View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Archivio della ricerca- Università di Roma La Sapienza

Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XVII Ciclo – 2005

Automatic Service Composition.
Models, Techniques and Tools.

Daniela Berardi

Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XVII Ciclo - 2005

Daniela Berardi

Automatic Service Composition.
Models, Techniques and Tools.

Thesis Committee

Prof. Maurizio Lenzerini (Advisor)
Prof. Giuseppe De Giacomo
Dr. Massimo Mecella

Reviewers

Prof. Richard Hull
Dr. Fabio Casati

Author's address:
Daniela Berardi
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
e-mail: berardi@dis.uniroma1.it
www: http://www.dis.uniroma1.it/∼berardi/

To my family

W*ir müssen wissen, wir werden wissen*
(We must know, we shall know)
David Hilbert (1862 - 1943)

Acknowledgments

During these three years of studies I received support from many people: it is therefore a pleasure to acknowledge all of them.

First of all, I express my deepest gratitude to Maurizio Lenzerini, Giuseppe De Giacomo, Diego Calvanese (unfortunately, now in Bozen) and Massimo Mecella, for their continuous support and valuable suggestions, that have made my research possible. Specifically, I would like to thank Maurizio, Giuseppe and Diego for all the many "logic-based" discussions, that indeed contributed forming my theoretical background, and Massimo for always focusing on possible applications of our results, that taught me not to forget the practical side of research. I have been given the rare privilege to grow up along to both theoretical and applicative directions. I am really grateful to all of them for this great pleasure.

A special thank goes also to Giuseppe De Giacomo for having inspired and encouraged since the years of my master thesis.

I wish to thank my Thesis Committee, Maurizio Lenzerini, Giuseppe De Giacomo and Massimo Mecella: they have not simply assisted me during my studies, but have worked with me in many papers. I also wish to thank the external reviewers, Rick Hull and Fabio Casati: their comments and suggestions have really been useful to improve the contents and the presentation of this thesis.

Many thanks to all the people of Dipartimento di Informatica e Sistemistica of Università di Roma "La Sapienza". In particular, I would like to thank Tiziana Toni, our system administrator, the "column" of the network in our Department, and Signora Pia for the coffee she used to prepare twice a day: we are really missing that good coffee!

Thanks also to all the friends of the Department, with which I shared these years, in particular, Alessia Milani, Khan Shahram Bahadori Gouchani, Diego Milano. Additionally, I would like to thank Monica Scannapieco, for hosting me when I was without a house in Rome (the life of non-resident students can be hard sometimes): I will always remember those days as among the most beautiful of my life! A great thank to Massimo Mecella, aka peer,

for introducing and explaining me the chaotic and incoherent bureaucratic rules of the University (and for standing me while I was living with Monica, now his wife). Of course, sport is an important part of life: so I thank all the members of my soccer team, Luigi Laura, Alessandro Termini (aka Bill), Valeria Mirabella and all the other guys with which I shared great moments.

I would like to thank Rick Hull for inviting me to spend some months at Bell Labs – Lucent Technologies Inc. Murray Hill (NJ, USA) and the people at the Network Data and Services Research Department, in particular, Chris McTague, Irini Fundulaki, Arnaud Sahuguet, Jerome Simenon (now at IBM), Roseanne Pasquarello, Rob Dinoff, Michael Benedikt (during his visits at Bell Labs), Sriram Varadarajan, Daniel Lieuwen, Peter F. Patel-Schneider and Vinod Anupam, that made my stay there so joyful. I am especially indebted to Rick Hull for the great and deep brainstormings that we make every time we meet. I am also grateful to Rick for having introduced me in the SWSL (Semantic Web Service Language) working group: having the possibility to talk to internationally known highest-level researchers and professors and share ideas with them is a privilege given to few people. Participating to SWSL teleconferences has allowed me to share interesting ideas in particular with Sheila McIlraith and Michael Gruninger, as well as with Rick Hull: I thank them for all the useful discussions. Finally, I would like to thank Noel Aird, Kathy McGarry and all my numerous relatives in the USA for taking care of me during my stay in the USA.

Last, but not least, I want to thank my family, my father and mother, Francesco e Maria Luisa, and my two youngest sisters, Silvia and Laura, for having always been near me in these last years of hard work and study. Without their support and patience this work would have never come into existence. I also want to thank my grandparents, Nonno Barba and Nonna Nì, Nonna Grazia and Nonno Sergio (that passed away while I was in the USA), that have always understood me for not having spent so much time with them.

Rome, Italy
December 2nd, 2004

Daniela Berardi

Contents

1 Introduction **1**

1.1 Service Oriented Computing 1
1.2 Services and Service Composition 2
1.2.1 What is a Service? 2
1.2.2 Research Issues on Services and Service Composition . . 4
1.3 Goals and main results of the thesis 8
1.4 Structure of the thesis 10

2 Related Work **13**

2.1 Service Description and Modeling 14
2.1.1 Industry 15
2.1.2 Academia 17
2.2 Service Composition Synthesis 22
2.2.1 Industry 24
2.2.2 Academia 24
2.3 Related Research Areas 30
2.3.1 Data Integration 30
2.3.2 Program Synthesis 32
2.3.3 Planning and Reasoning About Actions 33
2.3.4 Software Engineering 34
2.4 Comparison of Current Approaches 35
2.5 Results related to Service Description and Composition 39
2.5.1 Service Discovery 39
2.5.2 Service Orchestration 40

3 Services and Service Composition: General Characterization **43**

3.1 Basics on Services 43
3.2 Service Community 45
3.3 Service Schema 46
3.3.1 Labeled Trees 47
3.3.2 External Schema 47

3.3.3 Internal Schema . . . 50
3.4 Service Instances . . . 53
3.4.1 Running a Service Instance . . . 54
3.5 Client Specification as Target Service . . . 57
3.6 Composition Synthesis . . . 58
3.7 Discussion . . . 60

4 Service Behavioral Description as FSM and Automatic Composition 63
4.1 Services with Behavioral Description as Finite State Machines . 64
4.2 Preliminaries on Deterministic Propositional Dynamic Logic . . 72
4.3 Automatic Service Composition . . . 75
4.3.1 Checking Existence of a Composition . . . 75
4.3.2 Synthesizing a Composition . . . 81
4.4 Discussion . . . 89

5 Loosely Specified Target Services 91
5.1 Underspecified Target Service . . . 91
5.2 The Problem of Service Composition . . . 96
5.3 Automatic Service Composition Synthesis Technique . . . 98
5.3.1 Existence of a Composition . . . 98
5.3.2 Synthesizing a Composition . . . 104
5.4 Discussion . . . 106

6 Allowing Services to Take the Initiative 109
6.1 Servant and Initiator Services . . . 110
6.2 Client Specification . . . 113
6.3 The problem of Service Composition . . . 116
6.4 Composition Synthesis Technique . . . 121
6.4.1 PDL_{gm} . . . 121
6.4.2 Composition Existence . . . 122
6.4.3 Synthesizing a Composition . . . 128
6.4.4 Composition synthesis of our Running Example . . . 130
6.5 Discussion . . . 139

7 $\mathcal{ESC}$ E-Service Composer 141
7.1 Considerations about the PDL Encodings from an Implementation Perspective . . . 141
7.2 The Description Logic $\mathcal{ALU}$ . . . 143
7.2.1 The Correspondence between $DPDL_{lim}$ and $\mathcal{ALU}$ . . . 146
7.3 $\mathcal{ALU}$ Encoding of Service Composition . . . 148
7.3.1 Encoding in $\mathcal{ALU}$ the Framework of Chapters 4 and 5 . 148

7.3.2 Results on Composition Existence and Synthesis in $\mathcal{ALU}$ 150
7.4 The Service Composition Tool $\mathcal{ESC}$ 153
7.4.1 The Abstraction Module 155
7.4.2 Implementation of the Synthesis Engine Module 157
7.4.3 Implementation of the Realization Module 157

8 Conclusions and Future Work 163
8.1 Conclusions 163
8.2 Future Work 164
8.2.1 Simulation 164
8.2.2 FLOWS 165
8.2.3 The COLOMBO Framework 166
8.2.4 Modeling the Client 167

A Characterizing Services and Service Composition in Situation Calculus 169
A.1 Preliminaries on Situation Calculus 169
A.2 Service Composition in Situation Calculus 171
A.3 Computing Service Composition 176
A.4 Discussion 178

Bibliography 181

Chapter 1

Introduction

1.1 Service Oriented Computing

Service Oriented Computing (SOC [121, 69]) is a computing paradigm aiming at realizing distributed applications by reusing existing services as basic building blocks. Intuitively, services (or Web Services or *e*-Services)[1] are software platform-independent applications that export a description of their functionalities and make it available using standard network technologies. Services are able to perform a wide spectrum of activities, from simple requests to complicated business processes. In other words, they represent a new way in the utilization of the web, by supporting rapid, low-cost and easy interoperation of loosely coupled heterogeneous distributed applications. One of the most important sectors that can benefit from the application of SOC paradigm is the *e*-Business sector, e.g., *e*-Business solutions and on-line service provision. SOC provides the key enabling infrastructure required to integrate supply chains and end-to-end *e*-Business applications at a variety of different levels, from applications and data, to business processes, and across (and within) organizations. In other words, the SOC paradigm envisions to wrap existing applications and expose them as services. Indeed, it allows organizations to expose their core competencies declaratively over the Internet or a variety of networks, e.g., cable, UMTS, xDSL, Bluetooth, etc., using (open) standard (XML-based) languages and protocols. Such standards allow developers to access applications deployed over the network based on what they do, rather than on how they do it, or how they have been implemented. Thus, services help integrate applications that were not written with the intent to be easily

[1] The term "Web Service" refers to those services that use the Internet and open Internet-based standards for communication. The term "*e*-Service" is used to denote programs interacting over a general electronic medium. In this thesis we use the term "service" since we address them from a semantic and conceptual perspective.

integrated with other applications and define architectures and techniques to build new functionalities while integrating existing application functionalities. This is possible since they are independent of specific programming languages or operating systems.

Services are meant to be used by other applications (and possibly other services), and not only by humans. In other words, there is a clear distinction between a service and a web portal or a service over the web, the latter ones being applications invoked by humans *only*, through web pages. Currently, the web is populated primarily by applications targeted towards humans. However, there is a common understanding that to make a leap from services being recognized and used through human intervention, to services being recognized and used also by autonomous software applications, services must export their "semantics". Services should provide semantically-rich, application-targeted descriptions, including dynamic behavioral features, complex forms of dataflow, transactional attitudes, adaptability to varying circumstances, security and so on.

From the above discussion, it stems that the SOC paradigm poses many challenging research issues, both from a conceptual and from a technological perspective. In this thesis we concentrate on the former ones.

1.2 Services and Service Composition

In this section we define the basic notions which are addressed and studied in this thesis, namely, services and the problem of service composition.

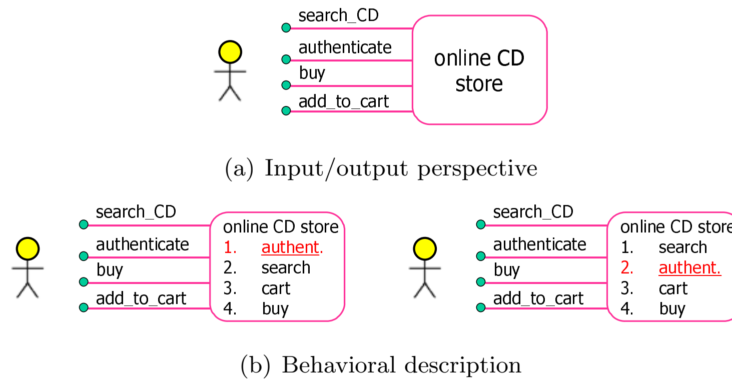
1.2.1 What is a Service?

In the literature there is no common understanding of what services are, since the term "service" not always is used with the same meaning. In [69] an interesting and detailed discussion is provided on existing definitions, on top of which we base what follows. Current definitions of the term "service" range from very generic and somewhat ambiguous, to very restrictive and too technology-dependent. On one end of the spectrum, there is the generic idea that every application that is characterized by an URL is a service, thus focusing on the fact that a service is an application that can be invoked over the Web. On the other end of the spectrum, there are very specific definitions such as for example, the one provided by the technical dictionary Webopedia [51], according to which a service is "a standardized way of integrating Web-based applications using the XML, SOAP, WSDL, and UDDI open standards over an Interned protocol backbone. XML is used to tag the data, SOAP is used to transfer the data, WSDL is used for describing the services

available, and UDDI is used for listing what services are available." This definition tightly relates services to today state-of-the-art web standards, such as Web Service Description Language (WSDL [6]), Simple Object Access Protocol (SOAP [147]), Universal Description, Discovery and Integration repository (UDDI [142]), which are, respectively, the description language for service, the protocol supporting interactions among services and the distributed repository where services are published.[2] However, such standards evolve continuously and may be subject to revisions and extensions, due to new requirements and possible changes/refinements in the vision of SOC. Other definitions, which lie in the middle of the spectrum are the following ones, provided by two standardization consortia, respectively, UDDI and World Wide Web (W3C [148]). According to the former, services are "self-contained, modular business applications that have open, Internet-oriented, standard-based interfaces". The latter envisions a service as a "software application identified by a URL, whose interfaces and bindings are capable of being defined, described and discovered as XML artifacts. A service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols."

