by Def. 38. Although we think that these assumptions are reasonable, we remark that [18] proposes a more profound approach to treating partial observability and asynchronisity of interaction that does not require the assumptions above. This approach completely compatible with our framework and, if needed, can be integrated into it. In this approach complete service protocols $s_1, \ldots, s_n$ are directly combined by means of asynchronous product into execution domain Σ that contains τ-transitions reflecting internal evolutions of services. To make use of effective planning techniques for fully observable domains, the partial observability of services, modeled by τ-actions, is compiled away from Σ by building a so-called *belief-level system* Σ_B, an STS whose states correspond to *beliefs* of Σ - that is, to sets of states of Σ which are equally plausible for an external observer sensing the STS's inputs and outputs. Σ_B can then be encoded into a planning domain D, and, under mild assumptions, one can prove that the composition problem is solved by identifying (the STS corresponding to) a plan π that satisfies the composition goal ρ for the planning domain D. Although we have never performed this kind of analysis, we expect that a belief-level system and an execution domain in our approach for the same sets of services are likely to be very close to each other and in many cases coincide. This is a consequence of the similarity of the ways both approaches collapse τ-closures into single states.

6.5 Service Annotations

In order to link services to context we allow for service annotations which can be of two types: action *preconditions* and action *effects*. Semantically,

they are very close to those used in fragment annotation. Action precondition indicates in which context states an action can be executed. The set of allowed states is specified through a context formula. Action effect contains a set of context events that are triggered by action execution. The formal definition is as follows

Definition 40 (Service Annotation)**.** *Let $C = \{p_1, p_2, \ldots, p_n\}$ be a context with a set of context states L_C and a set of context events E_C and let $s = \langle V, v^0, I, O, R\rangle$ be a service. A service annotation of the service s over the context C is a tuple $\omega = \langle \mathcal{P}, \mathcal{E}\rangle$, where:*

- *$\mathcal{P} : (I \cup O) \rightarrow R_C$ is a precondition function that connect observable service action (inputs and outputs) to a context formula describing a set of allowed states;*
- *$\mathcal{E} : (I \cup O) \rightarrow E^*$ is an effect function that relates observable service actions to context events that they trigger. To avoid nondeterministic context runs we require that for any context property $p = \langle L, l^0, E, T\rangle \in C$ the following holds: $\nexists e_1, e_2 \in \mathcal{E}(a) : e_1, e_2 \in E$.*

Example 10. Annotations of some service actions in Virtual Travel Agency scenario are shown in Table 6.3. For example, action *fAccept* (Fig. 6.9a) corresponds to the positive accomplishment of the flight booking process. In the precondition we require that the flight ticket is not yet booked ($st^s(flight)$) and that the user has already approved the package booking ($cr^s(package)$) since this operation is not revertible. The effect of the action suggests that after the action is executed, the Flight Ticket property will proceed through a transition labelled with event $booked^e(flight)$, which corresponds to ticket booking.

The definition of action executability coincides with Def. 12 and the definition of action impact coincides with Def. 13 and Def. 16 (we do not

Table 6.3: Service annotations from virtual travel agency scenario

a	$\mathcal{P}$**(a)**	$\mathcal{E}$**(a)**
$fRequest$	$st^s(flight)$	$\emptyset$
$fAccept$	$st^s(flight) \wedge cr^s(package)$	$booked^e(flight)$
$fCanceled$	$cr^s(flight)$	$unbooked^e(flight)$
$fMsgDelayed$	$\top$	$delayed^e(flight)$
$fMsgCancelled$	$\top$	$canceled^e(flight)$
$pRequest$	$st^s(package)$	$\emptyset$
$pAccept$	$st^s(package)$	$booked^e(package)$
$pmAccept$	$cr^s(package)$	$modified^e(package)$
$pCancel$	$cr^s(package)$	$canceled^e(package)$

use impact for goals since neither goals not abstract activities are used in service specifications and their complementary fragments).

6.6 Problem of Context-Aware Service Composition

Due to similarity between annotations of services and fragments, the service annotations can be easily converted into the annotations of the complementary fragments. Taking into account that, similarly to annotated fragments, services and their annotations form annotated services, a system of annotated services

$$S^+ = \{\langle s_1, \omega_1 \rangle, \dots, \langle s_n, \omega_n \rangle\}$$

can be converted into a system of complementary annotated fragments

$$F^+ = \{\langle f_1, \omega_1 \rangle, \langle f_2, \omega_2 \rangle, \dots, \langle f_n, \omega_n \rangle\}$$

that, when accompanied by context model C becomes a context-aware system of fragments $\Psi = \langle F^+, C \rangle$.

Since the elementary goal is expressed as a context formula ρ encompassing the goal states to be reached, the problem of context-aware service composition can be formulated as follows:

Definition 41 (Context-Aware Service Composition Problem). *Let S^+ be a set of services annotated over context C and let $\rho \in R_C$ be a context formula over C expressing composition requirements. The problem of context-aware service composition for S^+, C, ρ corresponds to a problem of fragment composition for $\Psi = \langle F^+, C\rangle$ and ρ (see Def. 23), where F^+ is a set of annotated fragments complementary to services S^+.*

The procedure of resolving the composition problem is practically the same as described in Chapter 4. The only difference is that the first step of translation of services into an execution domain consists in deriving the observable behaviours of services and converting them into complimentary fragments. The remaining steps are the same as for the context-aware composition of fragments.

6.7 Discussion

Coming back to the motivating example of Virtual Travel Agency and considering the context properties in Fig. 6.7 it can be easily observed that the composition requirements expressed in terms of context formulas are not enough to cover the three requirement expressions given in Section 6.1. Here we can identify a number of problems that have to be addressed.

The first requirement in the list expresses a need of transactional booking of all of the parts of the package from the initial situation. This requirement can be expressed in our formalism through the following formula:

$$(cr^s(flight) \wedge cr^s(hotel) \wedge cr^s(package)) \vee \\ (st^s(flight) \wedge st^s(hotel) \wedge st^s(package))$$

The formula indicates that either all respective objects are booked or none of them is booked. However, the strong planning for such goal with initial context state $(st^s(flight) \wedge st^s(hotel) \wedge st^s(package))$ will eventually return

an empty plan (the plan will be found but it will contain no actions). The reason is that this requirement does not distinguish the preferences between two goal states. Although state $(st^s(flight) \wedge st^s(hotel) \wedge st^s(package)$ is a goal state (and this is why the plan is empty: in the initial state we are already in the goal state and so the shortest way to satisfy the goal is to do nothing) it is not indeed what we would expect to get as a result of composition. Our primary goal is $(cr^s(flight) \wedge cr^s(hotel) \wedge cr^s(package))$ and only if it is not possible to reach it, we would like to guarantee that we at least stay with none of the objects booked. So the expected behaviour is where the composition does its best to book the package but guarantees that once it is not possible, nothing is booked. We can conclude that preferences in the requirements language are needed.

The second and the third statement of the requirements are actually reactive requirements. They specify some task that has to be performed exclusively as a *reaction* to some triggering situation. In general such expressions cannot be expressed as reachability goals. Moreover, the second requirement also features two alternative reactions that have to be given different preference (once a flight is delayed, the hotel and the package have to be aligned with them, and only if it is not possible the cancellation of all three has to be performed). From this, we conclude that reactive requirements are needed to target our motivating example.

Another important aspect is that sometimes context states are not enough to express the tasks to be performed. Indeed, in the second expression it is said that once the flight has been delayed, we have to modify both the hotel and the package. Going back to context properties in Fig. 6.7, it can be observed that once the initial booking is performed the context remains in state $(cr^s(flight) \wedge cr^s(hotel) \wedge cr^s(package))$. When the flight is delayed the state of the context is still the same but what is more important is that the state that will be reached as a result of modification

of hotel and package will also be the same. And so the current state and the goal state in this case coincide and the plan that will be returned in case of requirements expressed as a rechability goal will be the empty one. The solution here is to enable composition requirements that combine both states to be reached and events to be triggered (so-called procedural goals).

Finally, what is more important is that we would like to combine all requirements into one composition so that not only does it perform certain tasks but also provides maintenance of consistency by proper reaction to certain situations. The intuition of such approach in the VTA scenario, is that the composition first transactionally books the package and then supports its consistency by properly reacting to flight delays and cancellations.

The problems outlined in this discussion are addressed in the next chapter.

Chapter 7

Advanced Topics in Service Composition

Although the composition requirements expressed in terms of context states to be reached were enough for the tasks of process adaptation considered in Chapter 3, in the concluding discussion of Chapter 6 we showed that very often the expectation from the composition cannot be expressed as a simple reachability of context states and may need advanced requirement languages and resolution techniques. In the first section of this chapter we propose our context-based language for control-flow requirements that aims to overcome the limitations of composition goals exploited in Chapter 6.

Although the main focus of these dissertation is control-flow requirements, we realize the importance of data-flow requirements and devote the second section of this chapter to them. We overview one of the approaches to specifying data-flow requirements for service composition that was originally developed to be used in the planning-based service composition systems. After that, we propose a prototype solution for adopting this technique to be exploited in our context-based composition framework.

In this chapter we work with annotated fragments assuming that they are complementary fragments for some system of annotated services S^+.

7.1 Control-Flow Requirements

Similarly to context goals, our language for control-flow requirements is context-based, i. e., it expresses requirements on top of our context model. As we showed in the previous chapters, this increases requirements reusability and makes them more robust against certain types of dynamic changes.

At the same time, in our language we try to address the challenges that were discussed in the conclusion of Chapter 6 and that can be summarized as follows:

- the use of context events along with context formulas;
- the possibility of tasks with preferences;
- the possibility to express reactive tasks.

7.1.1 Language and Semantics

In the language syntax we reuse elements of our context model, in particular context events and context formulas. Conceptually, there are two types of requirements: *imperative* and *reaction*. An imperative contains a *task* to be unconditionally performed by the composition. A reaction consists of a context event (*trigger*) and a task. It requires that every time the trigger occurs in the context, the associated task is performed. A task consists of a few prioritized *clauses*. A task is considered to be performed if at least one of its clauses (preferably, the one with higher preference) is satisfied. Clauses are propositional formulas over context events and context formulas. A clause containing a context formulas is satisfied when the context transits via a context state satisfying the formula. A clause containing a context event is satisfied when this event occurs in the context. The syntax of the language is formally defined as follows:

Definition 42 (Control-Flow Requirements Language)**.** *Let C be a context. Composition requirements expr are specified as:*

$$\begin{array}{|l|}\hline expr := task \mid e \rightarrow task \mid expr \; ; \; expr \\ task := (clause, c) \mid task, task \\ clause := \rho \mid e \mid clause \vee clause \mid clause \wedge clause \\ \hline \end{array}$$

,where $e \in E_C$ is a context event, $\rho \in R_C$ is a context formula and $c \in \mathbb{N}$ is a task preference.

In the future with $Terms(clause)$ we will denote all terms (i.e., event and context formulas) appearing in clause *clause*. Similarly, with $Terms(task)$ we will denote all terms appearing in the clauses of *task*. For example, if $task = (clause, c)$ then $Terms(task) = Terms(clause)$ and if $task = (clause_1, c_1), \ldots, (clause_n, c_n)$ then $Terms(task) = Terms(clause_1) \cup \ldots \cup Terms(clause_n)$.

Composition requirements of form $expr = task$ or $expr = e \rightarrow task$ are called *elementary expressions* while requirements of form $expr = expr_1; expr_2; \ldots; expr_n$, where $expr_i$ is an elementary expression for all $i \in [1, n]$, are called *composite expressions.*

Example 11. Using the new language, the control-flow composition requirements from the Virtual Travel Agency scenario discussed in Section 6.1 can be specified in our requirements language as follows:

1.

$$(cr^s(flight) \wedge cr^s(hotel) \wedge cr^s(package), 2), \\ (st^s(flight) \wedge st^s(hotel) \wedge st^s(package), 1)$$

The expression is an imperative containing two tasks with different preferences. It ensures that all the components of the package are purchased transactionally. The task with the higher preference requires that all context properties are in state *cr* which corresponds to

the situation where both hotel reservation and flight ticket are purchased and the package is approved by the user. The recovery goal requires that none of the elements are purchased upon user's disapproval (or if some item is not available). As such, only two situations satisfy the expression: when all of the elements are purchased (more preferable) or none of them is purchased (less preferable);

2.

$$\begin{aligned} & delayed^{e}(flight) \rightarrow \\ & \qquad (modified^{e}(package) \wedge modified^{e}(hotel), 2), \\ & \qquad (unbooked^{e}(package) \wedge unbooked^{e}(hotel) \wedge unbooked^{e}(flight), 1) \end{aligned}$$

The expression specifies the reaction to the flight delay. The more preferable reactive task is where a hotel reservation is modified in order to be aligned with the flight changes. The modification has to be agreed with the user (the package has to be modified as well, which corresponds to the user's approval of changes. The less preferable alternative suggests that the whole package with all its constituent is unbooked;

3.

$$\begin{aligned} & canceled^{e}(flight) \rightarrow \\ & \qquad (unbooked^{e}(package) \wedge unbooked^{e}(hotel), 1) \end{aligned}$$

The reaction to the flight cancellation consists in unbooking the hotel reservation and the package (i.e., to inform the user about the cancellation).

Control-flow Requirements Semantics

We define the semantics of the control-flow composition requirements through the runs of the context-aware execution domain that satisfy it.

Definition 43 (Clause Satisfaction). *Let $\Psi = \langle F^+, C \rangle$ be a context-aware system, L_C and E_C be sets of context states and context events of context C respectively and let Σ_{CF} be the context-aware execution domain for Ψ (see Def. 20). A run $\pi = ((s_1, l_1), a_1, (s_2, l_2), a_2, \ldots, a_{n-1}, (s_n, l_n))$ of Σ_{CF} satisfies requirement clause cl defined over C (denoted $\pi \models cl$) if and only if:*

- *$cl = \rho$ and $\exists (s, l) \in \pi : l \models \rho$;*
- *$cl = e$ and $\exists a \in \pi : e \in \mathcal{E}(a)$;*
- *$cl = cl_1 \vee cl_2$ and $\pi \models (cl_1, n)$ or $\pi \models (cl_2, n)$;*
- *$cl = cl_1 \wedge cl_2$ and $\pi \models (cl_1, n)$ and $\pi \models (cl_2, n)$.*

Conceptually, the idea is that a context-aware run of the execution domain satisfies some context formula if its associated context evolution passes through a state where this formula is satisfied. The run satisfies some event if this event is among the events triggered within the run. The extension of these semantics to formulas over events and context formulas is trivial.

While extending the notion of satisfaction to imperative and reaction expression in requirements, we have to take into account the preferences presented in them. For the sake of simplicity, we assume that all tasks within an expression are given different preferences that are natural numbers $1, 2, \ldots, m$ without gaps:

Definition 44 (Elementary Expression Satisfaction). *Let $\Psi = \langle F^+, C\rangle$ be a context-aware system, and let Σ_{CF} be the context-aware execution domain for Ψ.*

A run $\pi = ((s_1, l_1), a_1, (s_2, l_2), a_2, \ldots, a_{n-1}, (s_n, l_n))$ of Σ_{CF} satisfies imperative expression

$$expr = (cl_1, 1), (cl_2, 2), \ldots, (cl_m, m)$$

defined over C with preference $i : 1 \leq i \leq m$ (denoted $\pi \models^i expr$) if and only if $\pi \models cl_i$ and $\nexists j : i < j \leq m : \pi \models cl_j$.

In a similar situation π satisfies reaction expression

$$expr = e_t \rightarrow (cl_1, 1), (cl_2, 2), \ldots, (cl_m, m)$$

defined over C with preference m if $\not\exists a \in \pi : e \in \mathcal{E}(a)$ (i.e., requirement is never triggered). Otherwise π satisfies expr with preference $i : 1 \leq i \leq m$ if there exists a terminating subsequence $\pi^k = ((s_k, l_k), a_k, \ldots, a_{n-1}, (s_n, l_n)) \in \pi$ such that $\not\exists a \in \pi^k : e \in \mathcal{E}(a)$ and $\pi^k \models cl_i$ and $\nexists j : i < j \leq m : \pi^k \models cl_j$.

In other words, a run satisfies an imperative expression with preference i if it satisfies a clause of a task with assigned preference i and there exists no task with preference higher than i whose clause is also satisfied by this run. For a reaction expression the definition is the same but the satisfaction is considered since the last occurrence of trigger e_t.

Finally, there is an issue of how to define the satisfaction of a group of expressions. Indeed, while each requirement expression prioritizes tasks within itself, it is not clear how to prioritize the tasks belonging to different expressions. In this case we can say that tasks within the requirements are partially ordered. The problem of dealing with such requirements is very close to the problem of dealing with partially ordered goals in planning with preferences (e.g., see [110]). The problem of flattening partially

ordered goals/tasks is a complex problem that, however, goes beyond the scope of this thesis. In our approach we use the most simple flattening technique where in order to calculate the satisfaction preference for a group of expressions we simply sum up the satisfaction preferences of each expression. Formally:

Definition 45 (Requirements Satisfaction)**.** *Let $\Psi = \langle F^+, C\rangle$ be a context-aware system and let Σ_{CF} be the context-aware execution domain for Ψ.*

A run $\pi = ((s_1, l_1), a_1, (s_2, l_2), a_2, \ldots, a_{n-1}, (s_n, l_n))$ of Σ_{CF} satisfies requirements $expr = expr_1; expr_2; \ldots; expr_k$ defined over C with preference $x \in \mathbb{N}$ (denoted $\pi \models^x expr$) if and only if there exists a set of natural numbers $c_1, c_2, \ldots, c_k : c_i \in \mathbb{N}, i \in [1, k]$ such that $\pi \models^{c_i} expr_i, i \in [1, k]$ and $c_1 + c_2 + \ldots + c_k = x$.

Since in our approach the composition is supposed to provide continuous satisfaction of the requirements. It means that infinite runs are legal (for example, the flight can be delayed infinite number of times, and the execution may be infinite as well). That is why we have to extend the requirements satisfaction to infinite runs. We consider an infinite run as satisfying if from any its point it provides requirements satisfaction in finite number of step. The satisfaction preference of the infinite run is the lowest achieved in the infinite perspective.

Definition 46 (Requirements Satisfaction (Infinite Run))**.** *Let $\Psi = \langle F^+, C\rangle$ be a context-aware system and let Σ_{CF} be the execution domain for Ψ. Let $\pi = ((s_1, l_1), a_1, (s_2, l_2), a_2, \ldots)$ be an infinite run of the context-aware execution domain Σ_{CF}. π satisfies requirements expr defined over C with preference x if and only if for any $n \in \mathbb{N}$ the following holds:*

- *there exists $m : n \leq m < \infty$ such that the starting subsequence $\pi^m = ((s_1, l_1), a_1, (s_2, l_2), a_2, \ldots, a_{m-1}, (s_m, l_m))$ of π is such that $\pi^m \models^{x'} expr : x' \geq x$;*

- *there exists no* $m' : n \leq m' < \infty$ *such that the starting subsequence* $\pi^{m'} = ((s_1, l_1), a_1, (s_2, l_2), a_2, \ldots, a_{m'-1}, (s'_m, l'_m))$ *of* π *is such that* $\pi^{m'} \models^{x''} expr : x'' < x$.

The idea is that infinite runs are supposed to keep on satisfying the composition requirements forever. Once requirements are satisfied, they still may become unsatisfied (due to uncontrollable actions executed by services). In this case, their satisfaction has to always be reached again in finite number of steps. The satisfaction preference for an infinite run is determined by a minimal satisfaction preference provided by this run in the infinite perspective.

7.1.2 Requirements as STS

From Def. 45 we see that the evaluation of satisfaction of requirements *expr* by some run of the context-aware execution domain can be reduced to the evaluation of satisfaction of clauses within tasks of expressions of *expr*. Consequently, a clause can be evaluated by evaluating all its terms, which are context formulas and events. The evaluation of satisfaction of a context formula ρ consists in tracking if we have traversed a state satisfying ρ. Similarly, the evaluation of satisfaction of event e consists in tracking if we have executed an action annotated with effect containing e. The tracking of satisfaction for terms appearing in imperative statements is different from the tracking of satisfaction of those appearing in reaction statements. For the imperative statements we track the corresponding situations starting from the very beginning of the system run. For the reaction statements we track the situations of interest starting from the last appearance of the reaction trigger as required by Def. 45. In other words, every time trigger fires during the system run we have to reset all the tracking information and start tracking anew.

In order to introduce the satisfaction of our composition requirements in the context-aware execution domain, we generally follow the approach proposed for planning with extended goals ([67]), where a special *control automaton* is introduced in order to track requirements satisfaction.