It is not our intention to discuss here and provide the right definition of service, rather, we want to observe that all the above definitions agree on the fact that
a service is a distributed application that exports a view of its functionalities.
In general, such a view can be described either from an input/output (I/O) perspective only, or, in addition, also in terms of a behavioral description. The I/O perspective is taken by standards such as WSDL [6] (see Section 2.1.1), by the Semantic Web Service community [5] (see Section 2.1.2), and it consists in describing a service in terms of the exported operations, with no or very limited specification of constraints on the execution order of operations. Consider Figure 1.1(a): it represents a service described from an I/O perspective that allows for buying CDs online, through the shown operations, which have intuitive meaning (for simplicity the input and output parameters are not shown in the figure). A client that interacts with it could in principle execute the offered operations in any order. For instance, a client could buy a song before adding it to the cart. Additionally, there are services in which the client needs to be authenticated before executing any other action (as in the leftmost service of Figure 1.1(b)), in other the authentication takes place before buying the CD or before adding it to the cart (as in the rightmost service of Figure 1.1(b)): such services export the same set of functionalities, but they do not support the same interactions with their clients. Therefore, they seem to be "equal" when considering the I/O perspective only, but actually they are not, since they have different behavioral descriptions. This example shows that *(i)* in

[2] WSDL and UDDI will be further addressed in Chapter 2.



(a) Input/output perspective

(b) Behavioral description

Figure 1.1: Different views of a service functionalities

order to completely describe a service, ordering constraints on operations need to be defined, and that *(ii)* such ordering constraints can be either intrinsic on the operations (`add_to_cart` must happen before `buying` a CD) or depend on the implementation logic of the service itself (`authentication` before `search`, or after `search` but before `add_to_cart`). For this reason, recently it is advocated [69, 89, 17] that services should also be described in terms of their behavioral description, that is, all the possible sequences of operations that the service supports for execution. Such behavioral descriptions are often called conversations. Therefore, in the rest of this thesis, we consider a service as *a distributed application that exports a view of its functionalities in terms of its behavioral description.*

1.2.2 Research Issues on Services and Service Composition

The commonly accepted and *minimal* framework for Service Oriented Computing is the Service Oriented Architecture (SOA) [87]. It consists of the following basic roles: *(i)* the *service provider* is the “owner” of a service, i.e., the subject (e.g., an organization) providing services; *(ii)* the *service requestor*, also referred to as client is the subject looking for and invoking the service in order to fulfill some goals, e.g., to obtain desired information or to perform some actions; and *(iii)* the *service directory* is the subject providing a repository/registry of service descriptions, where providers publish their services and requestors find services.

When a provider wants to make available a service, he *publishes* its interface (input/output parameters, message types, set of operations, etc.) in the directory, by specifying information on himself (name, contacts, URL, etc.), about the service invocation (natural language description, a possible list of authorized clients, on how the service should be invoked, etc.) and so on. A

requestor that wants to reach a goal by exploiting service functionalities, *finds* (i.e., discovers) a suitable service in the directory, and then *binds* (i.e., connects) to the specific service provider in order to invoke the service, by using the information that comes along with the service. Note that this framework is quite limited, for instance, a service is characterized only in terms of its interface: no support for behavioral description of services is considered. Also, SOA conceives only the case when the client request matches with just one service, while in general it may happen that several services collaborate to its achievement. Despite this, SOA clearly highlights the main research challenges that SOC gives rise to.

Research on services wrt SOA and SOC spans over many interesting issues. *Service description* is concerned with deciding which services are only needed for implementation purposes and which ones are publicly invocable; additionally, it deals with how to describe the latter class of services, in order to associate to each service precise syntax and semantics. *Service discovery, selection and invocation* considers how customers can find the services that best fulfill their needs, how such services, and, consequently, the providers that offer them, can be selected, how the clients can invoke and execute the services. Other interesting areas are: *service advertisement*, which focuses on how providers can advertise their services, so that clients can easily discover them; *service integration*, that tackles the problem of how services can be integrated with resources such as files, databases, legacy applications, and so on; *service negotiation*, dealing with how the entities involved negotiate their role and activities in providing services. The notions of *reusability* and *extensibility* are key to the SOC paradigm: they consider how more complex or more customized services can be built, by starting from other existing services, thus saving time and resources. Security and privacy issues are of course important, since a service should be securely delivered only to authorized clients and private information should not be divulged. Last but not least *quality of services* (QoS) should be guaranteed. It can be studied according to different directions, i.e., by satisfying reasonable time or resource constraints; by reaching a high-level of data quality, since data of low quality can seriously compromise results of services. Guaranteeing a high quality of services, in terms of service metering and cost, performance metrics (e.g., response time), security attributes, (transactional) integrity, reliability, scalability, availability, etc. allows clients to trust and completely depend upon services, thus achieving a high degree of *dependability*, property that Information Systems should have [150]. Note that all the research areas highlighted above are tightly related one with the other. For instance, discovery, selection and invocation of services requires correctly describing services.

In this thesis, we focus on another very interesting research topic, *service*

composition. Service composition addresses the situation when a client request cannot be satisfied by any available service, but a *composite* service, obtained by combining “parts of” available *component* services, might be used. The composite service can be regarded as a kind of client wrt its components, since it (indirectly) looks for and invokes them.

Example 1 *A researcher wants to arrange for a participation to a conference: he would like to register to a conference, book for a hotel room and arrange for the travel; in particular, the researcher wants to buy either a train ticket or a plane ticket, making the decision according to some information that he gathers during the service execution. Therefore, he connects to a publicly accessible (advanced) repository of services, where he specifies his request and in his turn he receives a service that realizes it (if one exists). Such returned service should interact with the researcher, and allow him to make some choices, which may depend on the results of previously executed interactions. Note that, in general, it is likely to happen that the researcher request cannot be realized by any* single *available service, but by a new composite service, obtained by* coordinating *a* set *of available services: however the researcher is unaware of how many services are involved in the fulfillment of his request. Consider the situation when (a software module of the advanced service repository finds out that) the reseacher request can be realized by coordinating the following set of component services:* `Conference Registration` *service that allows first for registering to the conference, then for booking a hotel, and finally for charging the researcher credit card with a suitable amount;* `Travel Information` *service that gives information about a specific location (i.e., weather in specific period, distance from given location, etc.);* `Travel Arrangement` *service that allows for buying either a plane or a train ticket. Note that a correct execution of the* `Conference Registration` *service consists of all the three operations, performed in the specified order. The researcher request can be realized by a new composite service, obtained by coordination of the available component services as follows: first it allows the reseacher for registering to the conference, by executing the first operation of* `Conference Registration` *service, then it lets him invoking* `Travel Information`*: on the basis of the information returned, the new service lets him to choose whether to buy a plane or a train a ticket with* `Travel Arrangement` *service. Finally, having decided the dates for travel, he is allowed to book the hotel room and to give his credit card for being charged, by executing the remaining operations of* `Conference Registration` *service.* □

Service composition involves two different issues, namely *composition synthesis* and *orchestration*:

1. Given a set of available services and a client request, the problem of *composition synthesis*, or simply *composition*, is concerned with synthesizing a new composite service, thus producing a specification of how to coordinate the available services to realize the client request. Such a specification can be obtained either *automatically*, i.e., using a tool that implements a composition algorithm, or *manually* by a human.

2. Given the specification of the composite service synthesized in the previous phase, the problem of *orchestration* deals with coordinating the various component services, and monitoring control and data flow among them, in order to guarantee the correct execution of the composite service.

Our main focus in this thesis is on automatic service composition synthesis.
Orchestration has been addressed in other research areas (e.g., workflow) and many issues have already been studied (e.g., transaction, exception handling, etc.), therefore we rely on results of other fields. In Chapter 2 we discuss some of such results. Note also that orchestration is modular to composition synthesis, since it takes in input the composite service produced by the composition synthesis.

Finally, it is clear that service composition is tightly related with all the previously mentioned research areas on services, especially with *reusability* and *extensibility*. In addition, it makes all of them more compelling. It is interesting, for example, to study *(i)* how to discover the (possibly minimal) set of available services to be used in the composition, *(ii)* whether and how to advertise component services which constitute part of the composite one, *(iii)* how to define the quality of the composite service in terms of the component service and wrt the client request, *(iv)* how to set security and privacy policies of the composite service based on those of the component ones. Service composition also involves new issues, namely *compatibility* and *substitution*. They came into play in the following situation. A service S_1 that constitutes part of a composed service S_c becomes unavailable: in order to continue exploiting S_c, another service S_2 that offers *at least* the same functionalities of S_1 should be found, so that the service S'_c, obtained from S_c by substituting S_1 with S_2, behaves as S_c. We will discuss some preliminary result on such issues in Chapter 2.

Service composition leads to enhancements of the SOA (Extended SOA [121]), by adding new elements and roles, such as brokers and integration systems, which are able to satisfy client needs by combining available services. For example, it is advocated in [121] that a service should not only be described and discovered on the basis of its interface, as it is now, but by considering also its behavioral description and quality, i.e., both functional and non-functional aspects. The Extended SOA provides also support for service

management, i.e., functionalities to monitor and verify the correct execution and coordination of composite services, in order to guarantee a certain quality.

From the above discussion, it emerges that it is common agreement that in order to achieve the SOC vision and have autonomous software applications to recognize and use services, services must export their behavioral description.

As we will see more in detail in Chapter 2, there are several research efforts towards the study of automatic composition of services. However, such efforts are concentrating on "static" aspects, such as ontologies of services, descriptions of the information the services are dealing with, and descriptions of the required inputs and produced outputs. As discussed in Section 1.2.1, we believe that, static aspects alone are not enough to precisely describe a service. On the contrary, the behavior exported by a service must also include descriptions of the process the service carries out. Such "dynamic" aspects of the behavior are crucial in fully understanding what a service does in order to be recognized and used by autonomous software applications.

1.3 Goals and main results of the thesis

Although an enormous interest is moving around services, several aspects related to service composition, and as an aside, to service description, including foundational ones, still remain to be clarified (see [89, 69] for a survey on different approaches to Service Oriented Computing).

- An agreed comprehension of what a service is, in an abstract and general fashion, is still lacking. Therefore, no general and common framework exists that contextualizes services and service composition.

- No consolidated formal definition of service composition currently exists.

- Due to the absence of a common vision, it is extremely difficult to compare the various approaches to composition. As a notable example, results on computational complexity of both the problem of service composition, and the algorithms for composition synthesis are still lacking, and this inhibits practical and commercial developments of tools for composition.

- A clear and consolidated awareness of the relations between languages and tools for describing services and composition techniques is not present.

- A consolidated characterization of an adequate set of operators for service composition is lacking, as well as, a definition and classification of possible languages for composition.

- No deep analysis of the possible types of composition, and their properties currently exists.

The aim of this thesis is to define a formal and comprehensive framework for the characterization and the theoretical investigation of the problem of service composition.

Although several papers have been already published that discuss either a formal model of services (see e.g., [40]), or propose algorithms for computing composition (e.g., [118]), to the best of our knowledge, the research done till now and reported here is the first one tackling simultaneously the following issues:

- *Presenting a formal framework where services are clearly defined and the problem of automatic service composition is precisely characterized.* In particular, we characterize services in terms of their behavioral description, modeled as execution trees, whose nodes correspond to sequences of actions executed so far and whose successor nodes represent the choices available to the client about which actions to do next. The problem of service composition amounts to find a new *composite* service (again represented as an execution tree), that realizes the client request, such that each action of the composite service is labeled with at least one available *component* service, in accordance with the behavior of such service. Intuitively, such labeling denotes the available service(s) an action can be delegated to. Although simplified in several aspects, our framework is general, comprehensive and coherent enough to accommodate various visions on services and service composition. Additionally, it is flexible and robust, so that changes in the vision on service composition can be reflected on it with few adjustments.

- *Providing sound, complete and terminating techniques for computing service composition in special but quite significant cases (both composite and component services modeled as finite state machines), and providing a computational complexity characterization of the algorithms for automatic composition.* The basic idea is to encode the composition problem (i.e., services, the client request, and some domain independent conditions) in a formula Φ expressed in specific logics, and to reduce the problem of service composition to satisfiability of Φ. Among the available logics, we resort to Propositional Dynamic Logics (PDLs), which are a family of logics used to represent and reason upon program schemas. We formally prove that a composite service exists if and only if Φ is satisfiable and the model of Φ is exactly the composite service. In other words, exploiting several properties of PDLs, we are able to compute in EXPTIME a composite service which is a finite state machine. We also

tackle the service composition problem in various frameworks, obtained by enriching the behavioral description of services and the client specification with various features. To the best of our knowledge, these are the first algorithms for automatic service composition, where services are modeled as finite state machines and the client's needs are explicitly taken into account.

- *Implementing our service composition techniques in an open source prototype tool.* From a practical point of view, standard tableau algorithms used to check satisfiability of a PDL formula are quite complex to implement. Therefore, in order to implement our service composition algorithms in a prototype tool, we resort to another family of logics, namely Description Logics (DLs). Indeed, there is a one-to-one correspondence between PDL formulas and DL knowledge bases (and therefore, between PDL models and DL models). Additionally, DL based tableau algorithms have been widely studied in the literature and several optimizations for effective and efficient implementations have been developed. However, we want to remark that we use DLs only for implementation purposes, since DLs are a family of logic used to represent *static* knowledge, i.e., that can be expressed in terms of classes and relationships between them.

Finally, note that several open issues remain to be solved and many possible extensions to our framework may be taken into account. Additionally, not all aspects highlighted above have been tackled yet. Doubtless, the research presented in this thesis constitutes a first step towards the definition of a theoretical framework for services and service composition.

The results of this thesis have been published in international journals, conference and workshop proceedings, [30, 28, 22, 21, 31, 25, 24, 32, 23, 19].

1.4 Structure of the thesis

The rest of this thesis is structured as follows. In Chapter 2 we discuss the state of the art in Service Oriented Computing and in particular wrt service composition. In Chapter 3 we first discuss our general framework where services are characterized in terms of their behavior, expressed as (possibly infinite) execution tree. Then, we provide a formal definition of the problem of service composition in such a framework. In Chapter 4 we restrict our attention to the class of services that can be represented using finite state formalisms. Then, we discuss a sound, complete and terminating algorithm for automatically computing service composition. Finally, we provide a computational complexity characterization of the algorithm. In Chapters 5 and 6

we extend the framework of Chapter 4 by considering client requests which present various forms of underspecification, and by specifying the role that a service has wrt an action, namely whether it delegates the action to other services (because it is not able to execute it) or it is delegated an action (because it is able to execute it). Then, we discuss sound, complete and terminating algorithms for automatically computing service composition, and characterize their computational complexity. In Chapter 7 we present the prototype tool we developed that implements our service composition algorithms. In Chapter 8 we summarize our work and discuss future research directions. Finally, in Appendix A we discuss an encoding of finitely representable services, which is based on Situation Calculus and is alternative to the one of Chapter 4; finally, we discuss a sound, complete and terminating technique for automatic service composition.

Chapter 2

Related Work

In this chapter we analyze the main works published in the literature that are relevant to the results presented this thesis, and that constitute the state of the art on services. First, we concentrate on service description and composition. Then, we show that several research areas (e.g., data integration, planning and reasoning about actions, workflow, software components, program and verification synthesis) present issues that are somehow related to those of services, therefore, solutions or approaches proposed in these fields can be suitably applied to help solving open issues regarding services, and in particular, service composition, from a conceptual point of view. We compare and classify all the results about service description and composition wrt a set of dimensions highlighting what has been done, which issues need to be solved, and how results from other fields can help in tackling them. Finally, we discuss the results in service discovery and orchestration, which are related to service description and composition, and on top of which the results of this thesis can be built; we also analyze the open issues they presents.

Observe that many industrial efforts have concentrated on services, therefore, we will discuss contributions from both the industry and the academia. This allows us to show that service oriented computing is supported by a rich technological substrate, that however is not suitable for service oriented computing at a conceptual level. Indeed, we argue that service oriented computing should be based on a conceptual representation of services from an external point of view, thus abstracting from internal (i.e., implementation) details; such an external point of view is the one to be considered in particular when composing services.

2.1 Service Description and Modeling

There are several ways in which a service can be described. This is especially clear when considering the enormous quantity of languages that are being proposed as standard by industrial consortia. In order to clarify the current approaches to service description and highlight the open issues, we compare them wrt the following properties, namely, their proposed *interaction model* between the service and the client, their degree of *completeness* in the specification of the service description, their degree of *observability* (of the current state), and how they model data, if present.

As fas as its interaction model, a service can be classified as *(i)* atomic, *(ii)* run based, *(iii)* tree based. An atomic service is described in terms of its operation(s), i.e., as a (possibly singleton) set of signatures with input and output parameters, and possibly preconditions and effects[1]. Atomic services are considered as "black box" entities, since they are characterized from an input/output perspective. Conversely, run based or tree based services are characterized by a non-atomic behavioral description, therefore, they are often referred to as "gray box". The difference between run based and tree based systems depends on the (conceptual) model of behavior they assume. Specifically, it is analogous to difference that exists between the model of time that is assumed by a Linear Temporal Logic (LTL) formula, and the model of time assumed by a branching temporal logic (CTL) formula, respectively. A service is characterized by a run based model, when its behavior is described by *linear sequences* of actions, each one describing a single run (i.e. computation). Therefore, in such case, each time point has a *single* successor time point. Finite state automata are an example of a run based model: they represent sets of runs (or of traces or of strings), i.e., finite length sequences of actions. Therefore a run based model could be also referred to as *language based*. On the contrary, tree based services are typical of a *branching setting*: each time point may have several successor time points, all equally plausible during execution. Therefore, such services are captured by structures which have the form of (possibly infinite) trees, each describing the behavior of a *possible* computation. Transition systems are an example of tree based model: they focus on *alternatives* encountered during runs, as each transition represents a choice point on what to do next. Therefore a tree based model could be also referred to as *transition system based*.

The behavior of a service can be captured by a complete or partially specified description. Each complete specification denotes the behavior of a *single* service. The presence of incompleteness in the description determine *a set* of services whose behavior is completely specified, in such a way to be coherent

[1]Effects can be also conditional.

with the partial description. For example, the incompleteness may regard the initial situation and/or the effects of actions: this means that after performing an action, the successor state is not always known. In other words, the knowledge (i.e., observation) of the current state is only partial: at each step of the computation, the (partially described) service is in a set of possible current states, characterized by common observable properties. Therefore, it is also interesting to study the implications due to a partial state observability.

Finally, since services handle data, in the form of input and output parameters, it is necessary to model the data (types) that a service can handle, and the flow of data (values) and control within a service.

The classification done in this section is reported in Table 2.4 in Section 2.4.

2.1.1 Industry

Recently, a plethora of languages for modeling and specifying different facets of service oriented computing have been proposed. In this section, we discuss some of them, with respect to the Service Oriented Architecture (SOA) (cf. Section 1.2.2), in order to correctly compare all such works.

First of all, note that since the transport medium is a parameter of SOA, such an architecture is easily integrable on top of different technologies. As briefly mentioned in Section 1.1, an architecture for *services* is generically based on an *electronic* transport medium, whereas for a *Web Service* architecture the transport medium consists of Web based technologies such as HTTP, SOAP and XML Protocol [146]. All emerging proposals consider the last scenario, in which services "live" over the Web, and address some basic technological issues of SOA, that is *(i)* the definition of the Web Service description language, possibly expressed in terms of the interactions a Web Service can be involved in (i.e., its behavior or conversations[2]), *(ii)* the definition of the Web Service directory, and *(iii)* the definition of how to compose and coordinate different Web Services, to be assembled together in order to support complex processes (i.e., the orchestration). In what follows, we concentrate on the first point. The remaining ones will be tackled in the next sections.

In the context of the W3C, many Web Service description languages are being proposed for specific purposes:

- Web Service Description Language (WSDL, [6]) for describing Web Services, in terms of their static interface;
- Web Service Conversation Language (WSCL, [77]) for describing the conversations a Web Service supports; its underlying model are the activity diagrams;

[2]Usually, the term conversation is used when the behavior is expressed as sequences of message exchanges.

- Web Service Choreography Interface (WSCI, [12]) for describing the observable behavior of a Web Service, i.e., as seen by its clients, in terms of temporal and logical dependencies among the exchanged messages.

With respect to the SOA, the languages WSDL, WSCL and WSCI concern the service provider, since he exploits them to describe the Web Service he wants to publish.

WSDL, analogously to an interface definition language (e.g., CORBA IDL), describes (the set of) operations it offers, ingoing and outgoing messages, and data types used by the Web Service which are defined in terms of XML Schemas. Moreover, it supplies a mechanism to locate the Web Service (e.g., using a URI), a protocol to exchange messages and the concrete mapping between the abstract method definition and the real protocol and data format. WSDL defines what the Web Service does, not how it does it. Finally, as briefly mentioned in Section 1.2.1, WSDL (Version 1.1) does not express the semantics of exchanged messages, neither their correct order. Additionally, WSDL does not currently support operations for monitoring Web Services such as checking the availability of an operation or the status of a submitted request. Recently, a first working draft of WSDL Version 2.0 has been published [48]. The most important changes concern *(i)* the definition of message patterns for the exchange of messages, which can be seen as a first step towards the association of semantics to message exchanges, and *(ii)* the support of inheritance (but not of overloading) between operations. It is very likely that other constructs will be added in future working drafts, for example for supporting composition.

WSCL models the conversations supported by a Web Service: it specifies the XML documents being exchanged, and the allowed sequence(s), in a fashion similar to an activity diagram. A WSCL document is composed of four main elements: *(i)* document type descriptions specifying the types (schemas) of XML documents the Web Service can accept and send during the conversation, *(ii)* interactions modeling document exchanges between two participants (i.e., the Web Service and its client), during the conversation, *(iii)* transitions specifying the order relationships between interactions, and *(iv)* conversations, listing all the interactions and transitions that make up the conversation.

WSCI can be considered as a evolution of WSCL: it describes the observable behavior of a Web Service and how operations can be choreographed in the context of message exchanges in which the Web Service participates. WSCI allows to describe the correct order of the exchanged messages and permits to define multiple behaviors for the same Web Service on the basis of the context in which is used. Furthermore, it provides methods to manage exceptional situations, such as timeouts and fault messages. WSCI, finally, allows to define the abstract behavior of a process that involves more Web

Services in term of interfaces and links between operations, thus giving a view of the process in terms of message exchanges; with such a respect, WSCI is also an orchestration language.

From the above discussion, it is clear that WSDL addresses only static interface specifications, whereas WSCL and WSCI consider also behavioral issues. Therefore, according to our classification, WSDL considers an atomic interaction model for Web Services, while WSCL and WSCI consider a tree based model, since they describe the Web Service behavior as the tree of all possible computations. Additionally, since WSCL and WSCI are industrial languages, they allow only for completely specified description of Web Services, and therefore, at each point of execution of WSCL and WSCI file, the current state is completely known. Finally, all of them explicitly handle data, in the form of ingoing and outgoing messages, that define the schemas of the XML documents being exchanged.

Also Business Process Execution Language for Web Services (BPEL4WS, [4]) and Web Service - Choreography Description Language (WS-CDL [90]) are languages for describing services. However, they are more targeted towards specifying how multiple services are coordinated and the state and logic needed for such coordination, therefore, we will address them in Section 2.5.2.

2.1.2 Academia

The **OWL-S** (formerly DAML-S) Coalition [5] defines a specific ontology and a related language for Semantic Web Services (SWS[3]), called OWL-S. Specifically, the ontology enables the definition of SWS content vocabulary in terms of objects and complex relationships between them, including class, subclass relations, cardinality restrictions, etc., and including all XML typing information. More in details, the SWS Ontology comprises:

- *Service Profile.* It focuses on what a SWS does, since it describes properties of a SWS necessary for automatic discovery, such as its capabilities, and its inputs, outputs, its preconditions and (possibly conditional) effects.
- *Service Model.* It focuses on how a SWS works, since it describes the SWS process model, i.e., the control flow and data-flow involved in using the SWS. It is designed to enable automated composition and execution of SWSs.
- *Service Grounding.* It focuses on how to access a SWS, in terms of communication protocol, marshalling and serialization. In particular, it

[3]They are Web Services equipped with machine-interpretable semantics.

connects the process model description to communication-level protocols and message descriptions in WSDL. Up to now, the Service Grounding has not been addressed yet.

Although the specifications of the SWS ontology are continually evolving, the tendency is to adopt a rule-based language for expressing Profile; as far the Service Model, two competing approaches, namely FLOWS and SWSL-Rule, that propose a Situation Calculus based language and a first-order rule based language, respectively. Such models have a quite high expressive power, therefore, they are not suitable for automatic synthesis: however, this is in line with their aim, which is only to fully characterize a SWS [74].

Several interesting works build upon SWS architecture (see, e.g. [85, 63]). In particular, in the Semantic Web Service literature, we want to mention the results in [118], where the authors provide a Situation Calculus based axiomatization of the Service Profile and provide an operational semantics characterization of SWS execution using Petri Nets. In [108, 107] **McIlraith and her group** propose a Situation Calculus based framework for services, where a SWS is described as a Golog/ConGolog [73] procedure, seen that the client sees as an atomic action presenting an input/output behavior, which is expressed in OWL-S in terms of its Service Profile. Services are stored in an OWL-S ontology of services.

According to our classification, SWSs provide an interaction model which is both atomic (according to the Service Profile) and based on a behavioral description (according to the Service Model). Due to the continuous evolution of the Service Model, it cannot be stated whether it is run or tree based, at the moment in which this thesis is being written, neither whether SWS exports a complete specification, or an incomplete one.

Note also that, to the best of our knowledge, all the works on SWS description, discovery and composition of SWSs focus on the Service Profile, without considering the process model. Therefore, it is not clear whether the process model is somehow exported to the SWS client, i.e., it can be accessed *directly* by him, or instead, it is internal to the SWS.

Many works in the literature consider an atomic interaction model to describe services [89, 122, 112, 57], that export a complete specification.

In [153, 120, 119] **Papazoglou and his group** propose to wrap (simple or complex) pre-existing services into a web component, so that their operations can be offered through a uniform interface. The authors consider a web component as a class of an object oriented language, that can be extended, reused, and specialized. These components are specified in the language SCSL (Service Composition Specification Language) and are stored in a common library.