We define our *elementary control STS* as a state transition system tracking the satisfaction of one single term. It has only two states $false$ and $true$ and the transitions between states can be labeled with events or context formulas:

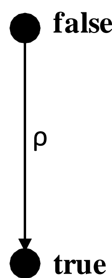
Definition 47 (Elementary Control STS). *An elementary control STS for context C is a state transition system $r = \langle V, v^0, T \rangle$ such that:*

- *$V = \{true, false\}$ is a set of control states and $v^0 \in V$ is the initial state;*
- *$T \subseteq V \times \{R_C \cup E_C\} \times V$ is a set of transitions labelled with context formulas or events of C.*

In fact, we can notice that in order to track the satisfaction of an arbitrary requirement *expr* expressed in our language we have to track four elementary situations: 1) the occurrence of a context formula, 2) the occurrence of an event, 3) the occurrence of a context formula after an event and 4) the occurrence of an event after another event. For that purpose, we define the following four elementary control STSs:

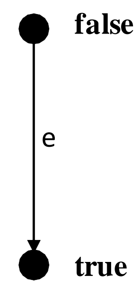
Definition 48 (Elementary Control STS for Language). *To translate our language to STS we use four elementary control STSs:*

- *to track a context formula ρ in imperative expressions we use elementary control STS r_ρ*



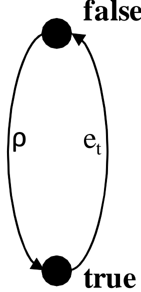
where the initial state is set to true if for the initial context configuration $l_C^0 \models \rho$*, and set to* $false$ *otherwise;*

- *to track a context event* e *in imperative expressions we use elementary control STS* r_e



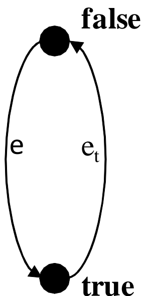
where the initial state is set to $false$ *if for the initial context configuration* $l_C^0 \models \rho$*;*

- *to track a context formula* ρ *in reaction expressions with trigger* e_t *we use elementary control STS* $r_{e_t\rho}$



where the initial state is set to true;

- *to track a context event e in reaction expressions with trigger e_t we use elementary control STS $r_{e_t e}$*



where the initial state is set to true.

The semantics of elementary control STSs with respect to the context-aware execution domain will be presented later. A control STS for some requirements *expr* contains elementary control STSs for all terms appearing in *expr*:

Definition 49 (Control STS for Requirements)**.** *A control STS for some requirements $expr = expr_1; expr_2; \ldots; expr_n$ defined over context C, where $expr_i$ is an elementary expression for all $i \in [1, n]$, is a set of elementary control STSs $R(expr)$ such that:*

- *for each context formula $\rho \in R_C$ if there exists $i \in \mathbb{N}$ such that $expr_i = task$ (i.e., $expr_i$ is an imperative expression) and $\rho \in Terms(task)$ then $\exists r_\rho \in R(expr)$;*
- *for each context event $e \in E_C$ if there exists $i \in \mathbb{N}$ such that $expr_i = task$ and $e \in Terms(task)$ then $\exists r_e \in R(expr)$;*
- *for each context formula $\rho \in R_C$ if there exists $i \in \mathbb{N}$ such that $expr_i = e_t \rightarrow task$ and $\rho \in Terms(task)$ then $\exists r_{e_t\rho} \in R(expr)$;*

- *for each context event* $e \in E_C$ *if there exists* $i \in \mathbb{N}$ *such that* $expr_i = e_t \rightarrow task$ *and* $e \in Terms(task)$ *then* $\exists r_{e_te} \in R(expr)$*;*
- *no other STSs belong to* $R(expr)$*.*

The state of the Control STS is described by states of its constituent elementary control STSs. If $R(expr) = r_1, \ldots, r_m$ *such that* $r_i = \langle V_i, v_i^0, T_i \rangle, i \in [1, m]$ *then the initial state is* $v_R^0 = (v_1^0, \ldots, v_m^0)$ *and the set of all states is* $V_R = \prod_{i=1}^{m} V_i$*.*

The control STS $R(expr)$ can be used to evaluate any clause within *expr* and, as such, can also be used to evaluate requirements *expr*. In the following we propagate the notion of requirements satisfaction introduced by Defs. 43, 44 and 45 for runs of the context-aware execution domain to the states of the control STS. We use the concept of state projection as it was defined for context in Def. 4.

Definition 50 (Requirements Satisfaction (Control STS))**.** *Let* $R(expr)$ *be a control STS for requirements* $expr = expr_1; expr_2; \ldots; expr_n$ *defined over context* C*, where* $expr_i$ *is an elementary expression for all* $i \in [1, n]$*, and let* $v_R \in V_R$ *be one of its state.*

Clause cl belonging to some task within elementary expression $expr_k$: $k \in [1, n]$ *is satisfied by state* v_R *(denoted* $v_R \models cl$*) if and only if one of the following holds:*

- $cl = \rho$ *and cl belongs to imperative expression and* $v_R \downarrow_{r_\rho} = true$*;*
- $cl = \rho$ *and cl belongs to a reaction expression with trigger* e_t *and* $v_R \downarrow_{r_{e_t\rho}} = true$*;*
- $cl = e$ *and cl belongs to an imperative expression and* $v_R \downarrow_{r_e} = true$*;*
- $cl = e$ *and cl belongs to a reaction expression with trigger* e_t *and* $v_R \downarrow_{r_{e_te}} = true$*;*

- $cl = cl_1 \vee cl_2$, *and* $r \models cl_1$ *or* $v_R \models cl_2$*;*
- $cl = cl_1 \wedge cl_2$, *and* $r \models cl_1$ *and* $v_R \models cl_2$*.*

If $expr_k : k \in [1,n]$ *is an imperative expression such that* $expr_k = (cl_1, 1), (cl_2, 2), \ldots, (cl_m, m)$ *then state* r *satisfies* $expr_k$ *with preference* $i : 1 \leq i \leq m$ *(denoted* $v_R \models^i expr_k$*) if and only if* $v_R \models cl_i$ *and* $\nexists j : i < j \leq m : v_R \models cl_j$*.*

If $expr_k : k \in [1,n]$ *is a reaction expression such that* $expr_k = e_t \rightarrow (cl_1, 1), (cl_2, 2), \ldots, (cl_m, m)$ *then state* v_R *satisfies* $expr_k$ *with preference* $i : 1 \leq i \leq m$ *(denoted* $v_R \models^i expr_k$*) if and only if* $v_R \models cl_i$ *and* $\nexists j : i < j \leq m : v_R \models cl_j$*.*

Finally, v_R *satisfies requirements* $expr$ *with preference* $x \in \mathbb{N}$ *(denoted* $v_R \models^x expr$*) if and only if there exists a set of natural numbers* $c_1, c_2, \ldots, c_n : c_i \in \mathbb{N}, i \in [1,n]$ *such that* $v_R \models^{c_i} expr_i$ *for all* $i \in [1,n]$ *and* $c_1 + c_2 + \ldots + c_n = x$*.*

Example 12. In Fig. 7.1 we show elementary control STS that are used to track and evaluate expression

$$canceled^e(flight) \rightarrow (unbooked^e(package) \wedge unbooked^e(hotel), 1)$$

in Example 11. STS r_1 is used to track the occurrence of event $unbooked^e(package)$ after event $canceled^e(flight)$, while STS r_2 is used to track the occurrence of event $unbooked^e(hotel)$ after the same event $canceled^e(flight)$. As a result, the combination of two STSs can be used to track the satisfaction of the whole expression. The requirement is considered as satisfied when the resulting control STS is in state $(r_1 = true \wedge r_2 = true)$.

In order to be able to fuse control STS with the context-aware execution domain so that the resulting STS reflects the satisfaction of composition

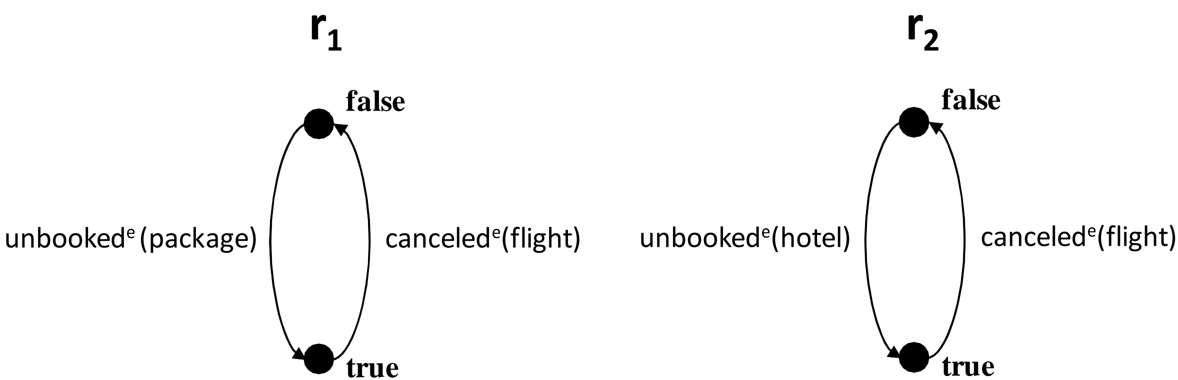


Figure 7.1: Requirements as STS

requirements at any point of domain evolution, we, using the analogy with the impact function for context, introduce the *requirements impact* function for control STS. It indicates how the execution of certain action in certain conditions affects the control STS:

Definition 51 (Requirements Impact)**.** *Let* $\Psi = \langle F^+, C\rangle$ *be a context-aware system, let* $\Sigma_{CF} = \langle \mathcal{S}_{CF}, s^0_{CF}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_{CF}\rangle$ *be the context-aware execution domain for* Ψ*. Let expr be composition requirements over* C *and let* $R(expr)$ *(with set of states* V_R*) be its control STS. The impact of transition* $t \in \mathcal{R}_{CF}$ *on the control STS* $R(expr)$ *in state* $v_R \in V_R$ *is a state* $v'_R \in V_R$ *(denoted* $v'_R = ReqImp_\Psi[R(expr)](v_R, t)$*) such that:*

- *if* $t = (s, a, s')$*, then for all* $r = \langle V, v^0, T\rangle \in R(g)$ *if there exists* $(v_R\downarrow_r, e, v') \in T$ *such that* $e \in \mathcal{E}(a)$ *then* $v'_R\downarrow_r = v'$*;*
- *if* $t = (s, a, s')$*, then for all* $r = \langle V, v^0, T\rangle \in R(g)$ *if there exists* $(v_R\downarrow_r, \rho, v') \in T$ *such that* $s' \models \rho$ *then* $v'_R\downarrow_r = v'$*;*
- *in all other cases for all* $r \in R(g)$ *the state of* r *does not change, i.e.,* $v'_R\downarrow_r = v_R\downarrow_r$*.*

In fact, the requirements impact shows that once an event fires in the

domain, it triggers all transitions in elementary control STSs that are labeled with this event. Similarly, when some new state is reached in the domain, it also triggers all transitions labeled with formulas satisfied by the new state.

Having the notion of requirements impact, we fuse the context-aware execution domain and the control STS into a *context-aware execution domain with tracking*, or simply domain with tracking. The state of the domain with tracking is supposed to reflect not only the evolution of the context-aware execution domain but also the satisfaction of composition requirements in the current situation. In fact, the domain with tracking encodes the simultaneous execution of the context-aware execution domain and the control STS, which "tracks" the evolution of the former and evolves according to the requirements impact function. In this regard, the control STS plays the role of "passive logger" of the execution domain and by no means restricts its evolution.

Definition 52 (Domain with Tracking)**.** *Let $\Psi = \langle F^+, C\rangle$ be a context-aware system, let $\Sigma_{CF} = \langle \mathcal{S}_{CF}, s^0_{CF}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_{CF}\rangle$ be the context-aware execution domain for Ψ. Let expr be composition requirements over C and let $R(expr)$ (with set of states V_R and initial state v^0_R) be its control STS. The domain with tracking for Ψ and expr is a state transition system $\Sigma_{RCF} = \langle \mathcal{S}_{RCF}, s^0_{RCF}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_{RCF}\rangle$ such that:*

- *$\mathcal{S}_{RCF} = \mathcal{S}_{CF} \times V_R$ is a set of states and $s^0_{RCF} = (\mathcal{S}_{CF}, v^0_R)$ is the initial state;*
- *the transition relation $\mathcal{R}_{RCF}$ is such that:*

$$((s, v_R), a, (s', v'_R)) \in \mathcal{R}_{RCF} \textit{ if } (s, a, s') \in \mathcal{R}_{CF}, \textit{ and}$$
$$v'_R = ReqImp_\Psi[R(expr)](v_R, (s, a, s')).$$

From the definition above it can be easily observed that for every run

$$\pi = (s_1, a_1, s_2, a_2, \ldots, a_{n-1}, s_n)$$

of Σ_{CF} there exists run

$$\pi_R = ((s_1, v_1^R), a_1, (s_2, v_2^R), a_2, \ldots, a_{n-1}, (s_n, v_n^R))$$

of the tracking domain Σ_{RCF}. It is the direct consequence of the fact that requirements impact can be calculated for any set of arguments. In other words, the domain with and without tracking has the same set of runs with respect to the operations executed. And so we can conclude that control STS does not restrict in any way the set of possible evolutions of the context-aware execution domain within the respective domain with tracking. At the same time, the control STS itself evolves only according to requirements impact. Consequently, the state of the tracking domain is supposed to reflect the satisfaction of requirements *expr* as defined by Def. 51.

The central idea behind the domain with tracking is that every its state unambiguously shows if requirements *expr* are satisfied in it and with which preference. As such, the domain with tracking "converts" complex extended goals expressed in our language into reachability goals. Consequently, it allows us to use planning techniques similar to those presented in Chapter 4 to resolve the composition problems with goals expressed in language of Def. 42.

We remark that the way we build a control STS and fuse it with the context-aware execution domain is an alternative way to describe the semantics of the requirements language from the perspective of planning. In this regard, Defs. 43, 44 and 45 and the theory of this section essentially serve for the same purpose and can be used independently, regarding the final goal of such definition. The execution-based definitions give better conceptual understanding of the language semantics. However, they

cannot be directly reused in planning. The state-based approach is more technical but is ready to be integrated with planning algorithms similar to those exploited in Chapter 4. Being complementary to each other, the two approaches demonstrate the methodology of how to use complex and extended control-flow requirements in planning.

In the conclusion of this section we introduce a theorem stating the equivalence of the satisfaction semantics for the execution-based definitions and for the domain with tracking:

Theorem 3 (Semantics of Domain With Tracking)**.** *Let $\Psi = \langle F^+, C\rangle$ be a context-aware system, let Σ_{CF} be the context-aware execution domain for Ψ. Let expr be composition requirements over C and let Σ_{RCF} be domain with tracking for Ψ and expr. A run*

$$\pi = (s_1, a_1, s_2, a_2, \ldots, a_{n-1}, s_n)$$

of the context-aware domain Σ_{CF} satisfies expr with preference x (i.e., $\pi \models^x expr$) if and only if its equivalent run

$$\pi_R = ((s_1, v_1^R), a_1, (s_2, v_2^R), a_2, \ldots, a_{n-1}, (s_n, v_n^R))$$

of the domain with tracking Σ_{RCF} is such that $v_n^R \models^x expr$;

Proof. The proof of the theorem is based on the way the domain with tracking is built. Using the Defs. 43, 44 and 45 (satisfaction for execution paths) on the one side and Def. 50 (satisfaction for states of control STS) on the other side, we can incrementally show that the theorem holds for clauses, elementary requirement expressions, and complex requirement expressions. Consequently, it is easy to show that the satisfaction is also preserved for infinite runs (Def. 46) □

7.1.3 Problem of Continuous Composition

Similarly to context-aware execution domain in Chapters 4 and 6, a domain with tracking $\Sigma_{RCF} = \langle \mathcal{S}_{RCF}, s^0_{RCF}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_{RCF} \rangle$ defined for some context-aware system Ψ and requirements *expr* can be used as a search space for service compositions for requirements *expr*. Naturally enough, we are interested in those runs of a domain with tracking that terminate in the states satisfying *expr*. However, differently from the approach of Chapter 4, the goal states of Σ_{RCF} are ordered according to their preferences. This has to be taken into account by potential solutions.

Let us consider some composition requirements $expr = expr_1; expr_2; \ldots; expr_n$ where $expr_i$ is an elementary expression for all $i \in [1, n]$. We assume that for each $expr_i$ the tasks are given preferences $1, 2, \ldots, Max[expr_i]$ without gaps. Then all the goal states

$$G = \{v_R \in \mathcal{S}_{RCF} : \exists x \in \mathbb{N} : v_R \models^x expr\}$$

can be split into $Max[expr] = \sum_{i=1}^{n} Max[expr_i]$ groups $G_1, \ldots, G_{Max[expr]}$ containing only those goal states that satisfy requirements with certain preference:

$$G_i = \{v_R \in \mathcal{S}_{RCF} : v_R \models^i expr\}, i \in [1, n].$$

In order to introduce the notion of solution executor for a domain with tracking we adopt Def. 21 and neglect the preferences of goal states.

Definition 53 (Solution Executor(Tracking)).
Let $\Sigma_{RCF} = \langle \mathcal{S}_{RCF}, s^0_{RCF}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_{RCF} \rangle$ be a domain with tracking for context-aware system $\Psi = \langle F^+, C \rangle$ and requirements expr over C. A solution executor for Ψ and expr is an STS $\Sigma_E = \langle \mathcal{S}_E, s^0_E, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_E \rangle$ such that:

- $\mathcal{S}_E \subseteq \mathcal{S}_{RCF}$, $s^0_E = s^0_{RCF}$, $\mathcal{R}_E \subseteq \mathcal{R}_{RCF}$, *i.e.,* Σ_E *is a subgraph of* Σ_{RCF}*;*

- *$Finals(\Sigma_E) \subseteq G$, i.e., all complete runs of Σ_E are satisfying for $expr$.*

In the same way we adopt the notion of consistent executor:

Definition 54 (Consistent Executor (Tracking)).
Let $\Sigma_{RCF} = \langle \mathcal{S}_{RCF}, s^0_{RCF}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_{RCF} \rangle$ be a domain with tracking for context-aware system $\Psi = \langle F^+, C \rangle$ and requirements expr over C. A solution executor $\Sigma_E = \langle \mathcal{S}_E, s^0_E, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_E \rangle$ for Ψ and expr is consistent if:

1. *for all $s \in \mathcal{S}_E$, if exists transition $(s, a', s') \in \mathcal{R}_E : a \in \mathcal{I}_F$, then it is the only transition from this state ($\nexists (s, a'', s'') \in \mathcal{R}_E$);*
2. *if $s \in \mathcal{S}_E : s \not\models \rho$ then $\forall (s, a, s') \in \mathcal{R}_{RCF} : (a \in \mathcal{O}_F) \rightarrow ((s, a, s') \in \mathcal{R}_E))$ (all uncontrollable actions are considered);*
3. *for every complete run π_E of Σ_E any state $s \in \mathcal{S}_E$ is traversed no more than once, i.e., Σ_E does not contain infinite runs.*

Finally we have to address the problem of preferences in goals. Indeed, when multiple consistent solution executors exist our choice has to be regulated by the preferences of goals they achieve. We remark that the problem of preferences is an important topic in AI planning that, however, goes beyond the scope of this work. In our approach, we adopt the way to order solution proposed in [111] and [110] and reuse some definitions from these works.

Considering some consistent solution executor $\Sigma_E = \langle \mathcal{S}_E, s^0_E, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_E \rangle$, we denote with $best(\Sigma_E, s)$ the highest preference among the final states of Σ_E reachable from state $s \in \mathcal{S}_E$. Similarly, with $worst(\Sigma_E, s)$ we denote the lowest preference among the final states reachable from s. So we can say, that Σ_E gives a possibility to reach at most goal state with preference $best(\Sigma_E, s)$ while guaranteeing that a state with preference at least $worst(\Sigma_E, s)$ will be reached. While comparing two

executors $\Sigma_E^1 = \langle \mathcal{S}_E^1, \mathcal{S}_E^{0\ 1}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_E^1 \rangle$ and $\Sigma_E^2 = \langle \mathcal{S}_E^2, \mathcal{S}_E^{0\ 2}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_E^2 \rangle$ we use so-called optimistic approach: the executor whose best preference is higher for this state is better. When the best preferences are equal we prefer the one with higher worst preference:

Definition 55 (Executor Ordering in State). *Let Σ_E^1 and Σ_E^2 be two consistent solution executors for the same context-aware domain Ψ and requirements expr and let s be a state belonging to both of them (i.e., a state of their respective domain with tracking). Then Σ_E^1 is better than Σ_E^2 in s (denoted $\Sigma_E^1 >^s \Sigma_E^2$) if:*

- *$best(\Sigma_E^1, s) > best(\Sigma_E^2, s)$, or*
- *$best(\Sigma_E^1, s) = best(\Sigma_E^2, s)$ and $worst(\Sigma_E^1, s) > worst(\Sigma_E^2, s)$.*

If $best(\Sigma_E^1, s) = best(\Sigma_E^2, s)$ and $worst(\Sigma_E^1, s) = worst(\Sigma_E^2, s)$ then executors are equivalent in this state (denoted $\Sigma_E^1 \simeq^s \Sigma_E^2$). We denote with $\Sigma_E^1 \geq^s \Sigma_E^2$ the fact that $\Sigma_E^1 >^s \Sigma_E^2$ or $\Sigma_E^1 \simeq^s \Sigma_E^2$.

Consequently, we order executors by considering their properties in common states $\mathcal{S}_{common}(\Sigma_E^1, \Sigma_E^2) = \mathcal{S}_E^1 \cap \mathcal{S}_E^2$:

Definition 56 (Executor Ordering). *Let Σ_E^1 and Σ_E^2 be two consistent executors for the same context-aware domain Ψ and requirements expr. Executor Σ_E^1 is better than executor Σ_E^2 (denoted $\Sigma_E^1 > \Sigma_E^2$) if:*

- *$\Sigma_E^1 \geq^s \Sigma_E^2$ for all $s \in \mathcal{S}_{common}(\Sigma_E^1, \Sigma_E^2)$ and*
- *$\Sigma_E^1 >^{s'} \Sigma_E^2$ for some $s' \in \mathcal{S}_{common}(\Sigma_E^1, \Sigma_E^2)$.*

If $\Sigma_E^1 \simeq^s \Sigma_E^2$ for all $s \in \mathcal{S}_{common}(\Sigma_E^1, \Sigma_E^2)$ then the two executors are equivalent (denoted $\Sigma_E^1 = \Sigma_E^2$).