Several works focus on information gathering services, whose execution is based on an underlying database. In particular, we focus on the following ones, that we consider significative.

In [54] **Vianu and his group** consider a service as a data-driven entity characterized by a database and a set of web pages. At each step of the computation, i.e., when the client is on a specific web page, a set of input choices are presented to him: some of them are generated as queries over the database; a fixed set (of input choices), representing specific client data, are treated as constants, whose interpretation is provided by the client during the run of the service, i.e., it is not fixed a priori. The client chooses one of such inputs, and in response, the service produces as output updates over the service database and/or performs some actions, and makes a state transition, which is seen by the client as a transition from a web page to another. Initially, the client starts from the home page. A service is therefore seen by the client as a tree of web pages, whose structure is completely specified in terms of a relational schema, constituted by a set of tuples that dynamically depend on the service database, the current state, the available actions and input choices. Also the service database, states, inputs, and actions are modeled as relational schema. The proposed model of services is based on Spielmann's work on Abstract State Machine (ASM) transducers [136, 137].

In [139, 141, 140] **Ambite, Knoblock and Takkar** consider services as views over data sources. They build on the idea that heterogeneity of data sources may be overcome by exploiting services as wrappers of different information sources, thus providing uniform access to them, exploiting standard protocols such as SOAP and XML. Each data source, i.e., service, is described in terms of input and output parameters (the latter provided by the source), binding patterns and additional constraints on the source. The latter allow to characterize the output data, e.g., that a certain data source returns restaurant in a given area.

Recently, it is advocated the need to characterize services in terms of their behavioral description [69]. From the survey in [89], it stems that most practical and currently adopted approaches for modeling behavioral descriptions of services, especially those targeted towards composition, are based on finite automata/state machines.

In [40], **Hull and his group** model a (completely specified) service (called peer) is as a Mealy machine, that exchange messages with other peers according to a predefined communication topology. Each peer receives messages belonging to its set of input messages and sends messages belonging to its set of output messages. In general, communication is asynchronous, since each peer is equipped with a bounded queue which is used to buffer messages that are received but not yet processed. Possibly, the queue of each peer may have length zero: in this case messages are exchanged in a synchronous way. At

each step, a peer can either *(i)* send a message, or *(ii)* receive a message, or *(iii)* consume a message from the queue, or *(iv)* perform an empty move, by just changing state. Therefore, the first set of results focus on a run-based perspective.

In [123] **Traverso and his group** define a framework where available services are partially specified, and the degree of observability on the current state varies from full observability (i.e., the current state is completely specified), to null observability (no information on the current state is available), to partial observability (only partial information is available). Each service is represented as a non-deterministic finite state machines, characterized by a set of initial states and by a transition relations that defines how the execution of each action leads from one state to a set of states. Because of incomplete knowledge on the initial states and on the outcome of actions, at each computation step, each service could be in a set of states, each equally plausible given the initial knowledge and the observed behavior.

In [17, 10, 15, 16], **Benatallah, Casati, Toumani and their group** model the behavior of a service in terms of its conversations with the clients, i.e., as the tree of message exchanges, on which order constraints among messages are imposed. In [17, 10] the (completely specified) behavior of a service is represented as a deterministic finite state machine, where transitions are labeled by operations, and states with the status of the conversation, for example the effect of the operation leading to it (if clearly defined). The initial state is labeled by “Start”. Some transitions are not labeled with operations since they model situations when the evolution of conversations from one state to another is not caused by an (explicitly invoked) operation, but by a (predefined) event, e.g., by time elapsing. The authors denote by conversation schema the specification of a conversation, and by conversation instance a particular execution of a conversation. Additionally, the authors further characterize the transitions, by specifying with human readable description (e.g., XML based) when they occurs and their implications (e.g., transactional semantics). In this way, one can easily denote precondition, effects of a transition, temporal constraints, policies for transition compensation, etc. In [10] the authors show how to automatically generate the skeleton of a BPEL4WS specification starting from the finite state machine modeling the service conversation, thus guaranteeing that the service implementation conforms to a given conversation specification. Note that the authors focus on producing a BPEL4WS skeleton containing the service implementation logic, rather than information on which services are actually invoked: such bindings will be manually defined by the service designer in a second moment. In [16] the same authors present a slightly different model for the service behavior (again considered in terms of its allowed conversations), that is represented as a possibly non-deterministic finite state machine, where transitions are labeled with either an input or an

output message. Non-determinism is due to the fact that the service may behave differently in different situations, on the basis of internal business logic which is not exposed to the clients. Therefore, non-determinism is caused by partial specification of the exported behavior. By unfolding in all possible way the finite state machine associated with a service, one obtains a conversation tree, which defines the set of all possible conversations that service is involved in. In other words, the authors propose a branching structure as the underlying model for conversations: at each step, the client has to make a choice on which message to send (if any).

In [128, 35] **Bordeaux and Salaün** propose to describe tree based service behavior using Process Algebras, which are formal languages used to model dynamic entities, using an approach based on transition systems. In this way, they associate clear operational semantics to behavior of services and are able to automatically verify desired properties of them, such as absence of deadlocks and starvations between interacting services. Several Process Algebras have been proposed, each one addressing specific features of processes: LOTOS [34] can be used to model data, Timed CSP [134] captures temporal constraints, π-calculus allows to model mobile processes, i.e. processes that communicate with other processes that change dynamically. The Process Algebra they focus on is CCS [116], which is the simplest one, yet equipped with all the operators needed to represent services, and on top of which all other process algebra have originated. In particular, they show how to model both a choreography and an orchestration scenario in CCS. The authors also show how to map a CCS specification of a (composite) service into a BPEL4WS file (skeleton). In [129] the authors continue to show the usefulness of applying process algebras to services by showing how to capture in LOTOS a framework for service negotiation.

In the framework we develop **in this thesis**, we study services exporting a complete[4] specification of their behavior, represented as the tree of all possible actions that each service can execute. At each step of its execution, the service presents its client with a choice on the next action to perform: the client selects one, according to its goal, and the service executes (the computation associated to) it. Then, it presents again the client with a new set of actions. We do not explicitly model data in our framework, even if it can be easily and naively encoded e.g., in the state (see Section 3.3.2)

[4]Note that, in general, the notion of "complete" is relative to the abstractions taken in the model. Specifically, the model in this thesis is "complete" wrt to actions.

2.2 Service Composition Synthesis

In this section we consider again the works presented in the previous section, which are targeted towards the problem of service composition synthesis. We recall that given a set of available services and a client request, the problem of service composition synthesis is concerned with synthesizing a new composite service that realizes the client request, by suitably coordinating the available services. For each of the works, we discuss how they tackle such problem by focusing in particular on *(i)* how the client request is modeled, *(ii)* the kind of composition, *(iii)* the referred architecture for orchestration.

The client[5] request denotes somehow the specification of the composite service he desires to interact with in order to achieve his goal. Therefore, a client request presents the same features of a service specification, i.e., it can be characterized according to its *interaction model* wrt the service, its degree of *completeness*, its degree of *observability* (of the current state), analogously to what discussed in Section 2.1. The client request is *atomic* if it specifies the signature of the service he would like to have realized, expressed in terms of input (values), output (data types), possibly preconditions and effects. The client request can also be expressed in terms of the actions that the composite service should execute, by specifying them in terms of properties over either its *runs* or its *tree*. In the first case, the client specifies (a set of) linear sequences of actions: at each point of the computation the future is uniquely determined on the basis of the properties expressed in the client request, therefore the composite service is executed with no intervention from the client, since the client is guaranteed that the service computation that will be executed achieves his request. If the client request denotes the (execution) tree of the service, it is actually specifying a set of possible futures for each point of the computation. Therefore, the composite service must behave correctly for any such possible future. Note that during execution, the client chooses which is the unique possible future, among those possible. Run based and tree based client specifications are expressed, respectively, using linear based and branching based formalisms, whose expressive power is not comparable: for example, branching based formalisms, such as branching temporal logics, cannot express properties over runs and linear based formalisms, such as linear temporal logics, cannot state features of trees. Finally, note that, in general, client request and composite service do not necessarily share the same interaction model, neither do composite and component services for example, an atomic client request can be realized by a run based composite service, however, the client is not aware of it, since the coordination of component services is completely hidden to the client.

[5] In general, the client can either be a human user or another service. Here, for simplicity, we refer to the client with the pronoun "he".

The client request can be either a complete or a partial specification. As usual, a complete specification denotes the single service the client would like to have realized, while the partial specification expresses a set of possible services, any of which realizes the client request. Note that from a practical point of view, it is more common to find client requests which are partially specified rather than complete specifications: therefore, it is interesting to consider how a partial client specification impacts the algorithm for automatic service composition synthesis. Again, in the latter case, it is also worthwhile analyzing the consequences of having a partial knowledge (i.e., observation) of the current state.

Now we consider the various ways in which component services can be arranged in a composite service, and therefore, are coordinated in the orchestration phase. The simplest case is when services are *sequential*ized: the execution of one component service can start only after the previous one has completed. Note that also data and control flow present a linear structure: the input of a service may depend on (either coincide with or be some aggregation of) the output returned by previous services. More challenging it is the case when component services are interleaved and *concurrent*ly executed, i.e., action a of service E is executed, followed by action b of service F, followed by action c of E, etc. Of course, also data and control flows are more complex, since several services are simultaneously active and concurrently running. Typically, a sequential composition is associated to atomic services, while concurrency is a characteristic of run-based or tree based services. However, it may happen that services whose description is atomic but whose execution is not (i.e., its execution consists of several actions, which are hidden to the client) are coordinated in a concurrent way while run or tree based services are sequentially coordinated.[6]

When component services are partially specified, two other forms of composition can be identified, namely *conformant* and *conditional*. A conformant composite service is constituted by a set of action sequences or a tree of actions that realizes the client request regardless the partially specified component services. In other words, the composite service realizes the client request, for all possible ways the component service actually behave during the coordination, i.e., for all possible initial states in which the composite service can be, and for all possible effects of actions. Conversely, a conditional composite service deals with partial specification of component services by gathering information on them at run-time, i.e., by performing some sensing activities. It is also characterized by a structure of the form "if-then-else" (i.e., if it is represented as a FSM, this is equivalent to have guard conditions on transitions):

[6] Another interesting issue that is worth mentioning, and that has not been tackled yet in the service literature, is the *spawning* (see, e.g., [88]) of services, i.e., the situation when several instances of a service are activated and executed by interacting one with the other.

at run-time, it is sensed the value of the "if-condition": if it is true the "then" path is followed, the "else" otherwise. The composition can be obtained either manually by a human, or automatically, by exploiting a tool that implements an algorithm, in our survey we consider also this aspect.

Since the specification of the composite service is then orchestrated by some engine, such a specification should also consider whether the architecture for orchestration, is *peer-to-peer*, i.e., distributed, or *mediated*, i.e., centralized, as identified in [89]: in the first approach the individual services interact among themselves and with the client directly, while in the second the control over the available services is centralized. In Section 2.5.2 we will further tackle such aspect wrt orchestration engines.

Finally, in order to achieve correct coordination among component services, the composite service should handle the flow of data (values) and control. Therefore, we will analyze how the various proposals address this point.

The classification done in this section is reported in Table 2.4 in Section 2.4.

2.2.1 Industry

Interestingly, to the best of our knowledge, very few efforts from the industry address the problem of composition synthesis, and all of them have in common the development of tools for *manual* composition of services where all component services are represented as WSDL, and therefore assume an atomic interaction mode. Indeed, the industry focuses more on how to orchestrate (manually obtained) specification of composite services, expressed in languages such as Business Process Execution Language for Web Services and Choreography Description Language, as it will be clear in Section 2.5.2.

2.2.2 Academia

In this section, we describe the various approaches to service composition from academia. We focus first on those works considering atomic services, and then on those addressing services that export behavioral descriptions.

In [112] **Bouguettaya, Elmagarmid and Medjahed** present a framework for composing atomic services, which are semantically described in terms of non-functional properties such as their purpose, their category and their quality. Such properties define a taxonomy of services. Then, the authors define a composability model, for comparing syntactic and semantic features of services, in order to check if they can be composed. Syntactic features depend on *(i)* the interaction protocol supported by each service, and on *(ii)* their binding protocols; semantic features depend on *(i)* comparison of the number of message parameters, their data types, etc., *(ii)* the semantics of service operations, i.e., their purpose and category; *(iii)* qualitative properties, and *(iv)*

composition soundness, which checks if combining services actually provides an added value service. On top of these notions the authors develop their algorithm for composition. The the client request is expressed in the Composite Service Specification Language, and (completely) specifies the sequence of desired operations (including their input/output parameters and semantic description) that the composite service should perform. The client may also specify the control flow between operations, but cannot impose which is the composite service(s) that will execute the operations. Thus, given the client request and a set of available atomic services, they develop a technique for (semi-)automatic composition, and show a prototype implementation of it. Note that the composite service, which is a sequence of operations, is obtained from the client specification by identifying, for each operation, the operation(s) of component services that matches it, on the basis of their syntax and semantic features. In general, several composite services exist, that realize a client request by coordination of a set of component services, which are ranked according some defined quality criteria. Finally, the composite service is translated into an orchestration language that is based on a mediated architecture.

In [153, 120, 119], **Papazoglou and his group** address the issue of service composition, where services are atomically represented as web components. As a composite service is constituted by a set of (possibly composite) services, also a web component can be constituted by other web components, manually "glued" together by the so-called composition logics. The authors have also developed the language SCPL (Service Composition Planning Language), for representing a composite service, by specifying relations among web components, in terms of execution order (either sequential or concurrent) of web components within composition, or dependencies among input and output parameters. In [154] a methodological framework for service composition and life-cycle management is also proposed, in which composite services are created by re-using, specializing and extending existing ones.

In [139, 141, 140] **Ambite, Knoblock and Takkar** propose data integration techniques to dynamically compose information retrieval, atomic services. The composition algorithm takes in input the set of available services modeled as data-sources, and a user query, expressed in terms of inputs provided by the user and requested outputs. The output is a new, composite service that can execute an integration plan for a *template* query, so that all the user queries that differ only for intensional input values can be answered by the same (composite) service. They adopt a mediator based framework. First, the specific user query is *generalized*, associating each specific user input (parameter) with its class: this is done by exploiting attribute level ontologies. In order to reformulate the generalized user query into the source queries, the mediator constructs an integration plan consisting of a sequence of source queries and

taking binding pattern into account. In [141] the integration plan is generated with a forward chaining algorithm; in [140] the authors implement an extension of the Inverse Rule Algorithm and map the produced datalog program into the integration plan. Then, the mediator optimizes the integration plan using a data flow analysis algorithm, to remove unnecessary source queries from the generated plan. Finally, the mediator utilizes source constraints and other services providing sensing functionalities to filter out the data at the tuple level, that do not meet the source constraints.

In [108, 107], **McIlraith and her group** present a tool for automatic composition of services represented as Situation Calculus actions, and encoded in OWL-S. Such tool works as follows: a user presents his request to the system, expressed as a kind of generic (i.e., skeleton) ConGolog [73] procedure with user constraints and preferences. Such a user specification cannot be executed "as is": it should be made executable by an agent that, exploiting an OWL-S ontology of services, automatically *instantiates* the user specification with services contained in such an ontology, by possibly pruning the situation tree corresponding to the generic procedure in order to take user preferences and constraints into account. Note that the ConGolog generic procedure is actually associated with a situation tree (i.e., a kind of process flow in the theory of Situation Calculus), which denotes a partial specification of the behavior of the desired composite service. Each node of the situation tree denotes a situation, i.e., a snapshot of the (desired) service configuration at each point of its execution. When a component service is executed, due to incomplete knowledge on its effects, it is not known the resulting new situation, since several situations can equally be possible. The actual (single) successor situation is defined by executing knowledge-gathering services. Therefore, the instantiated user specification is a (linear) sequence of atomic (world altering and information gathering) services which are then executed by a ConGolog interpreter. Note that in [107, 108] the user request is specified once and for all before the composition and during the execution of the composite services he has no control on the executed sequences of actions. Finally, in [108] the outcome of the composition is not an OWL-S service, and it cannot be re-used by another user, since it is an instantiated sequence of services satisfying the goal of a particular user.

In [40], **Hull and his group** present a framework for modeling and analyzing the global behavior of service compositions. Services exchange messages according to a predefined communication topology, expressed as a set of channels among services: a sequence of exchanged messages (as seen by an external virtual watcher) is referred to as conversation. In this framework properties of conversations are studied, in order to characterize the behavior of services, modeled as Mealy machines. Then, the authors tackle the problem of service composition synthesis as follows. The input to the synthesis problem are *(i)*

a desired global behavior (i.e., the set of all possible desired conversations) specified as a regular language[7], and *(ii)* a composition infrastructure, that is a set of channels, a set of (name of) services and a set of messages. The output of the synthesis is the specification of the Mealy machines of the services such that their conversations are compliant with the specification expressed by the regular language. Note that the conceptual model underlying the desired composition specification: a linear setting is taken, since composition focuses on linear sequences (i.e., paths) of actions.

In [2, 123, 100], service composition is addressed by using planning under uncertainty, model checking and constraint satisfaction techniques. In [2, 100] Aiello, Traverso and their group propose a request language, to be used for specifying client goals. The client request is modeled as a branching temporal formula, expressed in the EaGLe goal language [99]: it is an extension of CTL, that allows to express goals of the form "Try to achieve condition c and, in case of failure do achieve condition d" [99]. Therefore, the client request expresses a complete specification of his requirements, essentially by specifying a main execution to follow, plus some side paths that are typically used to resolve exceptional circumstances. In [123], **Traverso and his group** propose a composition algorithm, that takes in input a set of partially specified services, modeled as non-deterministic finite state machines, and the client goal expressed in EaGLe, and returns a plan that specifies how to coordinate the execution of concurrent services in order to realize the client goal. The plan can then be encoded in standard coordination languages and executed by orchestration engines. Note that once the plan is synthesized, the client *does not* further interact with it in order to choose what to do next, therefore, the interaction model is run based. The plan is also able to monitor the composition, in particular that the available services behave consistently with their (partial) specifications. The authors do not provide any complexity characterization of their approach, however, due to the presence of non-determinism and partial state observability, the search space explodes with very simple available services (i.e., with few states). In order to overcome this problem, symbolic model checking techniques are used to generate a plan, since they compactly represent the search space, and efficient heuristics are implemented to avoid generating the whole search space. Additionally, in [100] Aiello and his group propose an approach to service composition based on interleaving planning, monitoring and execution: in this way, the authors are able to adapt at runtime the composite service generated during the planning phase, to cope with possible changes in the service environment.

In **this thesis**, in order to address composition in an automatic way, we

[7] In [89] the authors mentions that it would be natural also to use LTL (or other temporal logics) to specify the desired global behavior.

specify our framework where services are modeled as (possibly infinite) execution trees, to the case where they admit a compact representation as deterministic finite state machines. The client request denotes the desired service he would like to interact with, i.e., as a tree of actions, finitely represented as finite state machine. Our composition algorithm works as follows. Given a set of available services and the client request (all of them represented as finite state machine), our algorithm finds a labeling of the tree associated to the client request, such that *(i)* each action is labeled with (i.e., delegated to) available services and *(ii)* each possible sequence of actions on the labeled tree corresponds to possible sequences of actions of the available services, suitably interleaved. We addressed and solved the problem of composition when the client request is either completely or partially specified. We also developed a tool that implements our algorithm. The works in the literature that present some similarity with ours are those in [107, 108, 89, 40, 2, 123]. The main difference between the work in [107, 108], and our technique is that services are atomic actions, therefore the client can not specify the interleaved execution of "pieces of" services (i.e., parts of atomic actions/procedures). Another difference is that in [107, 108], the client specifies his goal once and for all before the composition and during the execution of the composite services he has no control on the executed sequences of actions. Conversely, in our work the client has such control, since at each step of the execution he chooses the next action to perform. Finally, in [108] the outcome of the composition, which is a linear sequence of atomic services, is not a service, in the sense that it cannot be re-used by another client, whereas in our work the composition produces a reusable specification. The main difference between the approach in [89, 40] and our technique is that their approach to the synthesis is "top-down": a desired global behavior is specified, and it is assumed that services can be *designed* during the synthesis phase without constraints. Conversely, our technique is "bottom/up": the behavior of the services is also given, and the synthesis phase tries to *assemble* such behaviors in order to provide the desired behavior. Another difference consists in the conceptual model underlying the desired composition specification: in [40] a linear setting is taken, since composition focuses on linear sequences (i.e., paths) of actions; conversely, in our approach the client specification is based on a branching model: composition focuses on a tree-based structure, where each node denotes a choice point on what to do next. We recall that the expressive power of linear and branching temporal formulas is not comparable. In [123, 2] the client goal essentially specifies a main execution to follow, plus some side paths that are typically used to resolve exceptional circumstances, whereas in our approach all possible executions are (equally) considered.

Tightly related to service composition there are the issues of compatibility and substitutability. Actually, they can be seen as simplified variants of the

problem of service composition synthesis.

In [36] the authors propose various notions of compatibility and of substitutability of two services, represented by their behavioral descriptions. They start from the basic intuition that two services are compatible if they can interact properly. A first, strong notion tightly relates compatibility to behavioral equivalence: two services are compatible if they have "mirroring" behavior, i.e., at each step, whenever a service can send a message, the other can receive it and vice-versa. A slightly relaxed definition of compatibility requires that each service is ready to receive at least all the messages that its mate can choose to send, and possibly more. Finally, according to the weaker notions two services are compatible if there is at least one possible way in which the two services can interact, starting from their initial state and ending in their final states.

In [16] Benatallah, Casati and Toumani analyze when two services, modeled as transition systems that exchange messages, can correctly interact. First, they formally define several operations between the protocol (i.e., the behavior) of two services, say S_1 and S_2: *(i)* compatible composition, that defines the tree of all possible message exchanges (i.e., sent by S_1 and received by S_2 and vice-versa) between them; *(ii)* intersection, that defines the tree of all messages that can be either sent or received by both S_1 and S_2; *(iii)* difference, that defines the tree of all messages that can be exchanged by S_1, but not by S_2; *(iv)* projection over one service, say S_1, which identifies the behavior of S_1 wrt the protocol defined in terms of previous operations, e.g., the projection over S_1 of the compatible composition between S_1 and S_2 identifies the tree of messages of S_1 that are correctly exchanged with S_2. Then, making use of such operations, the authors define several classes of compatibility and substitutability, which are similar to those reported in [36] and independently obtained.

Related to service composition is also the analysis and verification of composite services, motivated by the dynamic flavor of composition, the consequent difficulties in testing and immaturity of service oriented development environments and methods. Preliminary results can be found in [68], where verification of BPEL4WS specifications is carried out exploiting model checking techniques, in [118], where OWL-S services are analyzed exploiting Petri Nets. In [54] Vianu and his group study how to automatically verifiy properties of inputs, actions, and states, of data-driven services, that are defined both over runs, i.e., that all runs satisfy certain conditions, and over sets of runs, i.e., that for every run with certain properties, some conditions hold. In the first case, the authors express the properties to be checked as variants of linear-time temporal logic, in the second case they use variants of branching-time logics. They characterize the complexity of verifying such properties for various classes of service with different expressive power. We want also to remark that

analysis and verification are more effective when composite services are *manually* synthesized; our technique *automatically* synthesize a composite service that is correct by construction, as proven in Sections 4.3, 5.3, 6.4, and 7.3.2.

Finally, it is worth mentioning the work in [44], where the authors show that service composition can be exploited to easily solve the task of record linking. Record linking, also referred to as object consolidation, is a data and computationally intensive task aiming at quickly and accurately identifying common entities in disparate data sources. Conceptually, they extend a standard data integration system with service operations and abstract representation of processes. On top of it, they develop a prototype tool that executes a workflow containing suitable calls to (atomic) services, thus implementing an algorithm for record linking.

2.3 Related Research Areas

The theoretical investigation of service composition synthesis has been explicitly or implicitly addressed in various forms by several research areas, specifically: *(i)* Data Integration, *(iv)* Artificial Intelligence, and in particular, Planning and Reasoning About Actions, *(v)* Theoretical Computer Science, and in particular, Program Synthesis and Verification, *(iii)* Software Engineering, and in particular, Software Components.

In the following subsections, we show interesting bridges between each of the mentioned research fields and services.

Finally, for sake of completeness, we want to mention that also the research field of Workflows has produced several results (and tools) on which service orchestration is founded (see Section 2.5.2).

The classification done in this section is reported in Tables 2.4 and 2.4 in Section 2.4.

2.3.1 Data Integration

Several works in the literature address services and service composition from a data integration perspective, since they claim services mainly produce data. Data integration aims at providing a uniform access to a set of autonomous and heterogeneous data sources, thus freeing the user from having to (manually) locate sources which are relevant to a query, interact with each one in isolation, and (manually) combine data from multiple sources. By substituting the term "(data) source" with service, one can understand how in fact service composition and Data Integration have much in common.

The basic idea is therefore to consider a service (for gathering information) as a query over a data source. Specifically, each service is modeled as an atomic operation with query/results as input/output parameters. As for

data integration systems, users pose queries in terms of a mediated (or global) schema, that contain (virtual) services. Service composition can be thus reformulated as a query rewriting problem (see [80] for a survey), where the query encoding the client request, which is posed over the mediated schema, has to be re-expressed in terms of (the query associated to) the component, available services. In order to solve the problem, one may take a query optimization approach, and focus on producing a query execution plan that involves the services (as in [139, 141, 140]): in this case, it is necessary that rewriting the query using the services is an *equivalent* rewriting, to guarantee correctness of the query execution plan. Alternatively, one could take a data integration approach, and focus on translating user queries formulated in terms of the virtual services of the mediated schema into available services. In the first case, the outcome of composition is the specification of how to orchestrate the component services, in order to answer the user query, in the second case, the outcome is a composite service, again represented as a query. In the latter case, where instead of services one deals with queries over data sources, the data sources may not entirely cover the domain, therefore, sometimes a *contained* query rewriting is looked for, rather than an equivalent one. Additionally, the work on data integration consider both the case in which the local views are *complete* (i.e., contain all the tuples in their definition) and the case where they are *incomplete* (as is common when modeling autonomous data sources). It is interesting to study how to apply such notions, typical of data integration, to the service setting.