Finally, the consistent solution executor is considered to be optimal if is in not worse than all the other possible consistent solution executors for the same system Ψ and requirements *expr*:

Definition 57 (Optimal Executor)**.** *Consistent solution executor* Σ_E *for context-aware domain* Ψ *and requirements expr is optimal if* $\Sigma_E \geq \Sigma'_E$ *for any other executor* Σ'_E *for* Ψ *and expr.*

The last important aspect we would like to reflect in the solution is that the preferable goal states are not always the terminal states of the execution domain and sometimes the system can be forced to leave such state through uncontrollable transitions. Consider, for example, the VTA scenario. Our booking requirement is satisfied when all the appropriate bookings are transactionally completed. However, we know that the goal state reached is actually not a final state of the domain and uncontrollable transitions are available from it (namely, those corresponding to flight delay and cancellation). Since we cannot prevent uncontrollable transitions from being triggered by external services, the idea is to take them into account in the executor, so that it is ready to react to this kind of situations.

First of all, a consistent executor has to include (and thus provide a solution for) all uncontrollable transitions also from goal states, which is different from condition 2 of Def. 54. Conceptually, once we reached some goal state we still have to consider the situation, in which we leave this state through an uncontrollable transition (if any), and guarantee that in this case we "bring" the system to the goal state. This is what we mean by continuous composition.

The second change in the definition of consistent orchestrator (Def. 54) concerns condition 3, prohibiting infinite loops. The point is that in continuous composition loops are affordable and almost unavoidable. Indeed, once you leave a goal state, an affordable solution may be to come back to this state. For example in the VTA scenario when the flight is delayed in a situation when everything is booked, the best solution is to adjust all other components (i.e., hotel and package) to the flight changes and, in fact, bring the system back to the state where the delay initially happened.

Such solution can still be considered as strong since, though we have loops, they are loops traversing goal states. As a result, we guarantee that from any its state, the goal state is reached in finite number of steps. Formally, the *continuosly consistent executor* is defined as follows:

Definition 58 (Continuously Consistent Executor).
Let $\Sigma_{RCF} = \langle \mathcal{S}_{RCF}, s^0_{RCF}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_{RCF} \rangle$ be a domain with tracking for context-aware system $\Psi = \langle F^+, C \rangle$ and requirements expr over C. A solution executor $\Sigma_E = \langle \mathcal{S}_E, s^0_E, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_E \rangle$ for Ψ and expr is continuously consistent if:

1. *for all $s \in \mathcal{S}_E$, if exists transition $(s, a', s') \in \mathcal{R}_E : a \in \mathcal{I}_F$, then it is the only transition from this state ($\nexists (s, a'', s'') \in \mathcal{R}_E$);*
2. *if $s \in \mathcal{S}_E$ then $\forall (s, a, s') \in \mathcal{R}_{RCF} : (a \in \mathcal{O}_F) \rightarrow ((s, a, s') \in \mathcal{R}_E))$ (all uncontrollable actions are considered);*
3. *for all states $s \in \mathcal{S}_E$, there exists no infinite run $\pi = (s_1, a_1, s_2, a_2, \ldots) : s_1 = s)$ of Σ_{RCF} such that $\forall i \in [1, \infty) : s_i \not\models expr$.*

Comparing this definition with Def. 54, we see that condition 2 requires that all uncontrollable transitions are considered also in goal states. Moreover, condition 3 is reformulated such that it allows for loops but only in case this loops traverse goal states. This ensures that from any state of the executor a goal is guaranteed to be achieved in finite number of steps.

We remark that continuously consistent executors can be compared to each other using the optimality criterion introduced by Defs. 55,56 and 57.

Finally, the problem of continuous service composition for extended control-flow requirements can be introduced as follows:

Definition 59 (Continuous Service Composition Problem). *Let S^+ be a set of services annotated over context C and let F^+ be a set of annotated fragments that are complimentary to S^+. The problem of continuous service*

composition for annotated services S^+, context model C and requirements expr expressed in language of Def. 42 over C consists in finding optimal and continuously consistent solution executor for context-aware system $\Psi = \langle F^+, C\rangle$ and requirements expr.

7.1.4 Approach Architecture and Prototype Algorithm

Approach Architecture

As it can be seen in Fig. 7.2, our approach to continuous composition generally follows the flavour of fragment composition approach depicted in Fig. 4.7 of Chapter 4. The main difference concerns the appropriate handling of control-flow requirements and their integration to the planning problem.

The derivation of the context-aware execution domain Σ_{CF} is the same as in Chapter 4 except for the fact that since we start from services, the A-EXTRACTOR has to additionally convert annotated services $s_1^+, \ldots, s_n^+$ to annotated complementary fragments $f_1^+, \ldots, f_n^+$.

The requirements processing is performed by the ENCODER. It essentially consists in defining the elementary control STS for *expr* and grounding them on annotations of fragments. The procedure of grounding is conceptually similar to the grounding of context: in every elementary STS all transitions labeled with context events and context formulas are intelligently replaced with transitions labelled with fragment actions. The grounding is performed such that when the grounded elementary STSs are joined into synchronous product Σ_R and then Σ_R is synchronously joined with the context-aware execution domain Σ_{CF} into STS Σ_{RCF}, the resulting STS Σ_{RCF} completely corresponds to the notion of domain with tracking of Def. 52. Another function of the ENCODER is to derive from requirements *expr* reachability goals $G = \{G_1, \ldots, G_{Max[expr]}\}$, which is

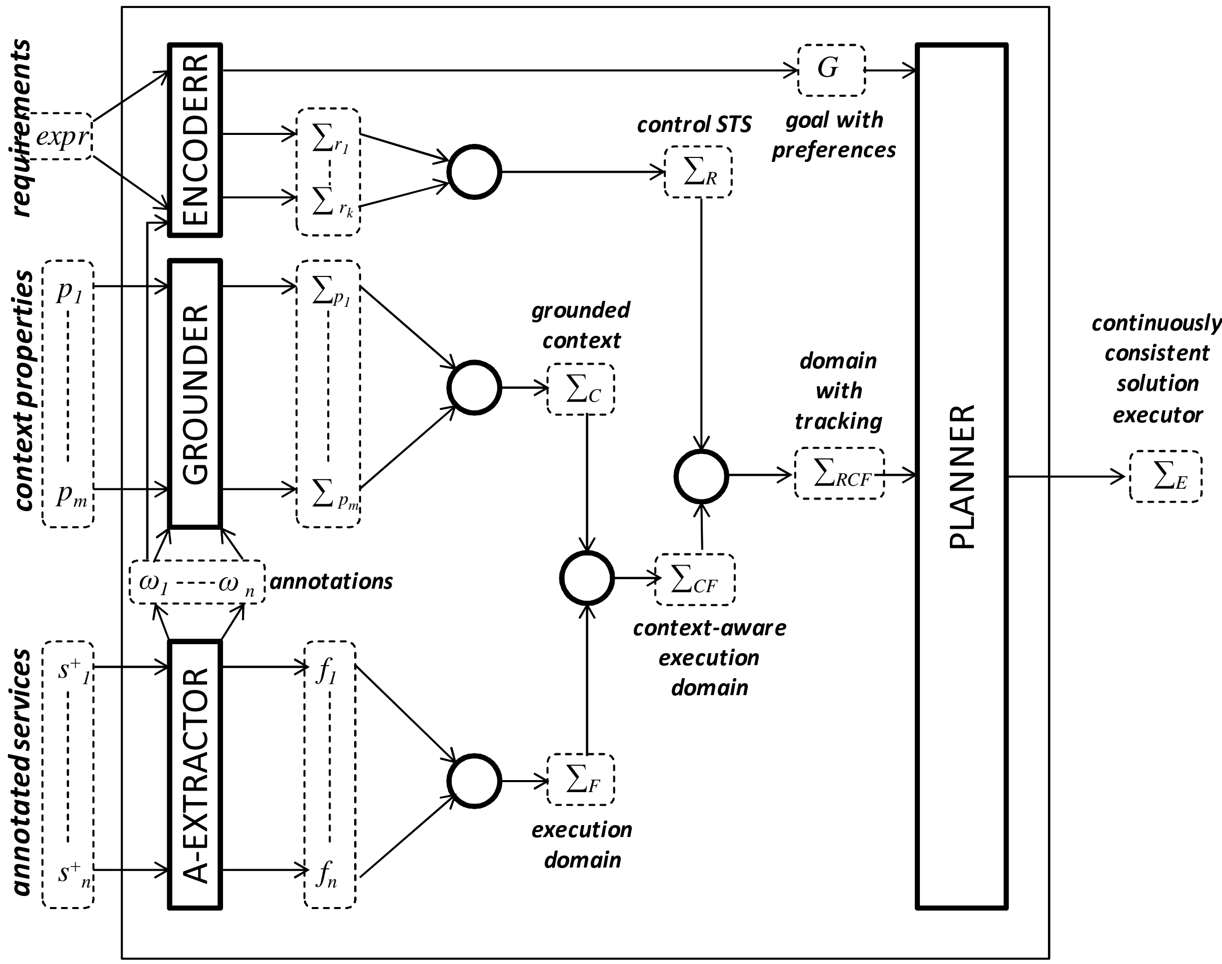


Figure 7.2: Continuous composition approach

essentially the list of satisfying states of the control STS Σ_R ordered according to their preferences (e.g., as it is shown in the beginning of Section 7.1.3).

After the domain with tracking Σ_{RCF} and the goal with preferences G are passed to the PLANNER, the latter is expected to produce the optimal and continuously consistent solution executor Σ_E that can further be converted into an executable process.

Prototype Algorithm

Once the domain with tracking $\Sigma_{RCF} = \langle \mathcal{S}_{RCF}, s^0_{RCF}, \mathcal{I}_F, \mathcal{O}_F, \mathcal{R}_{RCF} \rangle$ is passed to the planner, it becomes a planning domain $D = \Sigma_{RCF}$. For our convenience we will omit the indices and denote the domain as follows: $D = \langle \mathcal{S}, s^0, \mathcal{I}, \mathcal{O}, \mathcal{R} \rangle$. Its initial state becomes the initial state of the planning problem $I = s^0$. Finally, the set of goal states ordered according to their preferences becomes a goal with preferences $G = \{G_1, \ldots, G_n\}$ where $G_i \subseteq \mathcal{S}, i \in [1, n]$ is a set of goal states of preference i (see Section 7.1.3). The resulting composition problem $\langle D, I, \{G_1, \ldots, G_n\} \rangle$ is similar to the one used in [111] and is known as a *planning problem with preferences.*

In the algorithm description we keep on using the PruneStates and StrongPreImage routines introduced in Section 4.5.

Our prototype algorithm for deriving optimal continuously consistent solution executor (later in this section simply solution executor)adopts some ideas of [111]. The algorithm contains two major steps:

1. As a first step, we reduce the set of goal states $G = \{G_1, \ldots, G_n\}$ to their subset $G' = \{G'_1, \ldots, G'_n\}$ such that $G'_i \subseteq G_i, i \in [1, n]$. Goal states G' are a maximal subset of goal states G such that all uncontrollable transitions from G' lead to a state from which a strong plan for G' exists;

2. The second step is based on the backward search for the pruned goal states $G' = \{G'_1, \ldots, G'_n\}$. The plan search routines used here are close to those used in [111].

The pre-processing of goal states (Step 1) is necessary since some of goal states are not suitable for continuous composition. Indeed, when we reach a goal state among G, the only thing that is guaranteed is that the requirements are satisfied in this state. However, the continuous composition has

to guarantee that whenever a goal state is left through an uncontrollable transition we can always bring it back to some goal state. Therefore we have to filter off all the goal states that do not fit this criterion.

The filtering routines implementing the first step are shown in picture 7.3. Here and in the following algorithms we assume that the planning domain D is globally available. The routine FILTERSTEP takes as input a list of goal states $G = \{G_1, \ldots, G_n\}$ and returns a set of filtered goal states $G' = \{G'_1, \ldots, G'_n\}$. First, all the goals are collected into single set $flatGoal$ (lines 2-4). Then the backward pre-imaging for states $flatGoal$ is performed (lines 5) using the routine STRONGPREFP (lines 14-23) that implements strong pre-imaging for given states till the most fixed point. The result of pre-imaging is stored in state-action table SA. Finally, we prune all the goal states from *gList* that have uncontrollable transitions leading to states outside SA (line 6-11), i.e., to states from which no strong plan for $flatGoal$ exists. Suc goal states are "bad" since for them the continuous maintenance of requirements is not guaranteed. The main filtering routine FILTER (lines 26-33) consists in running FILTERSTEP till the least fixed point. It is necessary since after removing some goalStates within FILTERSTEP we may "make" some other goal states unsuitable for continuous composition and so the filtering should continue till the least fixed point.

The termination of the filtering algorithm can be easily proved by showing that at each iteration of cycle in lines 29-32 at least one goal state is filtered off. Since the initial set of goal states G is finite, the loop always terminates. The goal state filtering performed by the FILTER routine does not affect the completeness of the further planning algorithm and does not eliminate potential solutions. To show that, we prove that solution executor never traverses "bad" goal states eliminated by the filtering procedure. As a consequence, it is erroneous to consider them as "truly" goal

states since once they are included in the executor, the latter cannot be continuously consistent.

Let $G_{total} = \bigcup_{i=1}^{n} G_i$ be a union of all goal states of G, and let $G'_{total} = \bigcup_{i=1}^{n} G'_i$ be a set of goal states remained after filtering ($G'_{total} \subseteq G_{total}$).

Lemma 3 (Filtering)**.** *Let $D = \langle \mathcal{S}, s^0, \mathcal{I}, \mathcal{O}, \mathcal{R} \rangle$ be some domain with tracking, let G be its ordered set of goal states and let $\Sigma_E = \langle \mathcal{S}_E, s^0_E, \mathcal{I}, \mathcal{O}, \mathcal{R}_E \rangle$ be its solution executor for goals G. Let G'_{total} be a set of states remained after filtering by routine* FILTER*. Then $\mathcal{S}_E \cap (G_{total} \setminus G'_{total}) = \emptyset$, i. e. the solution executor never traverses goal states removed by filtering.*

Proof. Let us denote with $G_{bad} = G_{total} \setminus G'_{total}$ a set of goal states eventually filtered off. Since the filtering algorithm terminates then we will have finite number m of iterations of loop in lines 29-32, each of which will filter off states G^i_{bad} so that $G_{bad} = \bigcup_{i=1}^{m} G^i_{bad}$. Using induction on the number i of iteration of the filtering loop we show that once a state $s \in G_{bad}$ is included in the solution executor Σ_E (i.e., $s \in \mathcal{S}_E$) there exists an infinite run of Σ_E never reaching goal states G_{total} in infinite perspective, which contradicts to condition 3 of Def. 58 of continuously consistent executor.

Basis ($i = 1$). If $s \in G^1_{bad}$, then there exists an uncontrollable transition (s, o, s') such that from s' a strong solution for G_{total} does not exist. This transition also belongs to Σ_E (condition 2 of Def. 58). Since s' is not a goal state, s' cannot be final state of Σ_E. At the same time, 1) if s' contains only controllable outgoing transitions then all of them lead to states from which no strong solution for G_{total} exists and 2) if s' contains uncontrollable outgoing transitions, then at least one of them leads to a state from which no strong solution for G_{total} exists (otherwise, strong solution from s would also exist). Then we can conclude that there exists $(s', a', s'') \in \mathcal{R}_E$, such

```
1  function filterStep(gList)
2    flatGoal = ∅;
3    for (i:=1;i≤|gList|;i++)
4      flatGoal := flatGoal ∪ gList[i]
5    SA := strongPreFP(flatGoal)
6    gListPruned := ∅
7    for (i:=1;i≤|gList|;i++
8      foreach(g∈gList[i])
9        nextUncontrollable = {s∈𝒮: ∃ (g,a,s)∈ℛ, a∈𝒪}
10       if(nextUncontrollable⊆StatesOf(SA))
11         gListPruned[i] = gListPruned[i]∪g
12 return gListPruned
13
14 function strongPreFP(goal)
15   OldSA := ∅
16   NewSA := ∅
17   do
18     OldSA := NewSA
19     Pr := StrongPreImage(goal ∪ StatesOf(OldSA))
20     NewStates := PruneStates(Pr, goal ∪ StatesOf(OldSA))
21     NewSA := OldSA ∪ NewStates
22   while (OldSA ≠NewSA)
23 return NewSA
24
25
26 function filter(gList)
27   gListOld = ∅
28   gListPruned = gList
29   do
30     gListOld = gListPruned
31     gListPruned = filterStep(gListOld)
32   while (gListNew ≠ gListOld)
33 return gListPruned
```

Figure 7.3: Goal filtering algorithm

that $s'' \notin G$. Continuing the same reasoning we show that there exists an infinite run of Σ_E that never reaches goal states G_{total}.

Induction step. Let us denote with $G_{bad}^{[1,i]} = \bigcup_{j=1}^{i} G_{bad}^{j}$ a set of goal states filtered off after i iterations of the loop of lines 29-32 and for which the lemma is already proved. Let us denote with $G_{total}^{[1,i]} = G_{total} \setminus G_{bad}^{[1,i]}$ a set of goal states still remaining in consideration. If $s \in G_{bad}^{i+1}$ then using the same reasoning as in the Basis we can show that there exists a run of Σ_E that never reaches states of $G_{total}^{[1,i]}$. The only possibility for it to still be continuously consistent (condition 3 of Def. 58) is by constantly traversing states of $G_{bad}^{[1,i]}$ but it is already proved that the inclusion of states of states $G_{bad}^{[1,i]}$ results in an infinite run of Σ_E that never reaches G_{total} in infinite presepctive. □

The second step, which is the plan search itself, is a modification of the algorithm proposed in [111]. The main idea is to consider each of the goal sets G_i one by one, and to build for each of them a state-action table that shows for the states of D, which action (if any) leads towards G_i. In this way, it is fairly easy to finally merge state-action tables into an overall plan, by layering them according to the respective preferences. In particular, as discussed in [111], to correctly consider preferences so that the resulting overall plan is optimal, one has to build state-action tables starting from the less preferred goal and going to the most preferred one: only in this way, the state-action tables built for $G_i : i \in [1, j]$ can be used as a "recovery" basis for the state-action table referring to the more preferable goal G_{j+1}.

The core of this algorithm is the COMPUTESATABLES routine in Fig. 7.4, whose structure is inspired by the work in [111]. Starting from the lowest preference 1, for each preference i it performs the following steps:

```
1  function computeSATables(gList)
2    for (i:=1;i≤|gList|;i++)
3       SA     := StrongPreFP(gList[i])
4       oldSA  := SA
5       wSt    := StatesOf(SA) ∪ gList[i]
6       j:=i−1
7       while (j≥1)
8         wSt     := wSt ∪ StatesOf(pList[j]) ∪ gList[j]
9         sta     := StatesOf(SA)
10        preImage :=  StrongPreImage(wSt) ∩ WeakPreImg(sta)
11        SA      := SA ∪ PruneStates(SA, preImage)
12        if (oldSA ≠ SA)
13          SA     := SA ∪ StrongPreFP(StatesOf(SA) ∪ gList[j])
14          oldSA  := SA
15          wSt  := StatesOf(SA)
16          j    := i−1
17        else
18          j—
19      pList[i] := SA
20    return pList
21
22 function mergeTables(pList, gList)
23   plan := ∅
24   goals := ∅;
25   for (i:=|pList|;i>0;i++)
26     goals := goals ∪ gList[i]
27     foreach(⟨s,a⟩ : ⟨s,a⟩ ∈ pList[i])
28       if(a∈ 𝒪 ∧ s∉ StatesOf(plan))
29         plan := plan∪⟨s,a⟩
30       if(a∈ ℐ ∧ s∉ StatesOf(plan) ∧ s∉goals)
31         plan := plan∪⟨s,a⟩
32   return plan
```

Figure 7.4: Algorithm for deriving state-action tables

1. Strong pre-imaging till fixed point for goal set G_i (i.e., $gList[i]$) in the listing) is performed (line 3);

2. When a fixed point is reached, weakening is performed (lines 7-18). Although the states for which strong solution exists are already found, we are also interested in states for which 1) a possibility (i.e., a weak plan) to reach G_i exists and 2) the guarantee of recovery (i.e., a strong plan) for goal states $\bigcup_{j=1}^{i} G_j$ exists. To find the states satisfying condition 1, we calculate weak pre-image for the current states of SA using the following routine:

$$\begin{aligned}\textsc{WeakPreImage}(S) = \ & \{\langle s,a\rangle : (a \in \mathcal{I}) \wedge (\exists (s,a,s') \in \mathcal{R} : s' \in S) \wedge \\ & (\nexists (s,a',s'') \in \mathcal{R} : a \in \mathcal{O})\} \cup \\ & \{\langle s,a\rangle : (a \in \mathcal{O}) \wedge (\exists (s,a,s') \in \mathcal{R}) \wedge \\ & (\exists (s,a',s'') \in \mathcal{R} : (a' \in \mathcal{O}) \wedge (s' \in S))\}.\end{aligned}$$

   Similarly to the StrongPreImage routine, WeakPreImage correctly handles asynchronisity but, differently from StrongPreImage, includes also states that have *at least* one uncontrollable action leading to S. To find states satisfying condition 2, strong pre-image is calculated for the states of SA joined with the states of state-action tables ($pList[j] : j \in [1, i-1]$) with lower preference (such tables are added gradually within loop of line 7 in order to guarantee the optimality of the final plan). Finally, the states satisfying the both conditions are found and added to the current state-action table (lines 10-11). After that the execution of steps 1 and 2 goes on within loop of lines 7-18 till the weakening fails;

3. When the weakening fails we store the current state=action table as $pList[i]$ and exit (line 19).

The function MERGETABLES merges state-state action tables $pList$(lines 22-33 of Fig. 7.4) into a single state-action table $plan$ representing the final plan. By processing tables in descending order (starting from preference $|pList|$), it guarantees that controllable state-action $\langle s, a \rangle$ is copied to the plan from table $pList[i] : 1 \leq j \leq |pList|$ only if no other subplan with higher preference $pList[j] : i < j \leq |pList|$ manages state s and if state s is not itself a goal state with higher preference. So the optimality of the plan is guaranteed. At the same time, the function includes all uncontrollable state-actions $\langle s, a \rangle$ into the final plan even if s is a goal state with higher preference.

```
1 function Planning(I, gList)
2   prunedGList = filter(gList)
3   pList := computeSATables(prunedGList)
4   if I∈   ⋃            StatesOf(pList[i])
        1≤i≤|pList|
5     return mergeTables(pList,prunedGList)
6   else
7     return ⊥
```

Figure 7.5: Main planning routine

Finally, the main planning routine is presented in Fig. 7.5. The planning consists in filtering the goals states (line 2), computing the state-action tables for all preferences (line 3). If the resulting state-action tables "cover" the initial state I then the state-action table for plan exists and can be obtained by merging the respective state-action tables (line 5). Otherwise, $\perp$ is returned (line 7). The plan can be obtained from the resulting state-action table using the procedure of Def. 28. It is supposed to be a solution executor for the respective composition problem.