Finally, a mapping should be defined between each available service and the virtual services in the global schema. Several approaches can be considered, depending on whether: *(i)* the virtual services of the global schema are expressed in terms of the available services; *(ii)* the available service is expressed in terms of the (virtual services of the) global schema; *(iii)* both points *(i)* and *(ii)*. Note that all of them denote well known approaches to mapping [101] in Data Integration *(i)* is called global as view (GAV), *(ii)* is called local-as-view (LAV) and *(iii)* is called GLAV, since it combines the two. Note that, in general, a GAV mapping should be adopted in a fairly static environment, since whenever a local source or component services is modified or a new one is added, the global schema needs to be re-designed. However, query rewriting (and therefore service composition) is in general easier since can be based on some sort of unfolding (the views over the global schema into the views over the sources). Conversely, a LAV mapping provides a highly modular and extensible approach to query rewriting, since (if the global schema is well designed) whenever a source or a component service is modified, only its definition is affected, therefore it is suitable for highly dynamic environments. However, the process of query rewriting (and therefore of service composition) is complex. In [101] another approach to map component services and the

services in the global schema is presented, namely P2P: it deals with the situation where no global schema is present and mappings are defined between each pair of local sources. In the service context, it means that mapping is defined a between two services. This leads towards a distributed approach to service composition, which is very interesting to investigate.

Finally, note that from the discussion of Sections 2.1 and 2.2 all proposals that focus on services that export run based or tree based behavioral descriptions, do not explicitly model data. We argue that one can fruitfully exploit data integration techniques to take data into account when modeling services, and therefore, to automatically generate the specification of data flow among component services when (automatically) computing composition.

2.3.2 Program Synthesis

The area of program (or process) synthesis deals with synthesizing (the model of) a program P such that it satisfies a certain specification S, where both P and S are expressed in a suitable formalism. Tightly related to it, is the problem of program verification, that given P and S, deals with verifying if P satisfies S. In a framework for service composition, S may correspond to the client request and P to the composite service that should be synthesized.

Such problems have been widely studied in the literature (see, e.g. [82, 95, 94, 96, 93, 97, 60]), and several variants exist. A first variant depends on whether the model of P denotes a complete or partial specification: in the first case, the sets of input and output to P are known, in the second case, the input is (possibly partially) unknown. Therefore, P has to behave correctly wrt S independently of such events. The presence of unknown input corresponds, for example, to unpredictable events that the environment can generate, such as, in a service setting, to network failure, or exceptions resulting from the execution of component services. A second variant depends on whether the behavior of P depends only on the set of its states, or also on the set of interactions with the environment, which in general is not known. The first case is denoted as “closed world”, the second as “open world”. The latter situation is typical of services, that live in a dynamic and highly interacting environment. Finally, a third variant depends on whether underlying model of execution is linear or branching time, as already discussed at the beginning of Sections 2.1 and 2.2. In the first case, S is described by linear temporal logic formulas (LTL), in the second case by branching temporal logic formulas (CTL, CTL*). Note that when a program P is synthesized starting from an LTL formula ψ, it is required that *all* paths of P satisfy ψ. Therefore, it is not possible to express the *possibility* that ψ holds in some paths only (and does not hold in other): only branching time temporal logic can state such requirements.

Several results exist in the program synthesis (and verification) literature that address situations obtained by combination of the above features, making use both of finite state machines and model checking techniques (among the above citations, see, in particular [95, 94, 93, 60]).

With respect to this classification, we have tackled a setting for service composition which is complete, branching time and open, where the environment is represented by the client, since we don't know the choices that the client will make at execution time.

2.3.3 Planning and Reasoning About Actions

Interestingly, most approaches to service composition comes from AI community (e.g., [107, 3, 1]), where services are considered as atomic actions with input/output parameters, and agent-based technologies and planning techniques, supported by domain ontologies, have been advocated as basic tools for action and process composition.

Indeed, service composition can be seen as an advanced form of Planning.

In fact, as in Planning, the problem we are solving is the *synthesis of a program* of a specific form, which however is not a simple sequences of actions as in traditional planning, but more general execution trees obtained by reusing in a suitable way fragments, which in general are not atomic, but are themselves *sequences or trees* of actions. Also, as in Planning, we have a *goal* that we want to realize, which, however, is not a reachability goal, but the realization of a (linear or branching) behavior as specified by the client. Note that in classical planning goals expressed a condition over a single state, however, recently, many works are addressing temporally extended goals [9], i.e., that are satisfied over sequences of states. As in Planning, we have *constraints on the object that we synthesize*, that, however are not operators over actions, but are (component) transition systems that, suitably orchestrated, realize the goal. It is interesting to notice that the notion of plan composition has also arisen in planning, e.g., in [65], where a planning approach is presented, based on fulfilling the goal by suitably selecting the right plan from a plan library. There, the plan was only selected from those in the plan library, and possibly customized to achieve the current goal.

Recently, many works on planning and reasoning about actions have studied how to generate (either conditional or conformant) plans in presence of incomplete information in the initial situation and on the effects of actions [72, 124, 126, 43], possibly also dealing with partial state observability [33]. Such techniques can therefore be profitably applied to synthesize composite (conditional or conformant) services when only a partial description of the component services is available, and to analyze the implications of partial state observability.

2.3.4 Software Engineering

The idea of distributed computing over public networks, on top of which the service oriented computing paradigm builds, has been around for some decades, giving rise to the research area of software components.

Components are program modules that can be independently developed and delivered [138]. They allow for syntactic integration of existing heterogeneous applications, hiding all implementation details to their clients. No support for semantic integration is provided. Usually, components are distributed entities, supported by communication middleware frameworks, such as CORBA(Common Object Request Broker Architecture) [83], Microsoft's DCOM ((Distributed Component Object Model) [115]) and Sun's EJB (Enterprise Java Beans [117]). They export a well defined and agreed upon interface, consisting of a set of methods, that can easily be invoked over a network. Therefore, components enable tightly coupled interaction among partners, and assume a quite static environment, since no or very little support is provided for example in establishing dynamic relationships among components, or to cope with change in the interface definition of existing components. Finally, little support is provided to data, since interactions among components takes place at the communication layer, and consists in invoking methods (i.e., exchanging messages) using standard protocols such as HTTP. Several works on architecture design based on components and objects have addressed theoretical issues related to software composition [156, 70, 47]. Among the works on software components from a conceptual perspective [155, 38, 37, 45], we to focus on [155], which provides an interesting conceptual bridge (though implicit) between components and services. The authors propose to extend the interface of components with constraints defining the order in which methods (of the interface) can be invoked. They call protocol such sequencing constraints. Then, they give algorithms to check whether two protocols are compatible, i.e., the software components they are associated to, can communicate by correctly exchanging messages. If this is not the case, they introduce the notion of software adaptors, i.e., mediators that seats between two incompatible protocols and allow them to communicate. Finally, they present a tool that given two interfaces with incompatible protocols and a high-level description of the mapping between the two (i.e., defining rules that relate messages and parameters in the interfaces) automatically builds a software adaptor between them. Note that the results of this work can be easily applied in particular to compatibility checking between services that export a behavioral descriptions.

We argue that results in software components from a conceptual perspective can be fruitfully applied to service composition.

2.4 Comparison of Current Approaches

Tables 2.4 and 2.4 summarize the results about current or possible approaches to service description and composition, respectively, discussed in previous sections.

The tables are self-explanatory, however, we want to remark what follows.

- The tables do not contain dimensions referring to the degree of observability in the current state. In the approaches we analyzed, whenever the "Completeness of Exported Behavioral Description" is partial, so is the observability of the current state. As far as "Completeness in Client Request" dimension, since it is always full, so it is the observability of the current state. However, planning and reasoning about actions techniques can be profitably exploited to study this situation.

- Of course, even if it is not explicit in the tables, automatic composition of (possibly infinite) tree based or run based services and client request is possible when they admit some finite representation.

- Only two approaches to service composition, dealing with partial specifications (either of available services behavioral descriptions or of client request), produce conditional composition: both of them are based on planning and reasoning about actions techniques. Interestingly, to the best of our knowledge, no approach produces conformant composite services: again, techniques from AI can be used. Note also that usually standard AI planning techniques deal with atomic services, however, by mitigating them with approaches from Program Synthesis, one can fruitfully develop algorithms for conformant and conditional composite services that export both a run and a tree based behavioral description.

- As far as "Kind of Composition" dimension in Table 2.4, a sequential composition can be seen as a special case of concurrent composition. Therefore, the table implicitly expresses that each approach addressing sequential composition is able in general to tackle also the concurrent one.

- Only one approach proposes a peer to peer architecture for composition (and therefore orchestration). However, a mediated and therefore centralized architecture might result in bottleneck for invoking services. Therefore, peer to peer approaches, possibly derived from Data Integration (and Distributed Systems) are more than welcome.

- As far the "Support for Data" and "Data Flow" dimensions, we consider as "data", those input and output parameters of operations, or ingoing and outgoing messages, whose description is richer than a simple

name, e.g., on which constraints can be imposed. Therefore, parameters of Situation Calculus actions are treated as (albeit limited) data, while message names are not. Note that according to the tables, no approach exist to service composition where simultaneously services export a behavioral description (either run or tree based) and data flow issues are tackled. However, as also stated in [113], services require the integration and interoperation of both applications and data. Moreover heterogeneous data between all services involved in a composition must be dealt with. We argue that Data Integration based techniques should be suitably devised in order to tackle this point.

We want to conclude this section by pointing out the contributions of this thesis wrt the state of the art. The novelty and uniqueness of our approach consists in the fact that *(i)* we are able to automatically compute service composition in a framework where *(ii)* both the available services and the client request are based on a tree interaction model, *(ii)* the client request is either completely or partially specified, *(iii)* the composition is obtained by a concurrent coordination of the available services. In no other approaches those points *simultaneously* hold. In the following sections, we will explain in detail such statement.

Table 2.1: Comparison of various approaches to service description

	WSDL [6] **& WSDL-based** [112]	**WSCL** [77]	**WSCI** [12]	**OWL-based** [5]	**McIlraith & group** [108, 107]	**Papazoglou & group** [153, 120, 119]	**Vianu & group** [54]	**Ambite, Knoblock, Takkar** [139, 141, 140]
Interaction Model	atomic	tree based	tree based	Service Profile: atomic; Service Model: not decided yet	atomic	atomic	tree based	atomic
Completeness of Exported Behavioral Description	–	full	full	not addressed yet	–	–	full	–
Support for Data	yes (XML docs)	yes (XML docs)	yes (XML docs)	yes	limited (SitCalc params of actions)	yes (XML docs)	yes (relational schema)	yes (attribute of view over relational source)

	Hull & group [40]	**Traverso & group** [123]	**Benatallah, Casati, Toumani & group** [17, 10, 15, 16]	**Bordeaux, Salaün** [128, 35]	**Approach in current thesis**	**Data Integration** [80, 101]	**Program Synthesis** [95, 94, 93, 60]	**Planning & Reasoning About Actions** [33, 65, 126, 124, 72]	**Software Components** [155, 38, 37, 45]
Interaction Model	run based	tree based	tree based	tree based	tree based	atomic	run based, tree based	atomic, run based, tree based	atomic
Completeness of Exported Behavioral Descr.	full	partial	generally full, partial in [16]	full	full	–	full, partial	full, partial	–
Support for Data	no	no	no	no	no	yes	yes	yes	yes

Table 2.2: Comparison of various approaches to service composition

	Bouguettaya, Elmagarmid, Medjahed [112]	**Papazoglou & group** [153, 120, 119]	**Ambite, Knoblock, Takkar** [139, 141, 140]	**McIlraith & group** [108, 107]	**Hull & group** [40]	**Traverso & group** [123]	**Approach in current thesis**	**Data Integration** [80, 101]	**Program Synthesis** [95, 94, 93, 60]	**Planning & Reasoning About Actions** [33, 65, 126, 124, 72]
Client Request Model	run based	run based	atomic	mainly atomic, with run based aspects	run based	mainly run based, with tree based aspects	tree based	atomic	run based, tree based	atomic, run based, tree based
Completeness in Client Request	full	full	–	partial	full	full	full, partial	–	full, partial	full, partial
Kind of Composition	sequential	sequential, concurrent	sequential	sequential, conditional	sequential, concurrent	concurrent, conditional	sequential, concurrent	sequential	sequential, concurrent	sequential, concurrent, conditional, conformant
Kind of Orchestration	mediated	mediated	mediated	mediated	peer to peer	mediated	mediated	mainly mediated, also peer to peer	mediated	mediated
Type of Composition	semi-automatic	manual	automatic	automatic	automatic	automatic	automatic	automatic	automatic	automatic
Data Flow	yes	yes	yes	limited	no	no	no	yes	no	limited

2.5 Results related to Service Description and Composition

In this section we present the main results in the areas of service discovery (i.e., how to efficiently query against service descriptions), and orchestration (i.e., invocation, enactment and monitoring of both simple and composite services). These are tightly related to the areas of service modeling and composition.