In the following, we proof the termination, completeness and correctness of our algorithm:

Theorem 4 (Termination). *Function* PLANNING(I, G) *terminates for any planning domain D.*

Proof. Since the filtering procedure terminates as discussed above, the threat of non-terminating execution can only come form the loop of routine COMPUTESATABLES. However, in this routine it can be observed that the state-action table SA must monotonically grow up by including new and new states of D. Since the number of states in D is finite, the loop terminates and so does the whole algorithm. □

Theorem 5 (Completeness). *If function* PLANNING(I, G) *returns* $\perp$ *for some planning problem* $\langle D, I, G\rangle$ *then no plan encoding solution executor exists for* $\langle D, I, G\rangle$*.*

Proof. From the definition of the filtering algorithm it can be observed that only goal states belonging to G'_{total} can be traversed by the STS induced by the correct plan. Consequently, it can be observed that $\perp$ is returned by PLANNING(I, G) only when the initial state I is not covered by the resulting state-action table *plan*. So we conclude that the STS induced by the correct plan must be such that all its runs from the initial state reach goal states of G'_{total} in finite number of steps. For the future use, we denote with $All = \bigcup_{1 \leq i \leq n} (StatesOf(pList[i]) \cup G'_i)$ all states covered by subplans of *pList* and, consequently by plan *plan*, plus all unfiltered goal states.

We prove the theorem by contradiction. We assume that while $\perp$ is returned by the algorithm, there still exists a plan $plan'$ that encodes the solution executor. Then it covers state I (it has to provide a way to reach goals G'_{total} from I) and, $I \notin All$. If there exist uncontrollable transitions from I then there exists at least one of them that leads to states that are

not among *All*. Similarly, if there are no uncontrollable transitions for I then all its controllable transitions must lead to states that are not among *All*. The violation of the above conditions would result in the inclusion of I in *All* (see the definition of COMPUTESATABLES). Consequently, there exists a state-action pair $\langle I, a\rangle \in plan'$ such that $(I, a, s) \in \mathcal{R}$ and $s \notin All$. Since s is not a goal state it must also be covered by $plan'$. To state s' we can apply the same reasoning as to state I. Finally, we can prove that the STS induced by $plan'$ must contain an infinite run that never reaches G'_{total}, which contradicts to the definition of the solution executor. □

Theorem 6 (Correctness)**.** *If function* PLANNING(I, G) *returns* PLAN *for some planning problem* $\langle D, I, G\rangle$ *then plan encodes optimal continuously consistent solution executor for* $\langle D, I, G\rangle$*.*

Proof. We will built the proof on observing the STS Σ_E induced by the resulting state-action table *plan*. First of all, from the way COMPUTESATABLES and MERGETABLES are defined it can be observed that all states of D for which strong solution for G'_{total} exists (including G'_{total} themselves) belong to Σ_E (this can be proved by contradiction using the same reasoning as the one in Theorem 5). Through the way WEAKPREIMAGE, STRONGPREIMAGE and MERGETABLES are defined it can be seen that Σ_E correctly handles asynchornisity even if for goal states (conditions 1 and 2 of Def. 58).

The final states of Σ_E can only be goal states. Indeed, for all non-goal states there is at least one transition in Σ_E (see MERGETABLES).

Considering some state s belonging to Σ_E it can be proved that Σ_E provides a strong plan for s. Let us denote with $best(s)$ the highest preference achievable from s in D. Let us assume that $best(s) = i$ so that $1 \leq i \leq n$. From the definition of COMPUTESATABLES it follows that table $pList[i]$ will encode at least one run from s leading to G'_i. From the definition of MERGETABLES it can be seen that this run will further be included in

plan and consequently will appear in Σ_E as its run π. For any state s' to which π can deviate as a result of nondeterminism the following holds by definition: $best(s') = i' : i' \leq i$. Then we can apply the same reasoning to s': consequently table $pList[i']$ will induce a run π' of Σ_E from state s' that will lead to goal states with preference i' in finite steps and for all the deviations of it, we recursively apply the same reasoning again and again. Finally, it can be shown that Σ_E provides a strong solution for s.

The optimality of Σ_E can be proved by contradiction. Let us assume that Σ_E is not optimal and there exists Σ'_E that is optimal and it is better than Σ_E in some state s. Let $best(\Sigma'_E, s) = i$ and $worst(\Sigma'_E, s) = j$. And so Σ'_E is a strong solution for $G'_{[j,i]} = \bigcup_{k=j}^{i} G'_k$ and a weak solution for G'_i. At the same time, by observing the routine ComputeSATables and its weakening procedure we can observe that $pList[i]$ will provide for state s a weak solution for G'_i and a strong solution for $G'_{[j,i]}$ and this solution will be a part of Σ_E (see additionally mergeTables). Consequently, we conclude that $\Sigma_E \simeq^s \Sigma'_E$ which contradicts to the initial assumption.

As a result, we conclude that Σ_E is an optimal continuously consistent solution executor for $\langle D, I, G\rangle$.

□

It is worth to notice that the support of optimality for goal states is partially guaranteed by the way mergeTables joins the subplans into a final plan. In particular, a controllable action is allowed to be executed from a goal state only if it belongs to a table with higher preference. In other words, a goal state can only be left in controllable way only if we know that a goal with higher preference can potentially be achieved.

We remark that our algorithm, though based on [111], is significantly different from it. First, since we consider asynchronous planning domains including both controllable and uncontrollable transitions, the pre-imaging

primitives WeakPreImage, StrongPreImage are significantly different. Second and most importantly, the asynchronisity in goal states makes us build the plans considering maintainability ofgoals. Such plans are are likely to contain loops. This feature is provided by pre-filtering of goal states and proper adjustment of pre-imaging procedures, which now run until the fixed point and may include goal states in state-action tables.

The further details of the implementation and evaluation of the algorithm are presented in Chapter 8.

7.2 Data-Flow Requirements

In this section we introduce a prototype solution for specifying data-flow requirements in our context-aware composition framework. Our solution is based on the Datanet approach ([75] and [79] can be used for further reading), which is a graphical language for data-flow requirement in service composition. Datanet is specifically developed for the service composition techniques based on AI planning and can be easily integrated with them. In the following we give a brief overview of the Datanet language and its semantics. Using an example from the VTA scenario, we demonstrate that the standard modeling methodology of Datanet cannot implement data-flow requirements that follow our vision of dynamic context-aware composition. Finally, we show how, by means of changing the modelling methodology and introducing some pre-processing we can naturally adopt the Datanet as a data-flow requirements language in our framework. We remark, that in this case no changes are done to the language nor to its semantics.

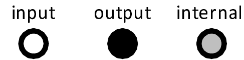
7.2.1 Datanet Overview

The way Datanet works is extremely intuitive since it explicitly specifies the flow of data within the orchestrator. In other words, Datanet specifies how the orchestrator has to handle and manipulate data pieces arriving with incoming messages in order to properly populate data for outgoing messages. The syntax of Datanet is expressive enough to specify even very complex data flow.

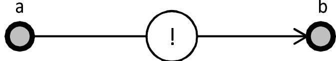
Syntax

The graphical notation of Datanet is organized such that it intuitively and explicitly shows how the data “flows” from the data parts of input messages to the data parts of output messages (the notion of data parts can be taken, e.g., from WSDL [120]). It basically consists of nodes representing variables within the orchestrator and arcs of different types representing data copying/manipulations. In the following we represent only a portion of the original language that is used in the examples of this section. The full language description can be found in [79].

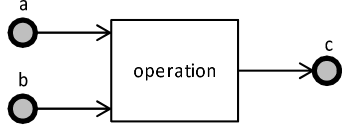
- Connection Node
  Connection nodes represent variables within the orchestrator. They can be of three types: input, output and internal. *Input* nodes represent variables where the data of input messages is stored. They become sources of information for a datanet. *Output* nodes are those variables where the data populating iuput messages is stored. They are destination for all data within a datanet. Finally, *internal* nodes are internal variables used for intermidiate storage of data “flowing” from inputs to outputs:



- IDENTITY
  Identity represents simple copying of data between two nodes. It has one starting node (it can be either input or internal node) and one finishing node (output or internal node). The graphical notation of a link `id(a)(b)` connecting nodes `a` and `b` is the following:



- OPERATION
  Operation represents processing of data from the starting nodes (they can be more than one) and sending the result to the finishing node. The graphical notation of a link `op[operation](a,b)(c)` applying operation `operation` to the information of nodes `a` and `b` and storing the result in node `c` is the following:



In the following we will use word "Datanet" for the language above and word "datanet" for specification written (drawn) in this language. The formal definition of a datanet essentially includes a number of nodes of all types (N^i, N^o, N^{int} for input, output and internal nodes respectively), a number of arcs of different types between them ($Arcs$), and a space of accepted values ($Values$). In the definition below we denote starting and finishing nodes of arc a with $in_nodes(a)$ and $out_nodes(a)$ respectively:

Definition 60 (datanet).
A data net Δ *is a tuple* $\langle N^i, N^o, N^{int}, Arcs, Values\rangle$ *where:*

- *for each* $n \in N^i$ *there exists at least one arc* $a \in Arcs$ *such that* $n \in in_nodes(a)$*;*

- *for each* $n \in N^o$ *there exists at least one arc* $a \in Arcs$ *such that* $n \in out_nodes(a)$*;*
- *for each* $n \in N^{int}$ *there exists at least one arc* $a_1 \in Arcs$ *such that* $n \in in_nodes(a_1)$ *and there exists at least one arc* $a_2 \in Arcs$ *s.t.* $n \in in_nodes(a_2)$*;*
- *for each* $a \in Arcs$*,* $in_nodes(a) \subseteq N^i \cup N^{int}$ *and* $out_nodes(a) \subseteq N^o \cup N^{int}$*.*

In the next example from the VTA scenario we show that even the limited edition of Datanet syntax presented above is enough to deal with quite complex scenarios:

Example 13 (VTA datanet)**.** In Fig. 7.6 we show a simple datanet for the services involved in the first phase of the VTA scenario, which is the initial package reservation. The datanet specifies data flow between different data parts of messages of services responsible for booking package, hotel and flight.

For example, it can be seen that the location presented in the flight and hotel requests (`fRequest.location` and `hRequest.location`) is the same and coincides with the one of the package request (`pRequest.location`) that comes from the customer. Another important example is where the costs of flight and hotel (`fOffer.cost` and `hOffer.cost` respectively) are summed up by operation `sum` in order to produce the total cost of the package (`pOffer.cost`) that will be sent to the customer.

Datanet Semantics

In order to formally define the semantics of the Datanet we encompass all possible flows of values through the nodes of datanet for given Datanet arc types. For a datanet $\Delta = \langle N^i, N^o, N^{int}, Arcs, Values\rangle$, a datanet

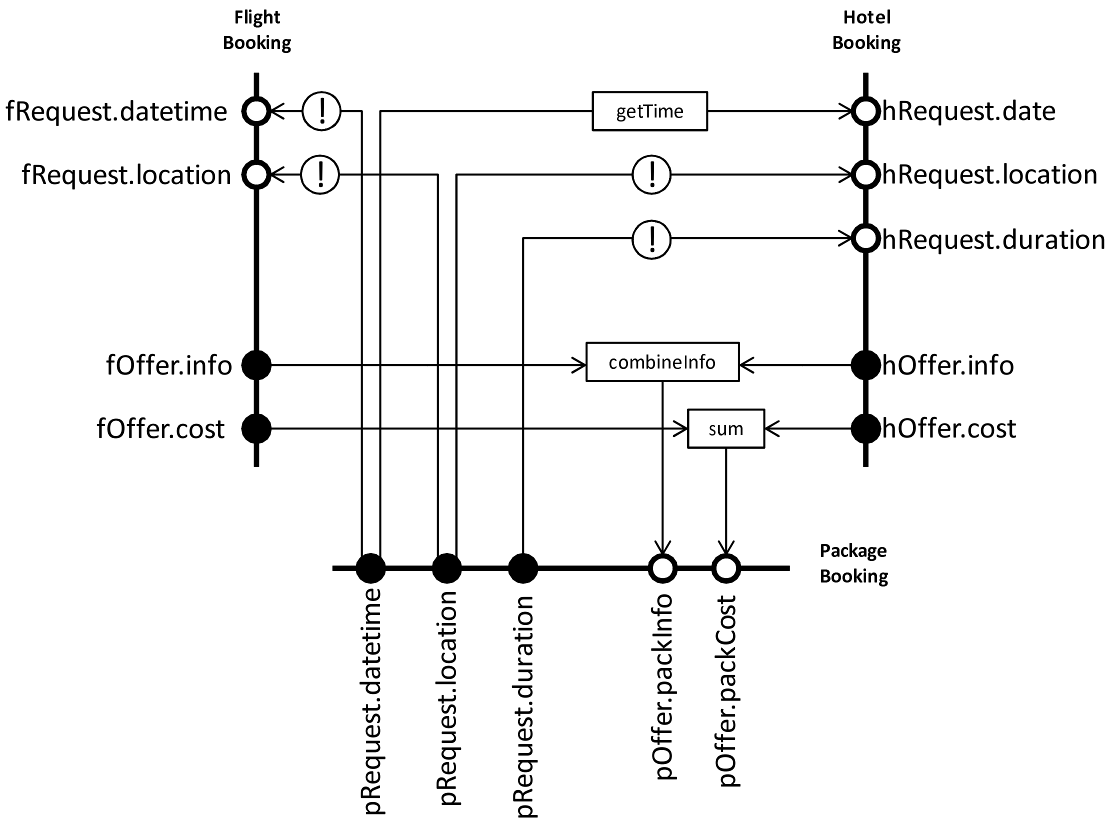


Figure 7.6: Original Datanet

event e consists in some value $v \in Values$ passing through some node $n \in N^i \cup N^o \cup N^{int}$, so that $e = \langle n, v \rangle$. Consequently, the *datanet execution* π_Δ is a sequence of events happening in the datanet. Given a set of nodes $N \in N^i \cup N^o \cup N^{int}$ we define the projection of π_Δ on N as a subsequence of π_Δ that contains only events of N (denoted $\Pi_N(\pi_\Delta)$). Using regular expressions and the notion of projection above, the Datanet semantics is formally defined as a set of *accepted executions* of a certain datanet Δ:

Definition 61 (datanet).
An execution π_Δ is an accepting execution for datanet

$$\Delta = \langle N^i, N^o, N^{int}, Arcs, Values \rangle$$

if the following holds:

- *each identity element* **`id(a)(b)`** *in* Δ

$$\Pi_{\{a,b\}}(\pi_\Delta) = \left(\sum_{v \in V} \langle a, v\rangle \cdot \langle b, v\rangle\right)^* ;$$

- *each operation element* **`op[f](a,b)(c)`** *in* Δ

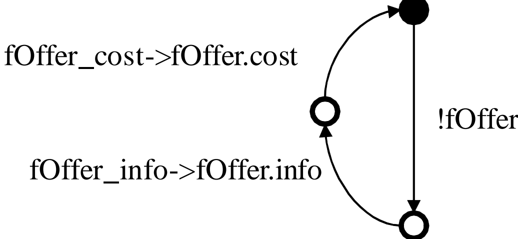
$$\Pi_{\{a,b,c\}}(\pi_\Delta) = \left(\sum_{v,w \in V} (\langle a, v\rangle \cdot \langle b, w\rangle + \langle b, w\rangle \cdot \langle a, v\rangle) \cdot \langle c, f(v, w)\rangle\right)^* .$$

The definition can be recursively applied to the datanet of arbitrary complexity.

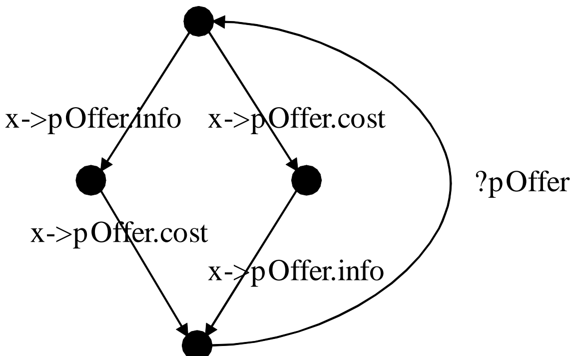
Datanet in Planning

Datanet semantics is defined by specifying all possible sequences of events that are allowed by a certain datanet. Such sets of sequence can actually be encoded using state transition systems. Indeed, the semantics of each arc a in the datanet is defined through a regular expression and can be naturally represented with a corresponding STS Σ_a. Consequently, all accepting sequences of events for the whole datanet can be derived from a system of such STSs. In order to synchronize the datanet STSs with the execution domain (e.g., the one introduced by Def. 17), we make a couple of intuitive observations. First, values "enter" input nodes when a corresponding output message of the execution domain is received, i.e., a domain's output message is always followed by events happening in its associated input nodes. Second, a domain's input message has to be always preceded by events in its associated output nodes, so that messages with unpopulated data parts are never sent to the execution domain.

In order to illustrate these ideas we will use examples related to Example 13. In them, a transition labeled with `a->b` stands for an internal action copying data from node `a` to node `b`. Node `x` is used as a placeholder for any node, so that the labeling `x->a` means any action copying data to node `a`. Moreover, each STS has an accepting state (black dot in the diagrams), in which it has to be in the end of the accepting execution. For each output message of the domain (i.e., input message for the orchestrator) we require that every time this message arrived all the associated input nodes of the datanet are populated with the message data. So, for the output message `!fOffer` and its two associated input nodes `fOffer.cost` and `fOffer.info` the following STS can be built (we use `fOffer_cost` and `fOffer_info` to denote internal data fields of the message):



Similarly, we want to ensure that a message is sent by an orchestrator only if it is completely populated with data. For an input message, `!pOffer` associated with output nodes `pOffer.cost` and `pOffer.info` such controlling STS would look as follows:

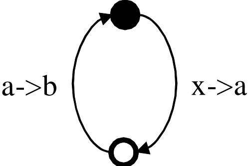


Such STS ensures that the message will be sent only after the both part

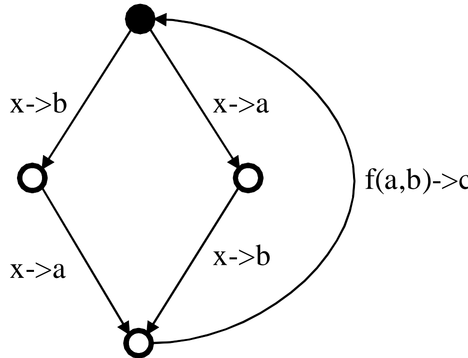
are populated with data.

Finally, we encode the arc between nodes using the following constructions:

- the identity arc `id(a)(b)` is modelled with STS:



- the operation arc `op[f](a,b)(c)` is modelled with STS:



The resulting datanet controller is an STS Σ_Δ that is the synchronous product of all STSs corresponding to all input and output nodes and all arcs in the datanet.

In a few words, the composition in the presence of datanet is performed almost in the same way as the composition of Chapter 6. The difference is that the final planning domain is obtained as a synchronous product of a context-aware domain Σ_{CF} (Def. 27) and a datanet controller Σ_Δ. Moreover, it is required in the planning problem that the datanet controller terminates in one of its accepting states. In [75] it is proved that the resulting plan reflects the data handling that guarantees the behaviour that respects the semantics of Datanet. In the final plan, along with input and output actions datanet-related actions exist. In the back translation

of the plan into an executable process, such actions have to be interpreted as copying or applying a function to data and then copying. For instance, in WS-BPEL such actions can be fully realized by the ASSIGN activity.

7.2.2 Context-Aware Datanet

Coming back to Example 13, it can be easily concluded that the Datanet approach as it is does not follow the principles of dynamic and context-aware composition showed in Fig. 6.6. As a result, it cannot be directly integrated in our framework for service/fragment composition. Indeed, while in our approach, control-flow requirements are specified on abstract level and can be grounded on any set of properly annotated services, Datanet-based requirements are implementation-dependent and have to be provided for concrete service implementations. In other words, every time we experience a run-time need for composition, we have to manually specify the respective datanet for a given set of services/fragments, which is not what we call dynamic composition. However, the concept of context properties can be easily integrated into the Datanet approach in order to make it dynamic and context-aware.

Our architecture for adopting the Datanet in our dynamic context-aware composition framework is shown in Fig. 7.7. The central idea is to introduce context properties to a datanet by associating certain data fields to context properties and representing them in a datanet with internal nodes (by analogy with service ports, we sometimes call the collection of such internal nodes the *context port*). For example, the Flight Ticket context property can be associated with data fields such as flight time, flight origin, flight destination, flight ticket cost etc. As a result, the datanet will contain input and output nodes corresponding to the ports of services and internal nodes corresponding to the data fields of context properties.

At this point, the whole datanet can be split into two conceptually dif-

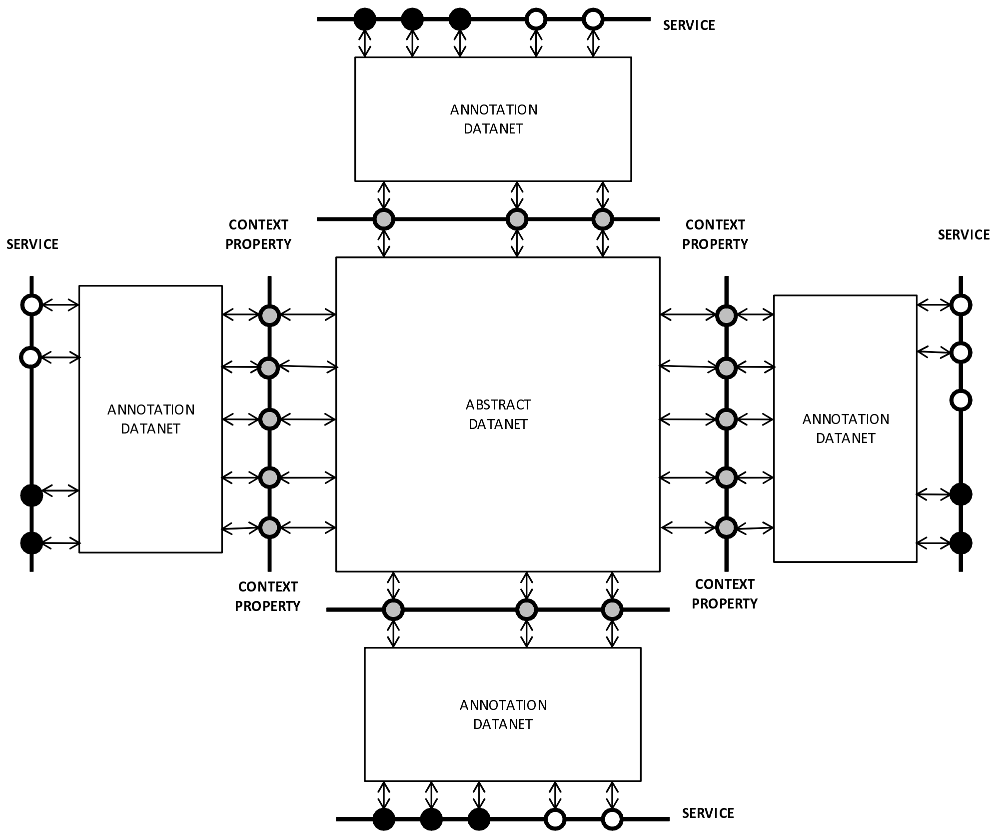


Figure 7.7: Architecture of context-based Datanet

ferent parts. One part determines data flow between the nodes of service ports and the nodes of context port. It is called *annotation datanet*. Another part specifies how data flows between the nodes of context port and is called *abstract datanet*. The relation between these two parts of datanet is essentially the same as between the context-based service annotations and context-based composition requirements. Abstract datanet specifies relations between data fields of different properties and does not depend on service implementations. We can say that it reflects conceptual dataflow requirements for a certain type of composition and can be specified at design time even without knowing which services will be composed. Similarly, annotation datanet can be perceived as a new type of service annotation indicating how data parts of service messages correlate with data

fields in the context model. In this regard, they look extremely similar to the service annotation for control flow introduced in Chapters 4 and 6

From the modeling perspective, abstract datanet is supposed to be specified by the designer of the composition and requires only the knowledge about the application domain. The annotation datanet, being a part of annotation, should rather be specified by a service provider who wants its service to be used within the composition. As a result, the modeling effort of the datanet can be distibuted among the participants of the composition. Once a need for composition emerges at run time and appropriate services are selected, the abstract datanet and the annotation datanets of participants can be joined together to make up the complete datanet as it is given in Fig. 7.7. Considering the whole picture, it can be noticed that in the complete datanet service-to-service data flows is organized: now the data values traverse the annotation datanet of the source service, then pass through the abstract datanet and by means of links in the annotation datanet of the destination service can populate its nodes. Using the terminology of Chapter 6 we can say that *the abstract datanet is defined in separation from services, but using the annotation datanets of services can always be grounded on them.*

Example 14 (Context-Aware Datanet)**.** In order to demonstrate context-aware Datanet in action, in Fig. 7.8 we propose a variant of a datanet presented in Example 13 when modelled using context properties. In this datanet, in addition to service ports, there exist three groups of internal nodes representing data fields of the three context properties of the VTA scenario. For example, the data fields of the hotel reservation reflect such aspects as original request information, check-in/check-out date of the reservation, its cost etc. The central part of the datanet, which is the abstract datanet, installs conceptual data relations between context properties. For instance, the links between the cost fields of all three properties

indicate that "the cost of the package equals the sum of the costs of constituent flight ticket and hotel reservation". (By the way, this requirement can be flexibly changed by only changing the operation `sum` and without affecting service providers and their annotation datanets). In this example, one more interesting observation can be made: each service provider and the composition owner may use their own data formats, which can be radically different. For example, the package-related service and the context model use starting and finishing date of the trip in their internal models. At the same time, the hotel provider uses the starting data and duration. In this case, the interoperation is guaranteed by the fact that each provider specifies its annotation datanet, which also plays the role of data mediator and that supports compatibility of different data models. In our example, such compatibility is realized by properly defined conversion functions `getDateH`, `getDurationH`, `getDateInH` and `getDateOutH`, which are defined by the hotel service provider.

Our preliminary estimation of the context-aware Datanet approach suggests that no additional effort is needed for the datanet to be properly integrated to the planning problem and the techniques of [75], [79] can be used unchanged.

7.3 Discussion

The goal of this chapter was to show that our context-based service model can actually be used to address various aspects of service composition. In particular, we showed that it allows for rich abstract control-flow and data-flow requirements, both following the same methodology where 1) conceptual part of requirements is expressed on abstract level separately from actual service implementations and 2) requirements grounding on particular service implementation is provided by service annotations. In this case,

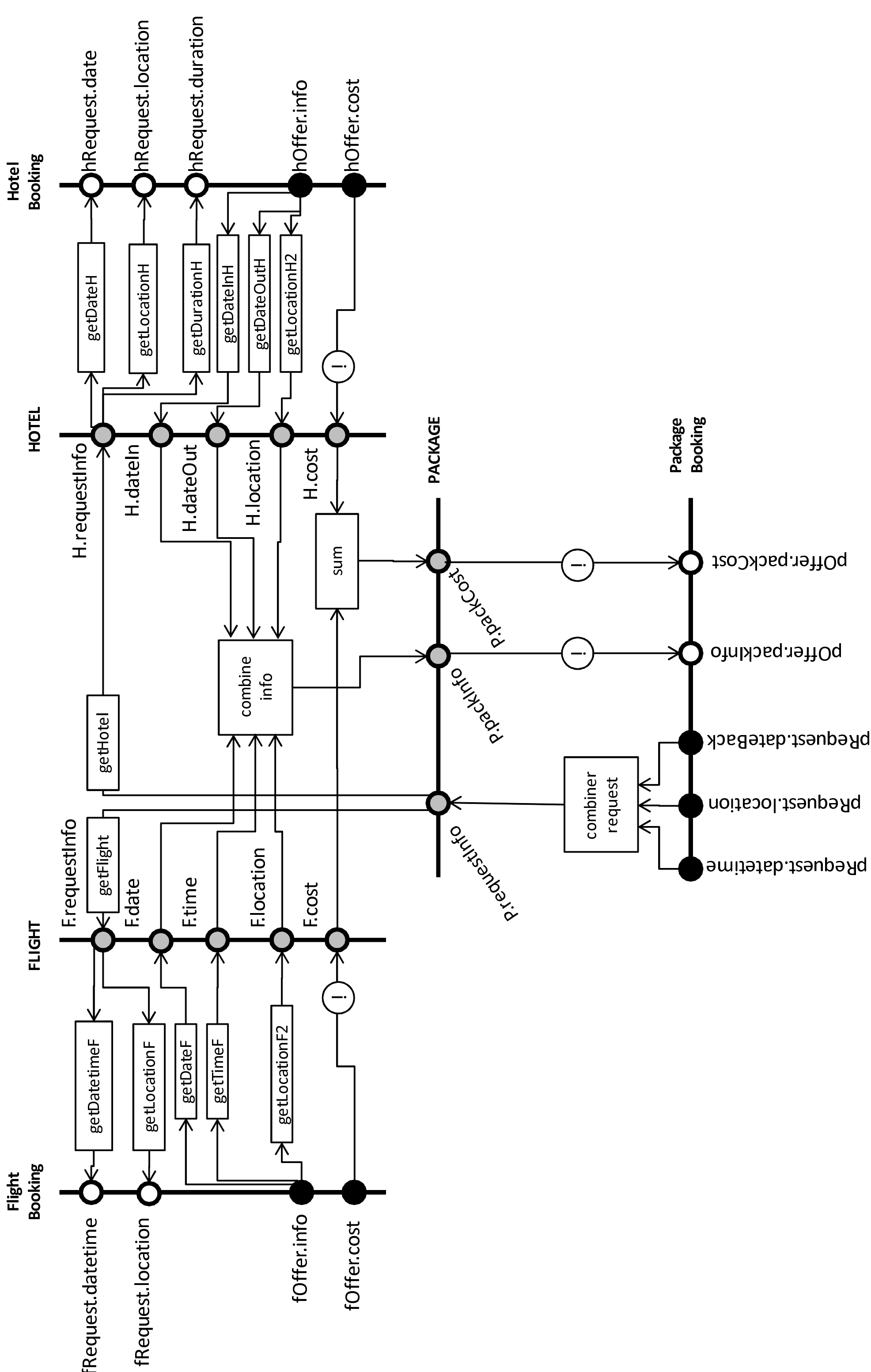


Figure 7.8: Context-based datanet for virtual travel agency scenario

the modeling effort can be distributed among the participating parties. Namely, the owner of the application (composition) is supposed to provide the context model and the conceptual requirements while all the partners wanting their service to be consumed within the application have to provide their proper annotations. As a result, using such requirements methodology, it is possible to create customizable applications where, while the concept of the application is predefined (e.g., "application for buying travel packages") and the respective conceptual requirements remain unchanged, the choice of services to be used within this concept can be entrusted to the end user and can be made at run time with consequent automatic composition. This brings us to the concept of user-centric service composition. One of the key issues to be addressed to make our approach ready for user-centric applications is the ability to generate the composition interface and protocol automatically. The first step in this direction has laready been made and the prototype solution to this problem is available in [60].

We also want to remark that our control-flow requirements and the respective planning algorithm bring up a novel aspect of *reactive requirements* where certain composition goals have to be achieved only as a reaction to context events. Moreover, we expect that our modeling methodology for encoding requirements in planning can be used for even more complex requirement constructions and languages.

Chapter 8

Implementation and Evaluation

In this chapter we present the implementation and evaluation of our approaches to service composition and process adaptation. In particular, we discuss 1) the implementation of the context-based service composer introduced in Chapters 4, 6 and 7 (this includes the implementation of our extensions to the planning algorithm proposed in Section 7.1), and 2) a demonstration platform *ASTRO-CAptEvo* for dynamic process adaptation that realizes the ideas introduced in Chapters 3, 4 and 5 and adopts the aforementioned service composition approach. The evaluation of the two tools considers both "qualitative" and "quantitative" aspect. By qualitative aspect we mean the ability of the tools to successfully solve the problems posed by the respective motivating examples. By quantitative aspect we mainly mean the performance of the planning algorithm and its ability to produce solutions in realistic setting in affordable time.

8.1 Composer

8.1.1 Implementation

All the planning algorithms presented in this dissertation essentially rely on the planning-as-model-checking technique ([32, 15]). In this technique,

symbolic representation of states of a planning domain based on Binary Decision Diagrams (BDDs [21]) is used to effectively encode and manipulate sets of states (symbolic model checking [81, 31]).

While various (e.g., XML) specifications of fragments/services and context properties can be used at design phase, all the STSs at the planning phase (i.e., grounded context properties, fragments/services and grounded control STSs) are uniformly encoded using the SMV (Symbolic Model Verifier) language. SMV is a rich modeling language providing diverse constructs and mechanisms for compactly encoding state transition systems. In particular, states and actions are encoded by means of special state and action variables, and formulas are used to encode transitions between states of variables. SMV is mostly exploited by symbolic model checking systems. One of such systems (NuSMV [31]) was originally adopted by the planning-as-model-checking algorithm of [18] (which is the core of our planning algorithms) and so SMV became the language for encoding a planning domain in our approach.

Luckily enough, SMV allows for defining a planning domain through a number of smaller STSs with possibly overlapping sets of actions. When such model is eventually represented in memory with BDDs, it generally corresponds to a synchronous product of these smaller STSs (the functionality for reading models into memory is provided by core NuSMV libraries). As such, the transformation of the composition problem into the planning problem generally requires 1) transformation of service/fragment specifications into STSs, 2) grounding of context properties and 3) grounding of elementary control STSs. After that, the STSs obtained can be straightforwardly "written down" in SMV and be passed to the algorithm. Our experience suggests that the transformation of the composition problem into the planning problem requires much less computational effort than the planning itself and does not affect significantly the overall performance

of the approach.

The SMV model is represented in memory as a complex BDD encoding all transitions between state variables that are possible in the domain. The fundamental primitives of the original planning algorithms of [18] and [111], such as states pruning and pre-imaging, essentially involve various manipulations over BDDs. For example, given a BDD encoding current states of the state action table and a BDD encoding the planning domain, the strong backward pre-image function finds a BDD encoding the state actions of the pre-image by means of standard logic operations over BDDs.

The algorithm used in Chapters 4 and 6 was essentially the algorithm of [18]. The algorithm of Chapter 7, though inspired by [111], is significantly different from it and required some important modifications to the original code. First of all, we needed to implement core pre-imaging routines WeakPreImage, StrongPreImage for the case of asynchronous domain. Second, we had to implement the preliminary filtering of the goal states of the original planning problem (see Fig. 7.3), which consisted in executing a strong planning algorithm for the whole set of goal states so that recoverable goal states were identified. Third, the implementation of the algorithm for deriving subplans for different priorities (Fig. 7.4) had to be modified to use new routines for pre-imaging. Finally, some minor changes to the merging procedure for subplans had to be made.

One important optimization, that we used to improve the performance of planning was based on the observation that Lemma 3 can be proved in the same way not only for goal states but also for all states of the domain. In other words, a state of the planning domain can appear in the final plan only if there exists a strong solution from this state for a filtered set of goals G'_{total}. As such, the filtering procedure can be used not only to filter off the "bad" goal states but also to shrink the domain to those states that can potentially appear in the plan. From the performance

perspective, although the filtering procedure imposes some overhead to the algorithm, it lets us decrease the size of the resulting domain. The further n runs of the planning algorithm within the COMPUTESATABLES routine can be performed for this smaller domain and, consequently, be accomplished faster.

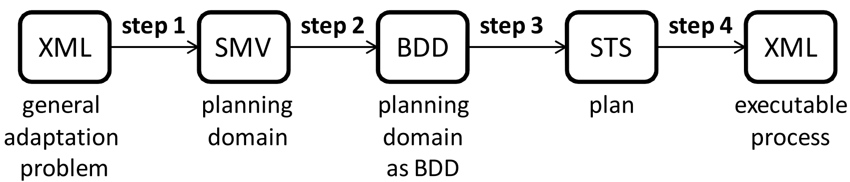


Figure 8.1: Planning procedure overview

To sum it up, in Fig. 8.1 we give a general overview of the planning procedure used by the algorithm of Chapter 4 (the algorithm of Chapter 6 generally works in the same way). The general adaptation problem is encoded using our custom XML-based format. The definitions of the context properties, fragments/services and general adaptation problem in XML files completely corresponds to the respective formal definitions of Chapters 4 and 5. Step 1 converts general adaptation problem into an SMV specification of the planning domain. This step follows the transformation rules described in Sections 4.2 and 4.4. Step 2 reads SMV specification to memory is form of BDD. Step 3 is realized by the planning algorithm of Section 4.5, which produces a plan in form of STS. Finally, Step 4 converts the plan obtained into an executable process using the transformations that are opposite to those used in Step 1.

8.1.2 Evaluation

All the experiments for evaluating the composer were carried out on a 2.6 GHz dual core machine with 4Gb of memory running Linux. The qualita-

tive evaluation has been done for the Virtual Travel Agency scenario (Section 6.1), while the quantitative evaluation exploited simple and scalable scenarios to measure performance in connection with complexity factors.

VTA Solution

Our experiment with the VTA scenario included creating all the necessary specifications of annotated services, context properties and composition requirements as shown in Chapter 6 and running the composition algorithm to produce an orchestrating plan.

As a result, we received a plan in Fig. 8.2 that orchestrates the five component services of the scenario in order to continuously satisfy the respective control-flow requirements. While considering the scenario in connection with the components (Fig. 6.9), we pay attention to a few important aspects. First of all, the plan encodes a runnable process (no more than one controllable action per state) and properly handles asynchronous behaviour of components (it never executes controllable actions in the presence of uncontrollable ones and accounts for all uncontrollable actions for a given state). The plan is a strong solution for all its states, i.e., from each state a goal state is reachable in finite number of execution steps.

From the high-level structure of the process (demarcated using bold dotted lines), it can be observed that its behaviour is generally inspired by control-flow requirement expressions. From the initial state, the process tries to perform the transactional booking of the package (expression 1 in Example 11 and part A in the figure), which includes booking of flight and hotel, correctly taking into account possible non-deterministic outcomes of the component services, and creating a travel offer upon successful reservations . Once the booking is successful, the process continuously handles flight delays and cancellations. In the first case (expression 2 in Example 11 and part B in the figure), it tries to modify the hotel reservation: if the

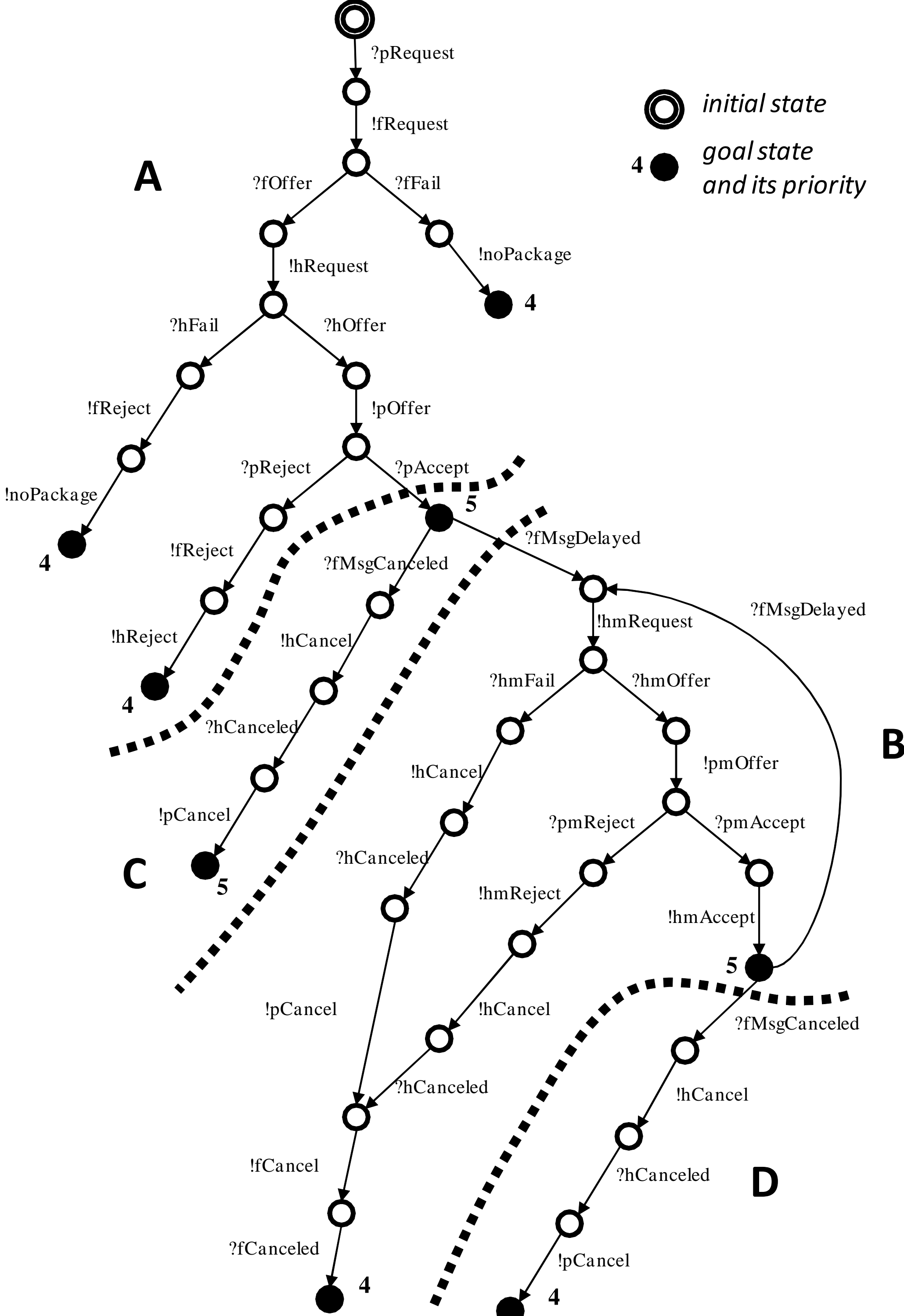


Figure 8.2: Solution for virtual travel agency scenario

hotel agrees and the user accepts the new offer, the process comes back to the goal state and is ready for new flight notifications. Otherwise, the process cancels all reservations and terminates. In the second case (expression 3 in Example 11 and parts C and D in the figure), the process cancels all reservations and terminates.