2.5.1 Service Discovery

Industry Currently, the leading industry standard for service discovery [59] is UDDI (Universal Description, Discovery and Integration). UDDI[8] [142] is a registry specification proposed by an industry consortium lead by IBM, Microsoft and Ariba. The repository, referred to as UDDI Business Registry is logically centralized and physically distributed: each partner of the consortium "owns" a UDDI Registry, but the information of each UDDI Registry is replicated in all the others. In other words, a service provider can publish a service to the UDDI Registry of any partner; this service is then duplicated to the UDDI Registry of all other partners, so that a user can invoke it from any UDDI Registry. The core of Business Registry consists of three conceptual components: the "white pages" that contain business information about the service provider; the "yellow pages", containing classifications of services in various taxonomies, as yellow pages do; the "green pages", that provide technical informations about published services. A (human) client that wants to exploit a given service, performs a search based on the service name and, if he finds it, the service description and a link to the service provider home page are returned. Following this link, the user can invoke the service. UDDI supports description of services in WSDL.

Academia In [57] the authors present algorithms to discovery services, modeled as WSDL files, based on similarity matching. In particular, similarity search is done both on input and output parameters of operations or on operations themselves. The similarity algorithms are based on clusters of input/output parameter names which co-occur in a certain set of services. Such clusters are computed by heuristics, validated in practice, based on the conditional probabilities of parameter occurrence in inputs and outputs of service operations, i.e., capturing the property that parameters expressing the same concept usually occur together.

In [143], services are considered as constituted by sub-services, thus modeled as a hierarchy of parts (expressing capabilities of services), based on a

[8] `http://www.uddi.org.`

common ontology. On the assumption that all descriptions of available services are stored in a common repository, an algorithm that select the service that best fits a given description (i.e., the request for specific capabilities) is presented, based on similarity notions. Such selection is currently carried out only on the basis of static features similarity. Other works on service discovery propose information retrieval techniques [149], peer-to-peer scenarios [133] and graph-based techniques in the context of Semantic Web Services [18]. Finally, in [56] the authors address the problem of discovering services in the UDDI Registry, and propose a framework that integrates the semantics of services into the UDDI Registry, by providing an extension of SWS ontologies.

Discussion and Open Issues UDDI supports a quite limited search of a service: if the client does not know the name of the service he wants to exploit, but knows only its functionality, he is not able to discover and invoke it. In other words, UDDI does not take into account search based on the semantics of services. The basic idea of the Semantic Web Service initiative is overcome such problem by providing a semantic structure to the web. The Internet contains a wide quantity of data, with no associated semantics: for instance, the search engines return information based only on the match between the user query and the words contained in the web pages, completely ignoring the semantic context. Therefore, contributions such as the one in [56] are needed, since they provide a bridge between the industry and academia efforts, by filling the gap between technological substrate and conceptual models.

We would like to remark that all such approaches tackle into account only “static” service signatures, whereas considering behavioral descriptions could improve the quality of the discovery process. Our work is orthogonal to service discovery issues: we assume that the set of services that can be used in the composition has already been assembled, therefore, service discovery techniques play an important role in the construction phase of such a set. However, until now all works addressing service discovery have tackled the situation when the client request is matched by one service only: instead, it would be interesting to consider also the case when a *set* of services matches the client request.

2.5.2 Service Orchestration

Most of the work on service orchestration is based on research in workflows, which model business processes as sequences of (possibly partially) automated activities, focusing on both data and control flow among them. Traditional workflow technology assumed intra-organization cooperation and tightly coupling of business processes participating in the workflow. Current efforts, such

as the Business Process Initiative[9], address the interconnection and interaction of heterogeneous business processes across different organizations [152].

Orchestration requires that the composite service is completely specified, in terms of both the specification of how various component services are linked, and the internal process flow of the composite one. In [89], different technologies, standards and approaches for specification of composite services are considered. In particular, in [89] two main kinds of (composition and) orchestration are identified: *(i)* the mediated approach, based on a hub-and-spoke topology, in which one service is given the role of process mediator/delegator, and all the interactions pass through such a service, and *(ii)* the peer-to-peer approach, in which the services directly interact among them, without any centralized control.

Industry The standard language for specifying orchestration and coordination of services is Business Process Execution Language for Web Services (BPEL4WS, [4]). It is an XML-based language able to express how multiple services are coordinated and the state and the logic needed for such a coordination. In other words, a BPEL4WS file denotes the specification of a composite service to be orchestrated. BPEL4WS constructs allows to specify composite services whose underlying conceptual model is based on trees: in particular, there are constructs *(i)* expressing branching in the form of "case" like statements, *(ii)* allowing to execute of one of several alternative paths, and *(iii)* indicating that a collection of steps should be executed concurrently. The coordination among component services is expressed in terms of allowed interactions between them, specified as sequences of operations and of message exchanges. Finally, BPEL4WS is also equipped with constructs for correlation of service instances, for exception management, and for limited time modeling. Note that BPEL4WS has no underlying conceptual model.

BPEL4WS has been developed with the aim of merging two competing proposals of standard languages, Web Service Flow Language (WSFL [102]), proposed mainly by IBM, and XLANG [130], proposed mainly by Microsoft. WSFL proposed to design composite Web Services starting from simple ones, taking Petri Nets as underlying conceptual model. XLANG specified both the behavior of services and their orchestration; in [114], it has been pointed out that this language is complete, it owns the property of composability and that the underlying theoretical model is the one of process algebra.

Recently, the W3C consortium has proposed a new language for specifying the coordination among services, namely Web Service - Choreography Description Language (WS-CDL [90]). A WS-CDL specification is "a multi-participant contract that describes the common observable behavior of the

[9]Cf. `http://www.bpmi.org`

collaborating WS participants" [127], from an external global point of view. WS-CDL is an XML-based language whose conceptual model is based on π-calculus: each participant service, whose behavior is described through a sequential finite state transition system, interacts with the others and shares resources through predefined channels. WS-CDL provides constructs for communication, choices, concurrency and iteration, having a clear formal semantics based on analogous π calculus constructs. Also, thanks to the clear formal conceptual model, it is possible to formally verify properties such as livelock or deadlock, on a WS-CDL specification.

Finally, note that BPL4WS assumes a mediated model for orchestration: a BPL4WS orchestration engine executes a BPL4WS specification by coordinates the component services on the basis such a specification. Instead, WS-CDL is based on a peer-to-peer architecture: the various services coordinates one with the other on the basis of the WS-CDL specification (which is not executable), with no centralized controller.

Academia Many orchestration platforms have been designed and proposed in the literature. In [46], a service that performs coordination of services is considered as a (meta) service, referred to as Composite service (CES). A provider can offer a value added service as coordination of different services: it registers the new service to the CES and let the CES enacting its execution. In [135], coordination of services is obtained by an enactment engine interpreting process schemas modeled as statecharts [151]. In [49] the orchestration of session-oriented, long running telecommunication services is studied. Finally, in [109], orchestration of services is addressed by means of Petri Nets. All these approaches can be classified into the mediated approach to composition.

The peer-to-peer architecture is considered in [62, 14], where a composite service is modeled as an activity diagram, and its enactment is carried out through the coordination of different state coordinators (one for each component service and one for the composite service itself), in a decentralized way.

Finally, we would like to remark that our results are orthogonal to service orchestration; once we obtain a composition, using with our technique, it is possible to translate it into a specific orchestration language, so that it can be orchestrated by any orchestration platform, supporting a mediated architecture. In this way, we get all system-level guarantees needed in complex distributed applications.

Chapter 3

Services and Service Composition: General Characterization

In this chapter, we present a general formal framework where services are characterized in terms of their behavioral descriptions. In other words, we clearly define, in the general case, the process semantics of services in terms of their service behavioral description. Then, on top of this, we formally characterize the problem of service composition.

3.1 Basics on Services

Generally speaking, a service is a software artifact (delivered over the Internet) that interacts with its clients in order to perform a specified task. A client can be either a human user, or another service[1]. When executed, a service performs its tasks by directly executing certain actions, possibly interacting with other services to delegate to them the execution of other actions. In order to address the Service Oriented Computing paradigm from an abstract and conceptual point of view, we start by identifying several facets, each one reflecting a particular aspect of a service during its life time.

- The service *schema* specifies the features of a service, in terms of functional and non-functional requirements. Functional requirements represent *what* a service does. All other characteristics of services, such as those related to quality, privacy and security, performance, transactions,

[1]In what follows, we refer to the client with the "he" pronoun, in order to avoid confusion when referring to the services and the clients using the pronouns. However, the reader should remember that we could as well as use the "it" pronoun for the client.

etc. constitute the non-functional requirements. In what follows, we do not deal with non-functional requirements, and hence we use the term "service schema" to denote the specification of functional requirements only.

- The service *implementation and deployment* indicate *how* a service is realized, in terms of software applications corresponding to the service schema, deployed on specific platforms (e.g., .NET, J2EE). This aspect regards the technology underlying the service implementation, and it goes beyond the scope of this thesis. Therefore, although implementation issues and other related characteristics such as recovery mechanisms or exception handling, are important issues in Service Oriented Computing, in what follows we abstract from these properties.

- A service *instance* is an occurrence of a service effectively running and interacting with a client. In general, several running instances corresponding to the same service schema may co-exist, each one executing independently from the others.

In order to execute a service, the client needs to *activate* an instance of a deployed service. In our abstract model, the client can then interact with the service instance by repeatedly *choosing* an action and waiting for either the fulfillment of the specific task, or the return of some information. On the basis of the returned information the client chooses the next action to invoke. In turn, the activated service instance executes (the computation associated to) the invoked action; after that, it is ready to execute new actions. Under certain circumstances, i.e., when the client has reached his goal, he may explicitly *end* (i.e., terminate) the service instance. However, in principle, a given service instance may need to interact with a client for an unbounded, or even infinite, number of steps, thus providing the client with a continuous service. In this case, no operation for ending the service instance is ever executed. In Section 3.4.1, we discuss in detail the interaction protocol between a service (instance) and its client wrt our framework. Here, we give an intuition of it in the following example.

Example 2 *A client wants to search and listen to mp3 files. Hence, he activates an instance of a deployed service that can fulfill his needs. Once the service instance is activated and all the necessary resources for its execution are allocated, it presents the client with the set of actions that can be executed next, namely* (i) `search_by_author`*, for searching a song by specifying its author(s),* (ii) `search_by_title`*, for searching a song by specifying its title, and* (iii) `end`*, for ending the interactions. The client chooses the first action and the service executes it. Again, the service presents the client with a new*

set of actions: let it be a singleton set, constituted by the action `listen`*, for selecting and listening to a song*[2] *Thus, the client chooses that action and the service executes it. At this point the service offers the client with another set of actions, that can be* (i)*,* (ii)*, and* (iii) *above. The client makes his choice, for example* `search_by_title`*, and the interactions continue. When the client has reached his goal, he selects the action* `end`*, the service instance de-allocates all the resources associated to it and its execution ends.* □

Note the difference between this approach, in which the focus is on *actions*, and the approach that can be found in languages such as WSDL, BPEL4WS, etc., where the focus is on *messages* (and parameters). For example, in WSDL, an interaction between the service and the client is not modeled by an action, say `search_by_author`, but by *(i)* a message that the client sends to the service for requesting a search, say `search_by_author_request`, and *(ii)* a message that the service sends to the client (and, in his turn, the client receives) for responding to a search, say `search_by_author_response`. In other words, the actions in our framework can be seen as the abstractions of the effective input/output messages and operations offered by the service. In Section 7.4 we will investigate this aspect in detail.

3.2 Service Community

In general, when a client invokes an instance e, activated of a service with a schema E, it may happen that e does not execute all of its actions on its own, but instead it *delegates* some or all of them to other (instances of) services, according to its schema. All this is completely hidden to the client. To precisely capture the situations when the execution of certain actions can be delegated to (instances of) other services, we introduce the notion of *service community*:

Definition 1 (Service Community) Let $\mathrm{E} = \{E_1, \ldots, E_n\}$ be a set of abstract names. A service community is formally characterized by:

- a finite common set of actions Σ, called the *action alphabet*, or simply the *alphabet* of the community,
- a finite set of services specified in terms of the common set of actions, and referred to using the abstract names in E.

□

[2]We assume for simplicity that the list of songs returned by `search_by_author` and `search_by_title` is non-empty.

In other words, all the services in a community *share a common understanding* over the actions in the alphabet Σ. Hence, to join a community C, a service needs to expose its behavior in terms of the alphabet of C. Also the clients interact with services in C using Σ. Note that the set E denotes simply the names of services in C. Additional structure will be associated to these names later on.

From a more practical point of view, a community can be seen as the set of all services whose descriptions are stored in a repository. We assume that all such service descriptions have been produced on the basis of a common and agreed upon reference alphabet/semantics. This is not a restrictive hypothesis, as many scenarios of Cooperative Information Systems, e.g., *e*-Government [11] or *e*-Business [50] ones, consider preliminary agreements on underlying ontologies, yet yielding a high degree of dynamism and flexibility. Therefore, the repository where services are published may be seen as an advanced version of UDDI [142], that offers to its clients functionalities which do not regard only service discovery, selection and invocation, but also service composition, personalization, (some aspects of) negotiation, etc.

The added value of a community is the fact that a service of the community may delegate the execution of some or all of its actions to other services in the community. We call *composite* such a service. If this is not the case, a service is called *simple*. Simple services realize offered actions directly in the software artifacts implementing them, whereas composite services, when receiving requests from clients, can (activate and) invoke other services in order to fulfill the client's needs.

This function of composing existing services on the basis of a target service is known as service composition, and is the main subject of the research in this thesis.