In the process, it is easy to identify the goal states. Consequently, all the subprocesses connecting goal states are associated to certain situations within the scenario. Our special interest goes to goal states with highest priority that are in the middle of the process. It is clear that the "conventional" planner would prefer to finalize the execution there. However, uncontrollable actions are available from these states and, according to our needs, this uncontrollable situations have to be (and actually are) accounted by the plan.

Finally, we can see that the plan does not contain "bad" loops: the only loop in the plan traverses a goal state which ensures that every run of the plan reaches goal state in finite number of steps. In general, the plan is quite complex, with several branching points, and much more complex than any of the involved components. We may conclude the task of building such composition manually is far from trivial. The composed service was generated in about 35 seconds; given the complexity of the task, we consider it as an important evidence of practical applicability of our approach.

Performance

The profound evaluation of the planning algorithm used in Chapters 4 and 6 is provided in [18]. The general conclusion about the performance of the plan search is that "the performance of synthesis appears to degrade sub-exponentially with the size of the components; and in vast majority of cases, it degrades polynomially with the number of components". It is also noticed that the performance does not depend significantly on the

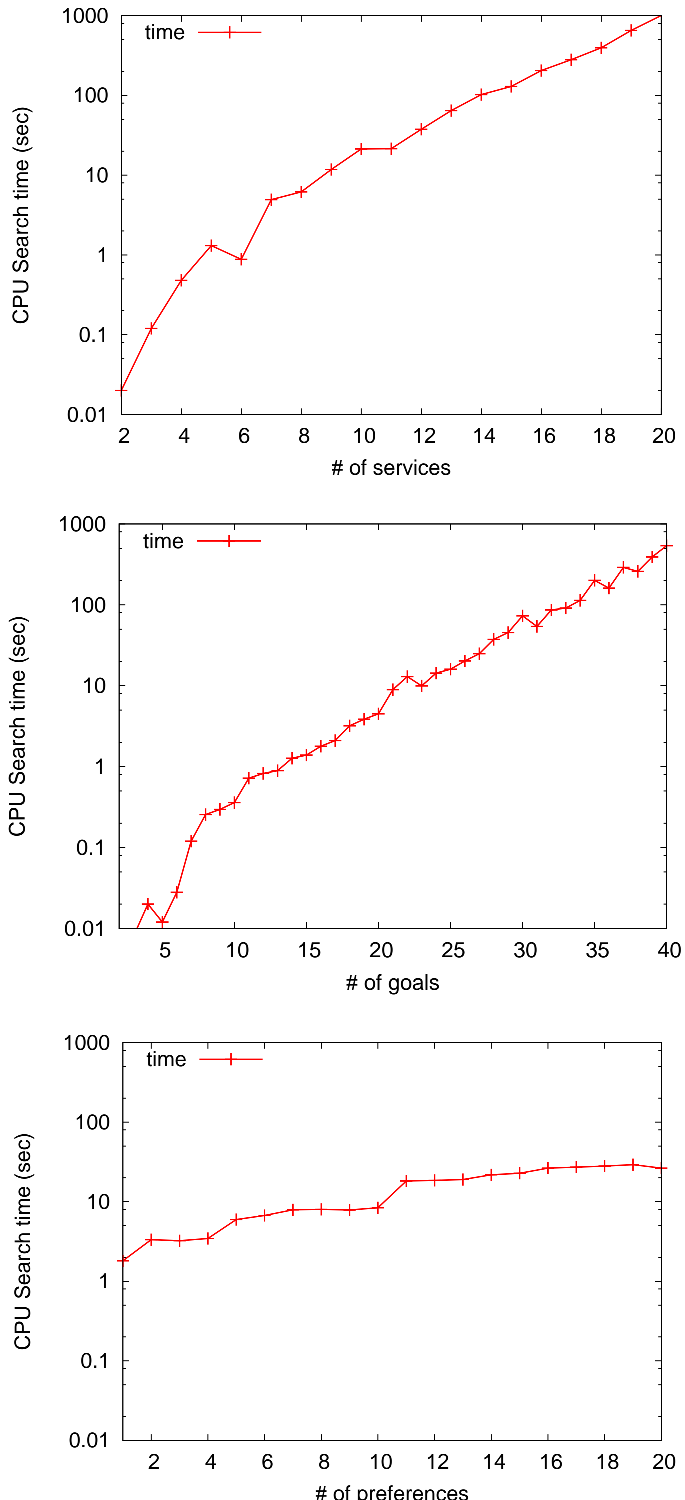


Figure 8.3: Performance scalability charts

nondeterminism in components.

In our algorithm for continuous composition presented in Chapter 6 we expected similar results. We did not evaluate the time on synthesizing the planning domain form the composition problem specification since in our experiments it was significantly smaller that the time on plan search (normally, by the order of magnitude of at least 2). We measured performance depending on the number of services participated, the number of composition constraints (reactive goals) and the number of preferences. For each set of experiments we used simple scalable scenarios.

In the first set, we evaluated the scalability of coordinating an increasing number of services. The scenario involved an "inviting" service and n "guest" services. The inviting service sent an invitation, and then kept listening to responses; vice versa, a guest was activated by an invitation, and then could continuously send updates on its decision. Our goal was to propagate invitation to all guests, and then to keep I continuously updated on the responses of each guest. Our results for this set of experiments are shown in Fig. 8.3, top. It can be seen that for smaller number of services the performance scales up polynomially, however after some point it turns out to be exponential. The most reasonable explanation for that is based on the implementation details of the BDD library used: big domains are much more memory-demanding and for them the garbage collection and data re-arrangement mechanisms may take considerable time to keep the memory consumption within certain limits. However, even in this case the algorithm is good enough to solve quite large problems in reasonable time (e.g., 10 seconds for problems containing up to 9 services).

In the second scenario, we evaluated the scalability with respect to the number of reactive expressions (goals) in the requirements. For this purpose, we considered a master and a slave service: the master continuously produced a command out of a set of n possible ones, and the slave was

awaiting for commands to be executed. To keep the two services continuously aligned, we used a set of n reactive requirements. The results are shown in Fig. 8.3, center. We can see that the results are essentially similar to the previous once. It can be easily explained by the fact that requirements are encoded with STSs and, from the planning perspective, the growth in the number of requirements is similar to the growth in the number of services: both result in comparable growth of the planning domain.

Finally, we correlated performance with the number of preferences in the goal. We did so by running the example with a set of services simulating a robot scenario where each 'robot' service could be commanded to guard a door, but might then autonomously break or decide to recharge, so becoming (temporarily or finally) unavailable. Considering 20 robots and 20 doors, we tested it with goals that used $n = 1, \ldots, 20$ preferences to express that we intended to keep n doors guarded, but whenever this could not be guaranteed, as many as possible. As we see from Fig. 8.3, right, the performance essentially scales linearly with the number of preferences. Since our algorithm calculates a subplan for each preference, the linear dependency shows that the time on calculating a single subplan is more or less constant and does not depend on the total number of subplans to be calculated within a given problem.

8.2 ASTRO-CAptEvo

This section describes the ASTRO-CAptEvo platform, which is a demonstrator of our approach to dynamic adaptation of fragment-based business processes based on the CLS scenario. The core of the platform is the algorithm for context-aware fragment composition via planning presented in Chapter 4.

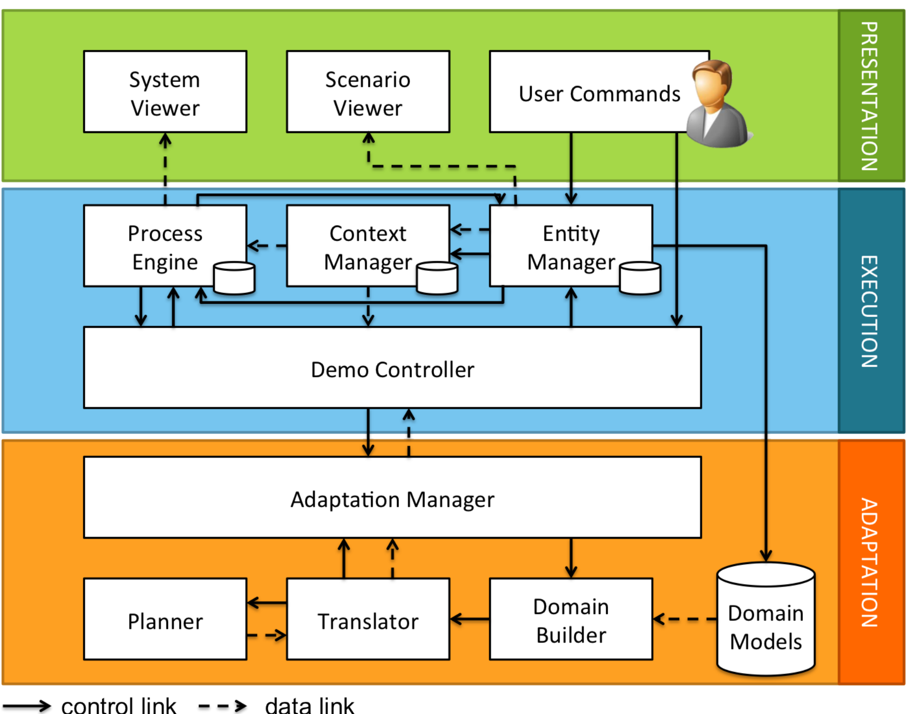


Figure 8.4: ASTRO-CAptEvo architecture

8.2.1 Implementation

The general architecture of the platform is shown in Fig. 8.4. All the elements of the architecture are split into three layers of abstraction communicating to each other. The *presentation* layer makes it possible for the user to receive complete visual information about all the aspects of system evolution and actively affect the course of the scenario, e.g., by firing exogenous events in order to simulate various extraordinary situations and see certain types of process adaptation in action. The *execution* layer contains all the components for executing adaptable business processes and for simulating the respective execution environment (i.e., entities collaborating with each other within the scenario). The *adaptation* layer implements all tasks related to process adaptation, from managing adaptation strategies to constructing an executable process.

In the rest of this section we overview each of the three layers and describe how they interoperate.

Execution layer

The Execution layer is in charge of 1) simulating the application domain consisting of a number of collaborating entities, 2) executing fragment-based process instances, 3) detecting execution problems and triggering adaptation by passing necessary information to the adaptation layer, and 4) adapting process instances according to the solution produced by the adaptation layer.

The ENTITY MANAGER controls all active entities within the scenario (e.g., ships, cars, tracks, storage managers, etc.), simulates their behaviour and provide the presentation layer with respective information in order to visualize the application domain. In fact the Entity Manager simulates the real world in which the CLS application operates. It also implements all fragment activities and simulates their execution and effects.

The instantiation of a new entity can be performed: 1) within the application initialization 2) as a result of fragment execution (e.g., ship unloading creates new car entities), or 3) as a result of user's command (new ship can be created in this way). When the Entity Manager creates a new entity, it deploys the entity process to the Process Engine, adds corresponding context properties to the context model in the Context Manager and puts all the entity-related specifications (such as fragment models and context property models) to the Domain Models repository for the future use by fragment composition engine. When the entity "exits" the scenario the opposite procedures are used.

One of the major functions of the Entity Manager is to actualize the execution of fragment activities upon the commands from the Process Engine and to simulate respective behaviour of entities. Once actions are

executed the synchronization information is sent back to the Process Engine. While simulating the evolution of the application domain, the Entity Manager keeps on updating the Context Manager with the actual state of the domain so that the context model always contains actual information about the current state of the context. Similarly, the Scenario Viewer is constantly updated in order to visualize the actual state of the virtual world. Finally, the Entity Manager can also accept user commands triggering various events and situations (e.g., exogenous events).

The Context Manager stores the system context as it is defined by Definition 2 (i.e., a set of context properties of all active entities) and constantly synchronizes its current state using the synchronization information coming from the Entity Manager. The system context is a simplified view of the real world (in this case, of the virtual world modeled by the Entity Manager) that reflects the information about the world that is of importance for process execution. The current state of the context is used by the Process Engine to check activity preconditions and by the adaptation layer for adaptation-related tasks such as adaptation strategy management and fragment composition.

The Process Engine is essentially a conventional process engine that is extended with some adaptation-related tools. The Process Engines executes all the process instances within the demonstrator, both the core processes and the adaptation ones. The extensions compared to the conventional process engines are the following:

- *Consistency checking.* The Process Engine detects conflicting situations that require process adaptation. Once a conflicting situation is detected, the information about it is passed to the Demo Controller and is further directed to the adaptation manager. The consistency checking relies on fragment specifications and annotations and on the run-time information about the world (current context and fragments

availability);

- *Execution suspension.* The Process Engine has to be able to suspend the execution of the process in case inconsistency is detected and to resume it after the adaptation is applied;
- *Adaptation integration.* The Process Engine has to provide facilities to implement the hierarchical adaptable processes as described in Chapters 3 and 5 . This basically includes the aforementioned possibility to deliberately suspend process execution with further resumption, and to perform "jumps" in suspended process instances.

The DEMO CONTROLLER provides the integration of the execution layer and the adaptation layer. When execution inconsistency is reported by the Process Engine, the Demo Controller aggregates the information needed by the adaptation layer in order to resolve the problem (e.g., current context state, set of available fragments, type of violation, status of the conflicting process instance etc.) and sends the complete problem description to the adaptation layer. Once the solution to the problem is provided by the adaptation layer, the Demo Controller supervises the adaptation procedure respecting the adaptation strategy chosen by the adaptation layer. This may be done by deploying necessary adaptation process(es), by changing the current states of process instances and by suspending/resuming execution if certain process instances.

Adaptation layer

The adaptation layer is responsible for producing solutions to adaptation problems. In particular, it 1) decides on the adaptation strategy to be used, 2) transforms a general adaptation problem into a planning problems 3) builds an adaptation plan, 4) converts the plan into an executable process

and returns it to the execution layer together with instructions for further integration.

The operation of the Adaptation layer is coordinated by the ADAPTATION MANAGER. Once the Adaptation Manager is notified about an execution problem, it decides on the adaptation strategy to be used (in our case, the rules of Section 3.4 are applied). The information on the adaptation goal and the status of the execution environment (i.e., set of available fragments, current context) is then passed to the Domain Builder and the fragment composition "chain" including the Domain Builder, the Translator and the Planner is activated. Once the solution process is reported by the Translator the Adaptation Manager sends the result to the execution layer together with the instruction for its integration. In principle, before producing the result the Adaptation Manager may need to run composition multiple times. For instance, if we use complex strategy management and if the first strategy cannot be used (plan is not found for it), the Adaptation Manager may switch to the strategy with lower preference and run the fragment composition again to find a solution for this trategy.

The DOMAIN BUILDER is supposed to specify a general adaptation problem (see Def. 32). For that purpose, the Domain Builder uses the run-time information obtained from the Adaptation Manager and extracts necessary fragment and context property models from the DOMAIN MODELS repository. Taking into account the current context and the adaptation goal, the *Domain Builder* uses basic optimizations to simplify the general adaptation problem. For instance, it can eliminate fragments that cannot be used in the current context or for the current goal, or prune all context states that are not reachable with the current set of fragments. Fragment annotations are used to enable this kind of reasoning. With such optimizations the size of the planning domain is supposed to further be reduced, which, in turn, significantly saves the time spent on planning.

The TRANSLATOR provides two-way translation. First, it translates a general adaptation problem specified in XML and received from the Domain Builder into a planning problem specified in SMV ([82]) such that it can be processed by the Planner. Second, it provides back translation of the plan generated by the Planner into an executable APFL process, which is passed to the Adaptation Manager. The translations basically follow the the transformation rules discussed in Sections 4.2 and 4.4.

The PLANNER is based on our composition-as-planning approach proposed in Chapter 4. The adaptation layer supports parallel runs of the planning algorithm in order to improve the overall performance of the platform.

Presentation layer

The presentation layer provides a detailed live view of the details of the operation of the execution and adaptation layers. It also gives certain control over the scenario evolution to the user and lets him model different critical situation to test the adaptation techniques.

The SCENARIO VIEWER provide graphical representation of the CLS scenario that is constantly synchronized with the simulation data of the execution layer. Visually, it is a map of the Bremen harbour where all the facilities (e.g., storage areas, treatment areas, gates, roads etc.) and all the entities participating in the scenario (e.g., cars, trucks ships, facility managers) are depicted at their current location and in their current status. With the Scenario Viewer the user can intuitively follow the progress of the scenario (Fig. 8.5).

The Scenario Viewer is integrated with the SYSTEM VIEWER, which gives the insight into the objects operated by the execution and adaptation layers. The System Viewer is represented by a number of windows covering various aspects of the system. The two main of them are the *Process Viewer*

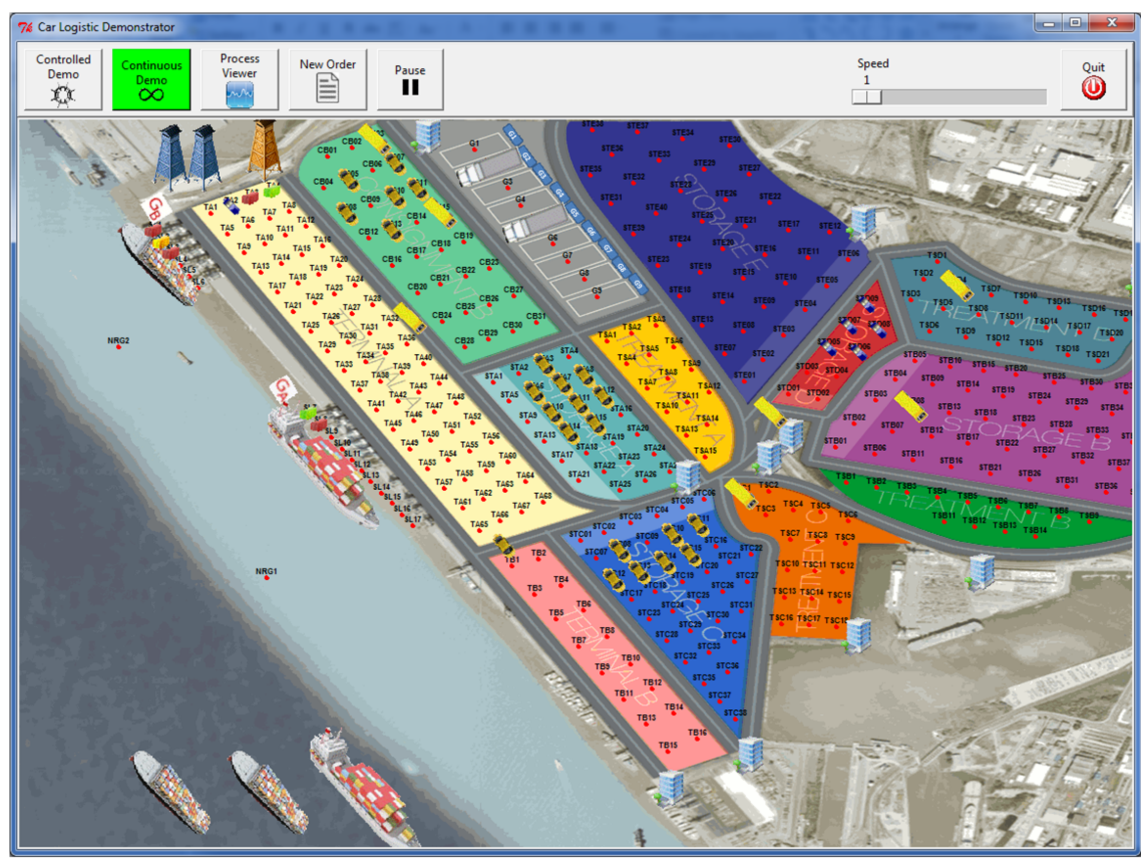


Figure 8.5: Scenario Viewer

and the *Adaptation Viewer*. The *Process Viewer* (Fig. 8.6) gives access to full details of a certain process instance including:

- the process model with the execution progress indicated (upper left part);
- the execution history including all adaptations applied, e.g., all refinements and local adaptation that have been used within the instance (lower part);
- execution context comprising all relevant context properties and their current states (upper right part).

For each adaptation case within a process instance, the *Adaptation*

Viewer can be called (Fig. 8.7). It provides details on how a certain adaptation problem has been resolved, including:

- the general adaptation problem featuring a set of fragments participating in the composition and the portion of context that is relevant for the process instance under adaptation (tab ADAPTATION PROBLEM);
- the result of fragment selection based on the pruning of useless fragments (tab SERVICE SELECTION);
- planning domain expressed in the SMV language (tab PLANNING DOMAIN);
- planning algorithm timing (lower part);
- the resulting APFL adaptation process (tab APFL PROCESS).

The USER COMMANDS are used to control the simulation running in the execution and adaptation layers and make it possible to lead the scenario to extraordinary situations where the capabilities of process adaptation can be demonstrated. The simulation can be controlled by pausing and resuming the execution or by increasing and decreasing the execution speed. The user can affect the scenario i) by triggering exogenous events (e.g., cause damages to cars, order cars stored at the storage areas and cause unavailability of storage areas) and ii) by creating new entities (e.g., new ships loaded with customizable number/types of cars).

The CAptEvo platform has been presented at Service Cup 2012 competition [105] and can be freely downloaded from the ASTRO project web site at `http://www.astroproject.org/captevo.php`. There, the reader can also find a video tutorial explaining how to run and use the demo.

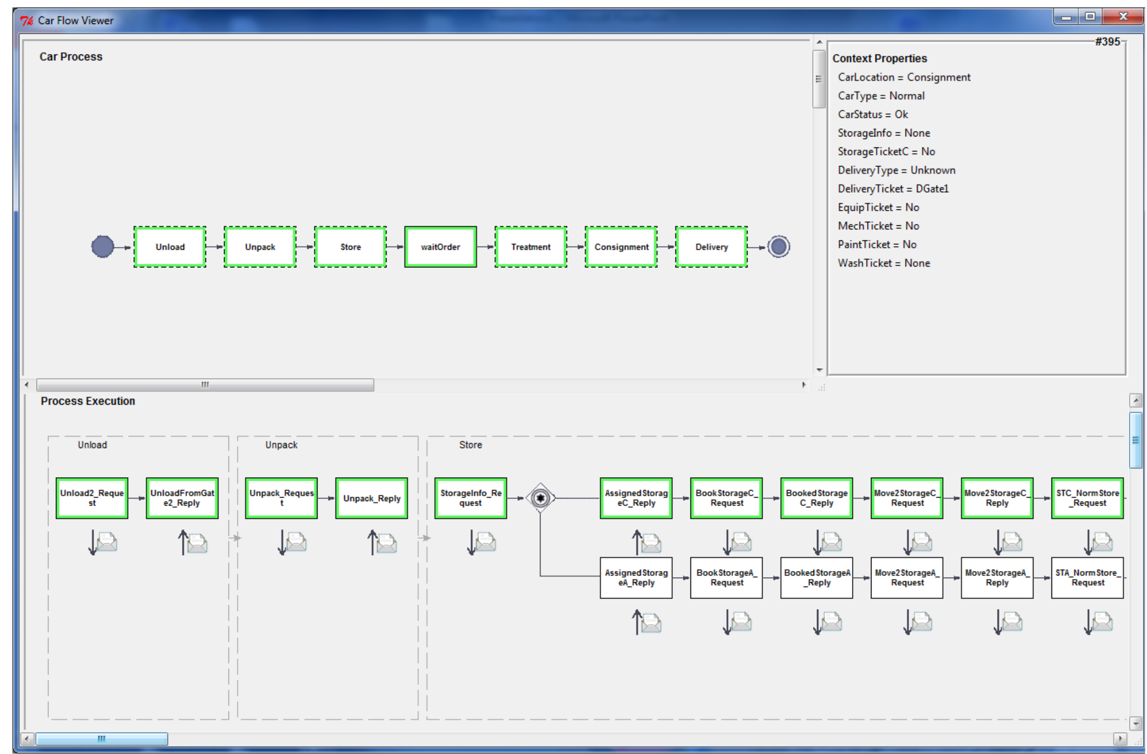


Figure 8.6: Process Viewer

While running the demo, we encourage the reader to pay attention to the evaluation cases that will be described in the next section.

8.2.2 Evaluation

The evaluation of our adaptation approach using the ASTRO-CAptEvo platform included three main parts. First, we made a qualitative evaluation to demonstrate how the challenges posed by the motivating example of Section 3.1 could be addressed. Second, we assessed the modeling effort required in order to enable the adaptation (i.e., time on context modeling, fragment annotations etc.). Third, we made some performance evaluations of the composition approach to see how it scaled up for the adaptation-related tasks.

All the evaluation was carried out using a dual-core CPU running at

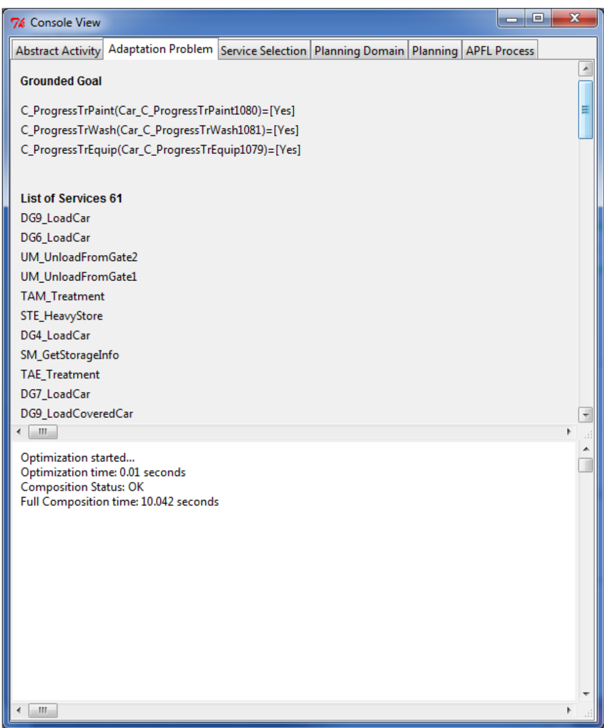


Figure 8.7: Adaptation Viewer

2.8GHz, with 8Gb memory. To give an idea about the complexity of the world modeled by ASTRO-CAptEvo, we have to mention 29 entity types (e.g., ships, cars, trucks, storage areas, various managers etc.) each including its own business process, 69 fragment models provided by entities and 40 types of context properties forming the context. During the runs of the demonstration, the number of entity instances simultaneously operating within the scenario reached up to 60.

Adaptation in Action

Using ASTRO-CAptEvo, we can show how the challenges posed by the motivating example of Section 3.1 can be addressed by our approach. For that purpose we can consider a process attached to a car (Fig. 8.6), which is the central process in the scenario.

By adopting the APFL language with context annotations (see Sections 3.3 and 4.2.2) and by implementing the refinement adaptation mechanism (see Section 3.2) we allow for highly customizable processes. Even though all three types of cars (normal, luxury and heavy) have to follow different procedures to accomplish the same tasks, the uniform process is used by all of them. This process contains mainly abstract activities, which are dynamically refined (i.e., customized) by the adaptation engine according to the needs of each individual car and taking into account the current conditions. To see that in the demo, it is enough to examine refinements of the same abstract activities for different cars using Process View.

By implementing all three adaptation mechanisms (local, refinement and compensation) and by enabling the adaptation engine to detect pre-condition violations, we can provide complex and intelligent reaction to exogenous events ("storage unavailable" and "car damaged" can be modelled in the current version of the demo). The strategy order used is to apply local adaptation, and, if it is not possible, to compensate the current refinement, and re-refine it. The first important observation is that the adaptation process produced in reaction to car damage (local adaptation), while always having the same objective (to repair a car and bring it back to the previous location), can be different for different cars/conditions.

Much more complex situations can be modelled by multiple failures happening in short period of time. In this case, an unexpected situation happens while the adaptation processes is still beign executed to handle the previous unexpected situation. For example, when a car gets damaged the respective adaptation process brings the car to the mechanical station, repairs it and brings it back to the location where it got damaged. If the second damage happens immediately after the repair of the previous damage the car will still be following a previous adaptation process the adaptation engine has to act in specific way. In particular, it skips the old

adaptation process, and re-plans it from scratch, taking into account the new conditions.

The most complex situation is where a car books a place at some storage facility (say, “Storage A”) and gets damaged on its way to it. Then, during the car repair, this facility becomes unavailable (similar situation is depicted in Fig. 3.4) Such chain of failures makes the adaptation engine to combine all three adaptation mechanisms to handle the situation correctly. After the car is repaired it tries to continue the execution of the old refinement of the Storage abstract activity. However, the car cannot be stored since “Storage A” is not available (the precondition of the activity storing a car to “Storage A” is violated). The attempt to resolve the precondition violation does not help (there is no way to force “Storage A” to be available) and so the refinement of activity Storage is compensated (the ticket is dropped), and a new refinement is generated (this one will store a car at “Storage C”).

The final important observation is that the vast majority of changes to the execution environment, such as adding new fragments, removing existing fragments, changing fragment implementations, changing fragment business policies (i.e., fragment annotations) require minimal or even no effort to keep the system operable. Indeed, the context-based fragment composition algorithm used as a core of adaptation engine, guarantees that all these changes are automatically reflected in the execution of all instances within the system, including those that already existed at the moment of changes..

Modeling Effort

Although the automated tools for process adaptation operate much faster than manual adaptation, we have to admit that they introduce quite some preliminary modeling overhead that has to be considered. Our approach is

not an exception since to enable automatic adaptation in the CLS scenario we have to create a context model (40 context properties) and properly annotate fragments (69 fragment) and entity processes (29 processes).

In order to compare the modeling overhead of our approach to other approaches, we chose one of the rule-based adaptation techniques ([42]) and used it to implement a very similar scenario. The adaptation was encoded as ECA (event-condition-action) rules [1].

While in [42] the context properties and fragment annotations are not modeled explicitly, the effort necessary for the encoding of the multiple rules and policies is estimated by us as comparable. In case of rules, the additional effort comes from the necessity to explicitly consider and encode various conceptually different adaptation cases (we counted them to be more than 20), which is not required in our goal-based approach. At the same time, we estimate our modeling approach to be better at modularization and reuse of the results of preliminary modeling. The rule-based systems seem to be much more centralized and less flexible with respect to changes. For example, the work on annotating fragments can be entrusted to fragment providers while rules being interleaved with each other have to be managed centrally. Another example is that a fragment annotation can be reviewed and adjusted by its provider separately from other elements of the application. In the centralized rule-based approaches any change in fragment and/or application policies may imply revision of the whole set of rules. In the same way, our framework can be easily and seamlessly extended with new fragments by simply annotating them properly and adding them to the repository. For the rule-based system, this would need to learn the whole rule system and modify and re-verify it again. Finally, some elements of our model can be reused by other applications in a given domain: the context model and fragment annotations, once defined, may

[1] http://soa.fbk.eu/Logistics-AGG.zip/

be adopted by different business processes. For the rules it is hardly the case.

Planning Performance

In order to evaluate the performance of the planning algorithm with respect to the adaptation-related composition problems, we ran our demo in continuous mode for around an hour and collected information about 1060 compositions performed within this time. For each composition we measured a number of indicators that characterized the complexity of the problem and the timing. Then we tried to organize them into charts that would allow us to prove or disprove the applicability of the approach.

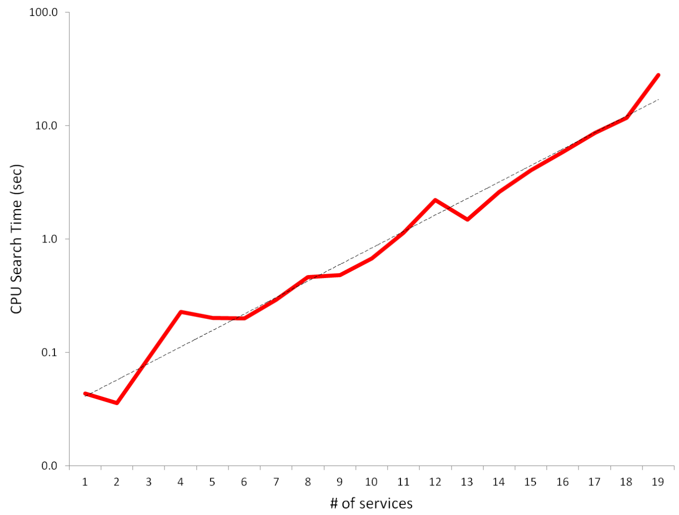


Figure 8.8: Dependency between performance and number of services composed

First of all, we check the scalability of the approach with respect to the number of services participating in the composition (Fig. 8.8). In general, the result corresponds to that of [18]. However, as we already mentioned in Section 8.1, the trend is supposed to be polynomial in the region of low values and to become exponential only for big values (since more time is

needed to optimize memory while working with big domains). The result of Fig. 8.8 can be explained taking into account that, in addition to services, the planning domain in context-aware composition contains context-related STSs, which makes it larger even for a small number of services presented. Consequently, even in the region of low values the exponential trend that is typical of larger domains dominates.

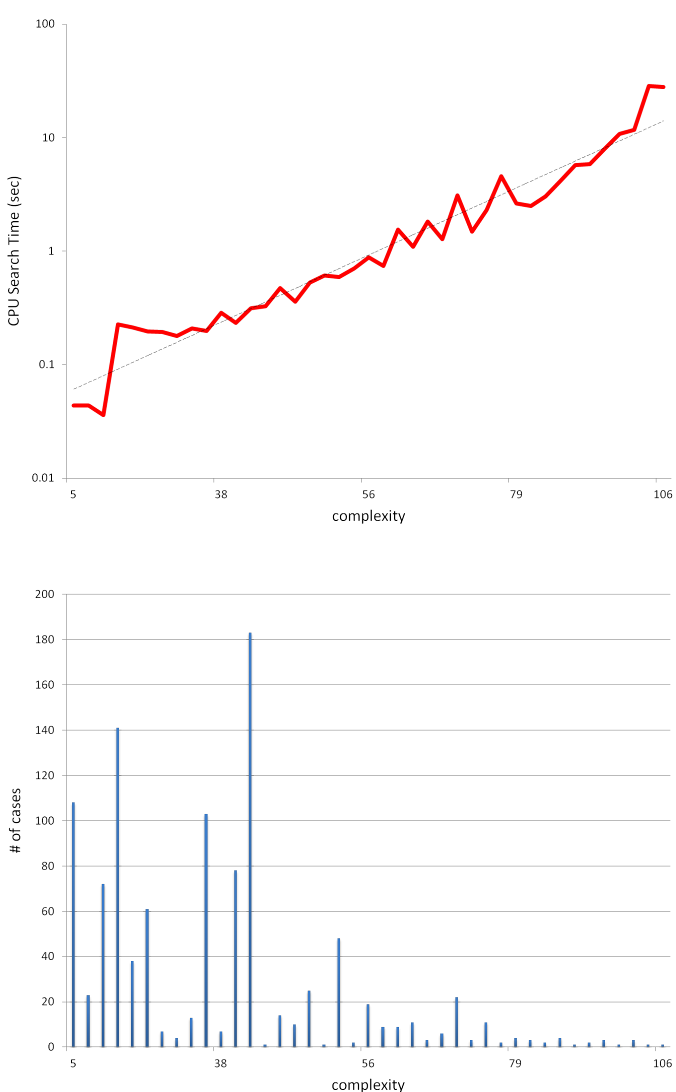


Figure 8.9: Complexity distribution and performance scalability

Alternatively, we propose our own indicator of domain complexity that is the total number of transitions in services and context properties making up the domain:

$$Complexity = NumContextTrans + NumServiceTrans.$$

We find this indicator more precise compared to the number of services. It also allows us to see a more fine-grained distribution of all composition problems with respect to complexity. The performance scalability with respect to composition complexity is represented by the chart in Fig. 8.9, top. It can be observed that it generally corresponds to the chart if Fig. 8.8 and features exponential growth. However, it is more informative if we consider this chart along with the complexity distribution of composition cases Fig. 8.9, bottom. It can be observed that the most adaptation cases reside in the region with low or moderate complexity, while the cases with high complexity are quite few. We remark that such distribution also affects the precision of the scalability chart in the region of high complexity (less experiments are carried out there).

Consequently, from the charts in Fig. 8.9 we can derive the following table showing the percentage of composition cases that are resolved in no more than n seconds:

n, sec	**compositions resolved within n**, %
0.1	19.07
1	91.12
3	96.51
10	99.62
30	100.00

From the table it can be observed that the vast majority of adaptation-related compositions actually take less than 10 seconds. This is the first evidence of practical applicability of our approach: although the perfor-

mance of context-aware composition degrades exponentially with growing complexity of a composition problem, it is still enough to be used for our purposes. This becomes especially true when we notice that in many application domain there are no severe restrictions on the performance of adaptation related tasks. For example, in the CLS scenario, the typical life cycle of a car may have duration up to several months. In this setting, even the composition that takes several minutes should not raise any problem.

The last important observation is that for each particular composition problem we build a planning domain that includes only the information that is relevant for this problem, namely: 1) the subset of context properties that are relevant for entities under consideration, which is normally a small portion of the overall context of the scenario and 2) the subset of all fragments that may be useful within the current composition problem, which is, again, only a small portion of all fragments currently available in the system. We expect that such fragment and context selection mechanism (whose prototype is already available in the current version of the demo platform) will allow us to preserve the same average size of the planning problem even for much larger (with respect to the number of entities) domains. Indeed, if within the scenario we operate thousands rather than dozens of cars at a time it is unlikely to increase the complexity of an average composition problem: the proper selection will always come up with more or less the same amount of relevant fragments and context. The fact that there are thousands of cars in the harbour rather than dozens does not functionally affects the way I park a car at a parking lot. As such, we expect our approach to be easily scalable in this regard. Of course, to run such a large system, much more powerful computers is supposed to be used.

Chapter 9

Conclusions and Future Work

In this dissertation, we investigated a problem of automated service composition in dynamic execution environments. We proposed a solution that integrally addresses the most important open issue associated with this problem. On the basis of our composition technique, we also proposed an approach to dynamic adaptation of service-based processes. The both solutions were implemented and evaluated.

We considered two types of composable components: conventional services and process fragments. We showed that under certain reasonable assumptions, services can be considered as fragments and, as such, the both types of components can be composed using the same composition techniques.

A large portion of the thesis was devoted to the idea of abstraction of composition requirements. We showed that once the requirements are separated from the details of composable components, they become much more robust against common run-time changes in the execution environment. As a consequence, the whole approach becomes much more suitable for dynamic conditions. In order to separate composition requirements from services, we introduced an explicit context model, which played the role of abstraction layer. We demonstrated how the abstract requirements can

be connected to services by means of service annotations. Finally, by extending and developing the ideas of [18], we showed that our context-based composition model can be resolved using planning techniques.

One of the main focuses of this research was composition requirements languages and their semanrics and implementation. To allow for more expressive control-flow requirements, we proposed our own abstract (context-based) language that introduced the following features: 1) the ability to express both reachability and procedural goals, 2) the ability to express reactive goals and 3) the ability to set preferences among alternative goals. Significant attention was paid to defining the reasonable semantics for the language and providing a way to encode this semantics in planning. To be able to process the new semantics, we proposed a new planning algorithm developing the ideas of [18, 111] and exploiting planning-as-model-checking approach. Although data-flow requirements were not considered in detail, we proposed a prototype solution based on the approach of [79].

On the basis of our composition technique, we developed a solution to the problem of dynamic adaptation of business processes. This required additional elaboration of some adaptation-related aspects. One of the major contributions here was our work on understanding and implementing various adaptation strategies. To enable rich adaptation possibilities, we used a special language for adaptable flows (APFL) additionally equipped with context-based annotations. All adaptation strategies in our adaptation engine were realized though run-time composition of process fragments.

One of the key contributions of the thesis is the ASTRO-CAptEvo demonstration platform. In it, we modeled a pervasive system based on the car logistics scenario and realized adaptable pervasive flows using our adaptation framework. The platform allowed us to evaluate not only the applicability of adaptation strategies but also the performance of the composition engine on adaptation-related tasks. It is also worth to mention

the implementation of a planning algorithm for continuous composition.

Of course, within this dissertation we could not cover every single issue related to the topics of interest. We admit that there are still many extensions and improvements we can consider in order to make the approaches more mature and profound. In the following we discuss the most important next steps we plan to take in the near future.

9.1 Future Work

User-Centric Service Composition

In the introduction, we mentioned user-centric systems as an example of dynamic environment where our ideas might be in demand. We expect that our approach to service composition can be extended to user-centric systems. Comparing business-centric and user-centric systems we can highlight some significant differences between them.

1. Business-centric composition usually has an ultimate goal to be achieved. User-centric composition aims to continuously support the user in performing a variety of different tasks and to react to possible changes in the user's objectives and execution environment;

2. The execution of a business-centric composition is normally driven and controlled by embedded business logic. The execution of user-centric service composition has to be controlled by the user, who is continuously informed about the execution progress and can make decisions;

3. In user-centric setting it is hard to provide an affordable predefined solution (we have neither information about services to compose nor information about user preferences). The composition has to be completely managed at run-time, with minimal human involvement.

We have to admit that user-centricity is SOA is an emerging topic and no complex solution to the above issues exist (some relevant works are [59, 34, 112]). At the same time, we can notice that we already have ingredients for addressing issues 1 and 3. Indeed, our requirements language already allows for composition constraints (preconditions), reactive behaviour and continuous composition requirements support (issue 1). Moreover, requirements are abstract and, while defined by IT experts at design time, can be grounded on services chosen by the user at run time, which contributes a lot to complete automation of the composition life cycle (issue 3). To address issue 2, we have to find a way to dynamically generate the composition interface/protocol that keeps the user "in the loop". Some preliminary results have already been published in [60]. We assume that the communication with the user can be linked to the evolution of the context model. For example, once a context event happens, the user has to be notified about that. Similarly, the interface has to allow the user to order the accomplishment of some task (e.g., to trigger some event).

The overall "idealistic" picture could be the following. At design time, the IT experts define a context model for some application domain and define a set of abstract requirements for a specific composition problem (e.g., integral management of trip elements, such as car rent, hotel reservation and flight tickets). In addition to that, service providers provide context-based annotations for their services. At run time, the user, who would like to organize a trip, first chooses a set of requirement expressions from the list that correspond to her needs (e.g., she wants "IF flight cancellation THEN book new flight" rather than "IF flight cancellation THEN trip cancellation" and so on). After that, she chooses service providers to be used and the resulting composition is produced automatically.

Composition Requirements

One of the direction for future work is the further development of the model of composition requirements. In the thesis, we noted that our methodology for representing complex, extended requirements in planning is powerful enough to handle even more complex constructs. The one obvious improvements is to make reaction expressions symmetric, i.e. to have both trigger and reaction part in form of any formula over events and context states (so far the trigger part is always a context event).

We plan to continue our work on data-flow requirements, which are not yet fully integrated to the approach. We also consider a problem of relation between data and control flow. For example, an activity precondition may be not only on the current context state but also on the values of data variables of the context. We expect it to be a very complex task that will probably partially adopt the ideas proposed in [101]

Performance Optimization

We mentioned that in the ASTRO-CAptEvo framework we already introduced some performance optimizations. They consist in removing the parts of the planning domain that will never belong to a plan. There are a few trivial observations that allow us to do so. For example, if we know that there is no service that triggers certain event, we can remove the respective transition because it will never be triggered. Similarly, we can remove components of STSs that are unconnected from the current states. In services, we can remove actions with preconditions that will never be satisfied through the execution. Our first experiments with such kind of optimization suggest that they can improve the performance of the planning by a factor of 2. We still need to structure and formalize these optimizing transformations.

Better Adaptation

There are a few directions for improving our process adaptation approach. One of them is proactive problem detection. For example, while executing a process we detect precondition violation only for the next activity to be executed. Our observations suggest that very often potential precondition violation can be detected much more in advance. In this case, we can act immediately, which may safe a lot of time and resources. The same is true for unrefined abstract activities.

Another direction is the further study of adaptation strategy selection and management. In this work we have chosen one most intuitive way of strategy selection. We presume that in different application domains efficient strategy order may be different. Moreover, we want to consider more sophisticated criteria for strategy selection such as QoS, status of the environment, execution and adaptation history etc.

Process Evolution

As we mentioned in Section 2.3, our adaptation approach belongs to short-term adaptation where changes are applied only to a single problematic process instance. At the same time, we remark that the by analyzing the adaptation history of a certain process it is possible to identify the directions in which we can change the process model (process evolution) in order to increase its efficiency. We currently consider the possibility to create an automated or semi-automated process evolution approach based on the aforementioned principles.

Bibliography

[1] EU-FET Project 213339 ALLOW. http://www.allow-project.eu/.

[2] S-Cube, the European Network of Excellence in Software Services and Systems. http://www.s-cube-network.eu/.

[3] *Service Research Challenges and Solutions for the Future Internet - S-Cube - Towards Engineering, Managing and Adapting Service-Based Systems*, volume 6500 of *Lecture Notes in Computer Science*. Springer, 2010.

[4] *ASTRO: Supporting the Composition of Distributed Business Processes*, 2012. http://www.astroproject.org/.

[5] M. Adams, A. ter Hofstede, D. Edmond, and W. van der Aalst. Worklets: A service-oriented implementation of dynamic flexibility in workflows. In *OTM Conferences 2006: CoopIS, DOA, GADA, and ODBASE*, Lecture Notes in Computer Science. 2006.

[6] R. Akkiraju, K. Verma, R. Goodwin, P. Doshi, and J. Lee. Executing abstract web process flows. In *Proceedings of the ICAPS Workshop on Planning and Scheduling for Web and Grid Services*, pages 9–15, 2004.

[7] A. Alamri, M. Eid, and A. El-Saddik. Classification of the state-of-the-art dynamic web services composition techniques. *IJWGS*, 2(2):148–166, 2006.

[8] L. Baresi, S. Guinea, and L. Pasquale. Self-healing BPEL processes with Dynamo and the JBoss rule engine. In *International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting*, ESSPE '07, 2007.

[9] L. Baresi, A. Maurino, and S. Modafferi. Workflow partitioning in mobile information systems. In *MOBIS*, IFIP International Federation for Information Processing, pages 93–106, 2004.

[10] S. Beauche and P. Poizat. Automated service composition with adaptive planning. In *ICSOC*, volume 5364 of *Lecture Notes in Computer Science*, pages 530–537, 2008.

[11] B. Benatallah, M. Dumas, M.-C. Fauvet, and F. A. Rabhi. Patterns and skeletons for parallel and distributed computing. Towards patterns of web services composition. pages 265–296. Springer-Verlag, 2003.

[12] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella. Automatic composition of transition-based semantic web services with messaging. In *VLDB 2005*, pages 613–624, 2005.

[13] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *Proc. of ICSOC 2003*, volume 2910 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2003.

[14] D. Berardi, D. Calvanese, G. D. Giacomo, and M. Mecella. Composition of services with nondeterministic observable behavior. In *ICSOC*, volume 3826 of *Lecture Notes in Computer Science*, pages 520–526. Springer, 2005.

[15] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: a Model Based Planner. In *IJCAI workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[16] P. Bertoli, R. Kazhamiakin, M. Paolucci, M. Pistore, H. Raik, and M. Wagner. Continuous orchestration of web services via planning. In *ICAPS*, 2009.

[17] 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. IEEE, 2009.

[18] P. Bertoli, M. Pistore, and P. Traverso. Automated composition of web services via planning in asynchronous domains. *Artificial Intelligence*, 174:316 – 361, 2010.

[19] F. Böse and J. Piotrowski. Autonomously controlled storage management in vehicle logistics applications of RFID and mobile computing systems. *International Journal of RT Technologies: Research an Application*, 1(1):57–76, 2009.

[20] A. Brogi and R. Popescu. Towards semi-automated workflow-based aggregation of web services. In *Proc. of ICSOC05, LNCS*, pages 214–227. Springer, 2005.

[21] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[22] A. Bucchiarone, N. Khurshid, A. Marconi, M. Pistore, and H. Raik. A car logistics scenario for context-aware adaptive service-based systems. In *Principles of Engineering Service Oriented Systems (PESOS), 2012 ICSE Workshop on*, pages 65 –66, june 2012.

[23] A. Bucchiarone, A. L. Lafuente, A. Marconi, and M. Pistore. A formalisation of Adaptable Pervasive Flows. In *WS-FM'09*, Bologna, Italy, 2009.

[24] A. Bucchiarone, A. Marconi, M. Pistore, and H. Raik. CAptEvo: Context-aware adaptation and evolution of business processes. In *ICSOC Workshops*, pages 252–254, 2011.

[25] A. Bucchiarone, A. Marconi, M. Pistore, and H. Raik. Dynamic adaptation of fragment-based and context-aware business processes. In *Proc. of ICWS 2012*, pages 33–41, 2012.

[26] A. Bucchiarone, M. Pistore, H. Raik, and R. Kazhamiakin. Adaptation of service-based business processes by context-aware replanning. In *Proc. of SOCA 2011*, pages 1–8, 2011.

[27] J. R. Burch, E. M. Clarke, K. L. Mcmillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond, 1990.

[28] M. H. Burstein, J. R. Hobbs, O. Lassila, D. Martin, D. V. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, and K. P. Sycara. DAML-S: Web service description for the semantic web. In *Proceedings of ISWC '02*. Springer-Verlag, 2002.

[29] D. Calvanese, G. D. Giacomo, M. Lenzerini, M. Mecella, and F. Patrizi. Automatic service composition and synthesis: the Roman Model. *IEEE Data Eng. Bull.*, 31(3):18–22, 2008.

[30] D. Calvanese and A. Santoso. Best service synthesis in the weighted Roman Model. In *ZEUS*, volume 847, pages 42–49, 2012.

[31] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), 2000.

[32] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artif. Intell.*, 147(1-2):35–84, 2003.

[33] M. Colombo, E. Di Nitto, and M. Mauri. SCENE: a service composition execution environment supporting dynamic changes disciplined through rules. In *Proceedings of ICSOC'06*, 2006.

[34] A. Corradi, E. Lodolo, S. Monti, and S. Pasini. A user-centric composition model for the Internet of Services. In *ISCC*, pages 110–117, 2008.

[35] G. De Giacomo, C. Di Ciccio, P. Felli, Y. Hu, and M. Mecella. Goal-based composition of stateful services for smart homes. In *CoopIS*, 2012.

[36] R. de Lemos and A. B. Romanovsky. Exception handling in the software lifecycle. *Comput. Syst. Sci. Eng.*, 16(2):119–133, 2001.

[37] M. de Leoni. Adaptive process management in highly dynamic and pervasive scenarios. In *YR-SOC*, volume 2 of *EPTCS*, pages 83–97, 2009.

[38] S. Dustdar and W. Schreiner. A survey on web services composition. *IInternational Journal on Web and Grid Services*, 1(1):1–30, 2005.

[39] H. Eberle, T. Unger, and F. Leymann. Process fragments. In *OTM Conferences (1)*, pages 398–405, 2009.

[40] W. W. Eckerson. Three tier client/server architecture: Achieving scalability, performance, and efficiency in client server applications. *Open Information Systems*, 10(1), 1995.

[41] H. Edelstein. Unraveling Client/Server Architecture. *DBMS*, 34(7), May 1994.

[42] H. Ehrig, C. Ermel, O. Runge, A. Bucchiarone, and P. Pelliccione. Formal analysis and verification of self-healing systems. In *FASE*, pages 139–153, 2010.

[43] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, pages 995–1072. Elsevier, 1995.

[44] A. Endpoints. *ActiveVOS*, 2010. http://www.activevos.com/.

[45] T. Erl. *Official SOA Principles Poster*, 2012. http://www.servicetechbooks.com/posters/SOA_Principles_Poster.pdf.

[46] K. Erol, J. Hendler, and D. S. Nau. Semantics for hierarchical task-network planning. Technical report, 1994.

[47] T. A. S. Foundation. *Apache ODE (Orchestration Director Engine)*, 2011. http://ode.apache.org/.

[48] M. Ghallab, A. Howe, C. Knoblock, D. Mcdermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL—The Planning Domain Definition Language, 1998.

[49] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[50] G. D. Giacomo, Y. Lespérance, and H. J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artif. Intell.*, 121(1-2):109–169, 2000.

[51] A. Hallerbach, T. Bauer, and M. Reichert. Managing process variants in the process lifecycle. In *10th Int'l Conf. on Enterprise Information Systems (ICEIS'08)*, pages 154–161, June 2008.

[52] 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.

[53] K. Herrmann, K. Rothermel, G. Kortuem, and N. Dulay. Adaptable Pervasive Flows - An Emerging Technology for Pervasive Adaptation. In *Workshop on Pervasive Adaptation (PerAda)*. IEEE Computer Society, September 2008.

[54] IBM and SAP. *WS-BPEL Extension for People*, 2005. http://www.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/.

[55] innoQ. *Web Services Standards Overview*, 2007. http://www.innoq.com/soa/ws-standards/poster/.

[56] Z. Jaroucheh, X. Liu, and S. Smith. Apto: A MDD-based generic framework for context-aware deeply adaptive service-based processes. In *Web Services (ICWS), 2010 IEEE International Conference on*, july 2010.

[57] S. Kalasapur, M. Kumar, and B. Shirazi. Dynamic service composition in pervasive computing. *Parallel and Distributed Systems, IEEE Transactions on*, 18(7):907 –918, july 2007.

[58] D. Karastoyanova, A. Houspanossian, M. Cilia, F. Leymann, and A. P. Buchmann. Extending BPEL for Run Time Adaptability. In *Proc. EDOC'05*, pages 15–26, 2005.

[59] R. Kazhamiakin, P. Bertoli, M. Paolucci, M. Pistore, and M. Wagner. Having services "YourWay!": Towards user-centric composition of mobile services. In *FIS*, pages 94–106, 2008.

[60] R. Kazhamiakin, M. Paolucci, M. Pistore, and H. Raik. Modelling and automated composition of user-centric services. In *Proc of OTM Conferences 2010*, volume 6426 of *Lecture Notes in Computer Science*. Springer, 2010.

[61] R. Kazhamiakin, M. Pistore, and M. Roveri. Formal verification of requirements using SPIN: A case study on web services. In *SEFM*, 2004.

[62] R. Khalaf, N. Mukhi, and S. Weerawarana. Service-oriented composition in BPEL4WS. In *WWW (Alternate Paper Tracks)*, 2003.

[63] W. Kongdenfha, R. Saint-paul, B. Benatallah, and F. Casati. An aspect-oriented framework for service adaptation. In *In ICSOC06*, pages 15–26. ACM Press, 2006.

[64] J. Kopecky, T. Vitvar, C. Bournez, and J. Farrell. SAWSDL: Semantic annotations for WSDL and XML Schema. *IEEE Internet Computing*, 11:60–67, 2007.

[65] J. Korhonen, L. Pajunen, and J. Puustjärvi. Automatic composition of web service workflows using a semantic agent. In *Web Intelligence*, pages 566–569, 2003.

[66] U. Küster, M. Stern, and B. König-Ries. A classification of issues and approaches in automatic service composition. In *WESC 2005*, 2005.

[67] U. D. Lago, M. Pistore, and P. Traverso. Planning with a language for extended goals. In *Eighteenth national conference on Artificial intelligence*, pages 447–454, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[68] I. Lanese, A. Bucchiarone, and F. Montesi. A framework for rule-based dynamic adaptation. In *Proceedings of the 5th international conference on Trustworthly global computing*, TGC'10, 2010.

[69] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *J. Log. Program.*, 31(1-3):59–83, 1997.

[70] F. Leymann. Web services: Distributed applications without limits. In *Proc. BTW'03*, pages 26–28, 2003.

[71] F. Leymann. *BPEL vs BPMN 2.0: Should you care?*, 2009. http://leymann.blogspot.it/2009/12/bpel-vs-bpmn-20-should-you-care.html.

[72] X. Liu, Y. Hui, W. S. 0001, and H. Liang. Towards service composition based on mashup. In *IEEE SCW*, pages 332–339. IEEE Computer Society, 2007.

[73] Y. Liu, A. H. H. Ngu, and L. Zeng. QoS computation and policing in dynamic web service selection. In S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, editors, *WWW (Alternate Track Papers Posters)*, pages 66–73. ACM, 2004.

[74] A. Lotem, D. S. Nau, and J. A. Hendler. Using planning graphs for solving HTN planning problems. In J. Hendler and D. Subramanian, editors, *AAAI/IAAI*, pages 534–540. AAAI Press / The MIT Press, 1999.

[75] A. Marconi. Automated process-level composition of web services: from requirements specification to process run, 2008. PhD Dissertation.

[76] A. Marconi and M. Pistore. Synthesis and composition of web services. In M. Bernardo, L. Padovani, and G. Zavattaro, editors, *SFM*, volume 5569 of *Lecture Notes in Computer Science*, pages 89–157. Springer, 2009.

[77] 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.

[78] A. Marconi, M. Pistore, A. Sirbu, H. Eberle, F. Leymann, and T. Unger. Enabling adaptation of pervasive flows: Built-in contextual adaptation. In *ICSOC/ServiceWave*, volume 5900 of *Lecture Notes in Computer Science*, 2009.

[79] A. Marconi, M. Pistore, and P. Traverso. Specifying data-flow requirements for the automated composition of web services. pages 147–156. IEEE Computer Society, 2006.

[80] S. A. McIlraith and T. C. Son. Adapting Golog for composition of semantic web services. April 2002.

[81] K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publ., 1993.

[82] K. L. McMillan. The SMV language. Technical report, Cadence Berkeley Labs, March 1999.

[83] J. Mendling and M. Hafner. From WS-CDL choreography to BPEL process orchestration. 21(5):525–542, 2008.

[84] H. Meyer and M. Weske. Automated service composition using heuristic search. In *Proc BPM 2006*, pages 81–96. Springer, 2006.

[85] F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. JOLIE: a Java Orchestration Language Interpreter Engine. *Electron. Notes Theor. Comput. Sci.*, 181, June 2007.

[86] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing execution of composite web services. In *OOPSLA*, pages 170–187. ACM, 2004.

[87] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 77–88, 2002.

[88] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN planning system. *J. Artif. Intell. Res. (JAIR)*, 20:379–404, 2003.

[89] A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 2003.

[90] OASIS. *UDDI Version 3.0.2*, 2004. http://uddi.org/pubs/uddi$_v$3*.htm*.

[91] OASIS. *Web Services Business Process Execution Language 2.0*, 2006. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html.

[92] OMG. *Business Process Model And Notation (BPMN)*, 2011. http://www.omg.org/spec/BPMN/2.0.

[93] OMG. *Common Object Request Broker Architecture (CORBA/IIOP)*, November 2011. http://www.omg.org/spec/CORBA/3.2/.

[94] Oracle. *Java Remote Method Invocation*, 2011. http://docs.oracle.com/javase/6/docs/technotes/guides/rmi/index.html.

[95] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Krämer. Service-oriented computing research roadmap. In *Dagstuhl Seminar Proceedings 05462*, pages 1–29, April 2006.

[96] J. Peer. A PDDL based tool for automatic web service composition. In H. J. Ohlbach and S. Schaffert, editors, *PPSWR*, volume 3208 of *Lecture Notes in Computer Science*, pages 149–163. Springer, 2004.

[97] J. Peer. Web Service Composition as AI Planning - a Survey. Technical report, University of St. Gallen, Switzerland, 2005.

[98] C. Peltz. Web service orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[99] M. Phan and F. Hattori. Automatic web service composition using ConGolog. In *ICDCS Workshops*, page 17. IEEE Computer Society, 2006.

[100] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and monitoring web service composition. In *AIMSA*, pages 106–115, 2004.

[101] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated composition of web services by planning at the knowledge level. In *Proceedings of the 19th international joint conference on Artificial intelligence*, IJCAI'05, pages 1252–1259, 2005.

[102] M. Pistore, P. Traverso, and P. Bertoli. Automated composition of web services by planning in asynchronous domains. In *ICAPS*, pages 2–11. AAAI, 2005.

[103] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite BPEL4WS web services. pages 293–301. IEEE Computer Society, 2005.

[104] S. R. Ponnekanti and A. Fox. Sword: A developer toolkit for web service composition. In *Proceedings of the 11th International WWW Conference (WWW2002)*, 2002.

[105] H. Raik, A. Bucchiarone, N. Khurshid, A. Marconi, and M. Pistore. ASTRO-CAptEvo: Dynamic context-aware adaptation for service-based systems. In *Proc. of: IEEE 8th World Congress on Services - SERVICES 2012*, 2012.

[106] J. Rao and X. Su. A survey of automated web service composition methods. volume 3387 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2004.

[107] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, 2007.

[108] S. Sardina, G. De Giacomo, Y. Lespérance, and H. Levesque. On the semantics of deliberation in IndiGolog from theory to implementation. *Annals of Mathematics and Artificial Intelligence*, 41:259–299, 2004.

[109] H. Schonenberg, R. Mans, N. Russell, N. Mulyar, and W. M. P. van der Aalst. Towards a taxonomy of process flexibility. In *CAiSE Forum*, pages 81–84, 2008.

[110] D. Shaparau. Complex goals for planning in nondeterministic domains: preferences and strategies, 2008. PhD Dissertation.

[111] D. Shaparau, M. Pistore, and P. Traverso. Contingent planning with goal preferences. In *AAAI*, pages 927–935, 2006.

[112] E. Silva, L. F. Pires, and M. van Sinderen. On the support of dynamic service composition at runtime. In *Proceedings of the 2009 international conference on Service-oriented computing*, ICSOC/ServiceWave'09, 2009.

[113] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic composition of web services using semantic descriptions, 2002.

[114] D. Skogan, R. Gr, and I. Solheim. Web service composition in UML. In *EDOC*, pages 47–57. IEEE Computer Society, 2004.

[115] S. Thakkar, J. L. Ambite, and C. A. Knoblock. A data integration approach to automatically composing and optimizing web services. In *Proceedings of the ICAPS Workshop on Planning and Scheduling for Web and Grid Services*, 2004.

[116] S. Thakkar, C. A. Knoblock, and J. L. Ambite. A view integration approach to dynamic composition of web services. In *Proceedings of 2003 ICAPS Workshop on Planning for Web Services*, 2003.

[117] P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *International Semantic Web Conference (ISWC)*, pages 380–394. Springer-Verlag, 2004.

[118] W. van der Aalst and A. ter Hostede. YAWL: Yet Another Workflow Language. *Information Systems*, 2004.

[119] W3C. *Simple Object Access Protocol (SOAP) 1.1*, 2000. http://www.w3.org/TR/soap/.

[120] W3C. *Web Services Description Language (WSDL) 1.1*, 2001. http://www.w3.org/TR/wsdl.

[121] W3C. *OWL-S: Semantic Markup for Web Services*, 2004. http://www.w3.org/Submission/OWL-S/.

[122] W3C. *Web Services Architecture*, 2004. http://www.w3.org/TR/ws-arch/.

[123] W3C. *Web Services Choreography Description Language Version 1.0*, 2004. http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/.

[124] W3C. *Web Services Glossary*, 2004. http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/.

[125] W3C. *XML Schema 1.1*, 2004. http://www.w3.org/XML/Schema.

[126] W3C. *Web Service Modeling Ontology (WSMO)*, 2005. http://www.w3.org/Submission/WSMO/.

[127] W3C. *Web Services Activity*, 2007. http://www.w3.org/2002/ws/.

[128] W3C. *Web Services Activity Statement*, 2012. http://www.w3.org/2002/ws/Activity.

[129] B. Weber, S. Sadiq, and M. Reichert. Beyond rigidity dynamic process lifecycle support. *Computer Science - Research and Development*, 2009.

[130] D. Wu, B. Parsia, E. Sirin, J. A. Hendler, and D. S. Nau. Automating DAML-S web services composition using SHOP2. In *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2003.

[131] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for web services composition. *Software Engineering, IEEE Transactions on*, 30(5):311–327, 2004.

[132] Y. Zhai, J. Zhang, and K.-J. Lin. SOA middleware support for service process reconfiguration with end-to-end QoS constraints. *2012 IEEE 19th International Conference on Web Services*, 0, 2009.