3.3 Service Schema

From the external point of view, i.e., that of a client, a service E, belonging to a community C, exhibits a certain *exported behavior* represented as sequences of atomic *actions* of C with constraints on their invocation order. From the internal point of view, i.e., that of an application deploying E and activating and running an instance of it, it is also of interest how the actions that are part of the behavior of E are effectively executed. Specifically, it is relevant to specify whether each action is executed by E itself or whether its execution is delegated to another service belonging to the community C, in a way which is hidden to the client of E. To capture these two points of view we introduce the notion of service schema, as constituted by two different parts, called *external schema* and *internal schema*, respectively. Before this, we introduce the notion

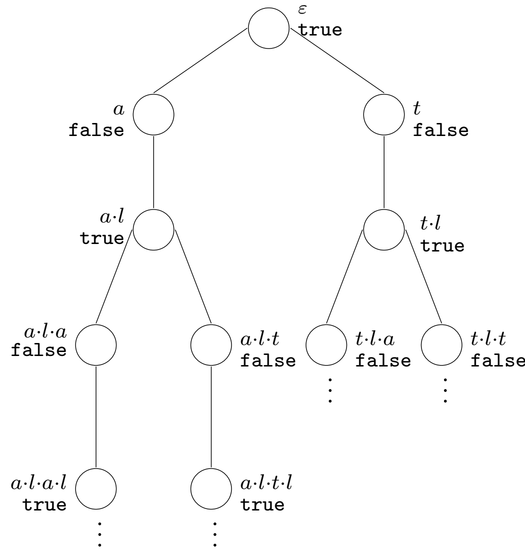


Figure 3.1: Labeled Tree

of labeled trees, that we will use throughout the chapter.

3.3.1 Labeled Trees

A tree $\mathcal{T}$ over an alphabet Σ is a prefix closed (possibly infinite) set of finite words over Σ, i.e., a set of words $\mathcal{T} \subseteq \Sigma^*$, called *nodes*, such that if $x{\cdot}c \in \mathcal{T}$, with $x \in \Sigma^*$ and $c \in \Sigma$, then also $x \in \mathcal{T}$. The empty word ε is called the *root* of $\mathcal{T}$, and for every $x \in \mathcal{T}$, the node $x{\cdot}c$, with $c \in \Sigma$, is called the *successor* of x. The pair $(x, x{\cdot}c)$ is called *edge* of the tree. A *labeled tree* is a pair $(\mathcal{T}, f)$, where $\mathcal{T}$ is a tree and f is a *labeling function* assigning to each node of $\mathcal{T}$ an element of a given labeling domain.

Example 3 *Figure 3.1 shows a (portion of an infinite) labeled tree* $(\mathcal{T}, f)$ *over the alphabet* $\Sigma = \{a, t, l\}$*.* f *is a boolean labeling function: it labels with* `true` *the root* ε *and the nodes* $a{\cdot}l, t{\cdot}l, a{\cdot}l{\cdot}a{\cdot}l, a{\cdot}l{\cdot}t{\cdot}l, t{\cdot}l{\cdot}a{\cdot}l, t{\cdot}l{\cdot}a{\cdot}l$ *(i.e., all the nodes ending with* l*); it labels with* `false` *the nodes* $a, t, a{\cdot}l{\cdot}a, a{\cdot}l{\cdot}t, t{\cdot}l{\cdot}a, t{\cdot}l{\cdot}t$ *(i.e., all the nodes ending with* a *or* t*).* □

3.3.2 External Schema

The aim of the external schema is to specify the exported behavior of a service. For now, in order to guarantee a general applicability of our framework, we do not refer to any particular specification formalism, rather we only assume that, whatever formalism is used, the external schema specifies the behavior in terms

of a tree of actions, called *external execution tree*. The external execution tree abstractly represents all possible executions of a generic instance of a service. Therefore, when activated, an instance of a service executes a path of such a tree.

Definition 2 (External Execution Tree) Let E^{ext} be the external schema of a service E. Let Σ be the alphabet of the service community C, to which E belongs. The external execution tree specified by E^{ext} is a labeled tree $T(E^{ext}) = (\mathcal{T}, \mathit{fin})$, where $\mathcal{T}$ is a tree over Σ and *fin* is a boolean labeling function, where:

- Each node x of $T(E^{ext})$ represents the history of the sequence of actions of each service instance[3], that has executed the path to x.
- For every action a belonging to the alphabet Σ of the community, and that can be executed at the point represented by x, there is a (single) successor node $x \cdot a$. We say that each each edge $(x, x \cdot a)$ is *labeled* with action a.
- The root ε of the tree represents the fact that the service has not yet executed any action.
- The nodes of the execution tree labeled `true` by *fin* are *final*: when a node is final, and only then, the client can stop the execution of the service. In other words, the execution of a service can legally terminate only at these points[4].

$\square$

Note that each node $x \cdot a$ represents the fact that, after performing the sequence of actions leading to x, the client chooses to execute action a, among those possible, thus getting to $x \cdot a$. Therefore, each node represents a choice point at which the client makes a decision on the next action the service should perform.

In what follows, for sake of readability, we denote the external execution tree of a service in C by means of a mapping T^{ext} from E to $T(E^{ext})$, where E is the set of abstract names of the services in the community C (so, in what follows, instead of writing $T(E^{ext})$, we will write $T^{ext}(E)$, where $E \in$ E). We use the mapping as a means of associating additional formal structure to the (names of) services in C, or, equivalently, as a way to enforce the fact that the external execution tree is associated to the service E.

[3]In what follows, we omit the terms "schema" and "instance" when clear from the context.

[4]Typically, in a service, the root is final, to model that the computation of the service may not be started at all by the client.

Notably, in our framework, an execution tree does not represent the information returned to the client by the service instance execution, since the purpose of such information is to let the client choose the next action, and the rationale behind this choice depends entirely on the client.

Consider the above assumption that for every action a there is (at most) one single successor node $x{\cdot}a$: we argue that this assumption here is perfectly valid and coherent with our framework since our focus is *on actions* that a service performs. Therefore, at the level of abstraction taken here one has to pay careful attention on the meaning of having two a labeled edges originating from the same node. Such a situation, indeed, in our framework, means that the effects of an action are partially specified, i.e., they are not completely known, as in the situation when, if a client deposits a certain amount of money on his bank account, he is not sure whether after executing such operation the money are actually in his account or have been deposited on another account, unknown to him. The case when services export a partial description of their behavior is, of course, very compelling: it gives rise to many interesting issues, in order to avoid using services whose effects may be undesirable. However, it goes beyond the scope of this thesis and it is left for future research (see Sections 5.4 and 8.2).

Finally, observe that we avoid introducing data at the level of abstraction taken here: in this way the complexity which is intrinsic in the data does not have a disruptive impact on the complexity which is intrinsic in the process. In fact, introducing data in a naive way is possible in our setting (e.g., by encoding data within the state) but it would make composition exponential in the data. This is considered unacceptable: the size of data is typically huge (wrt the size of the services) and therefore the composition should be kept polynomial in the data. The issues on how to introduce data in a "smart" way is left for future work (see Section 8.2).

Example 4 *Figure 3.2 shows (a portion of) an (infinite) external execution tree characterizing the behavior of service E_0, that allows for searching and listening to `mp3` files. The client starts by choosing whether to stop, or to search for a song by specifying either its author(s) or its title (action `search_by_author` and `search_by_title`, respectively). Then, the client selects and listens to a song (action `listen`). Finally, the client chooses whether to perform those actions again.*
Note that the tree in Figure 3.2 is actually the same tree of Figure 3.1, with the following differences, introduced to increase readability: (i) *final nodes are represented by two concentric circles, instead of being labeled by* `true`*, and* (ii) *each edge $(x, x{\cdot}a)$ is labeled with the last executed action a, instead of having nodes labeled by prefix closed words over Σ.* □

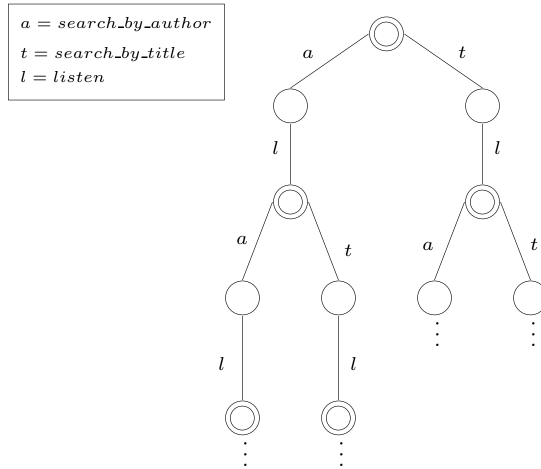


Figure 3.2: External execution tree of service E_0

3.3.3 Internal Schema

The internal schema specifies, besides the tree of actions representing the external behavior of the service, the information on which service instances in the community execute each given action. As before, for now we abstract from the specific formalism chosen for giving such a specification, instead we concentrate on the notion of *internal execution tree.*

Definition 3 (Internal Execution Tree) Let E^{int} be the internal schema of a service E. Let Σ be the alphabet of the service community C to which E belongs. The internal execution tree specified by E^{int} is a labeled tree $T(E^{int}) = (\mathcal{T}, \mathit{fin})$, where:

- $\mathcal{T}$ and *fin* are defined as in Definition 2.
- In addition, each edge of $\mathcal{T}$ is labeled by the pair (a, I), where a is the executed action and I is a nonempty set denoting the service instances executing a. Every element of I is a pair (E', e'), where E' is a service and e' is the identifier of an instance of E'. The identifier e' unambiguously identifies the instance of E' within the service community, and, therefore, within the internal execution tree.

□

In what follows, for sake of readability, we denote the internal execution tree of a service in C by means of a mapping T^{int} from E to $T(E^{int})$, where E is the set of abstract names of the services in the community C (so, in what

follows, instead of writing $T(E^{int})$, we will write $T^{int}(E)$, where $E \in$ E). We use the mapping as a means of associating additional formal structure to the (names of) services in C, i.e., as a way to enforce the fact that the internal execution tree is associated to the service E.

In general, in the internal execution tree of a service E, some actions may be executed also by the running instance of E itself. In this case we use the special instance identifier `this`. Note that, since I is in general not a singleton, the execution of each action can be delegated to more than one other service instance.

An internal execution tree *induces* an external execution tree: given an internal execution tree T_{int} we call *offered external execution tree* the external execution tree T_{ext} obtained from T_{int} by dropping the part of the labeling denoting the service instances, and therefore keeping only the information on the actions. An internal execution tree T_{int} *conforms to* an external execution tree T_{ext} if T_{ext} is equal to the offered external execution tree of T_{int}.

Definition 4 (Well-formed Service) A service E is *well-formed*, if $T^{int}(E)$ conforms to $T^{ext}(E)$, i.e., its internal execution tree conforms to its external execution tree. □

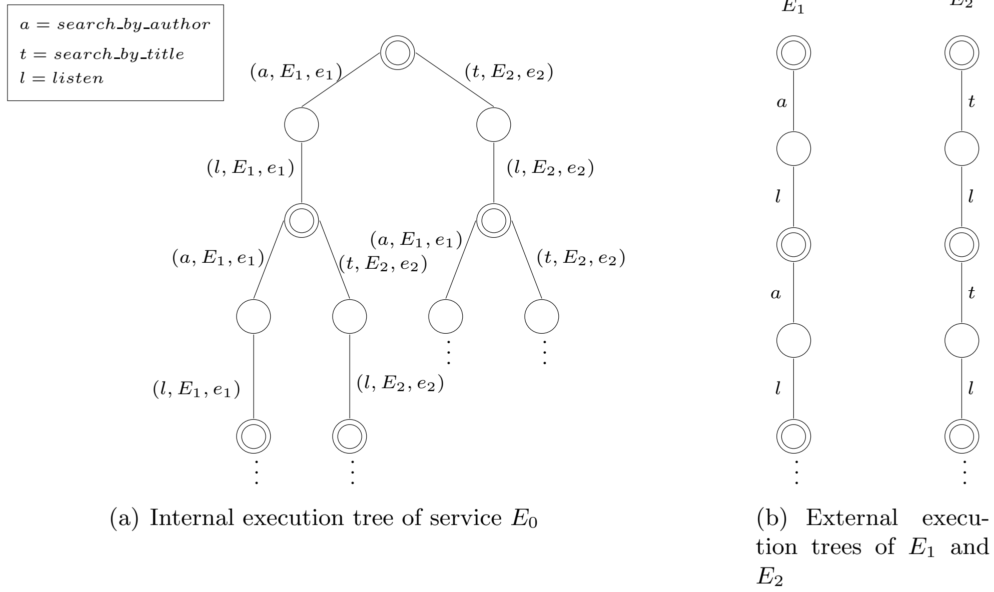
We now formally define when a service of a community *correctly delegates* actions to other services of the community. We need a preliminary definition: given the internal execution tree $T^{int}(E)$ of a service E, and a path p in $T^{int}(E)$ starting from the root, we call the *projection* of p on an instance e' of a service E' the path obtained from p by removing each edge whose label (a, I) is such that I does not contain e', and collapsing start and end node of each removed edge.

Definition 5 (Coherency) The internal execution tree $T^{int}(E)$ of a service E is *coherent* with a community C if:

- for each edge labeled with (a, I), the action a is in the alphabet of C, and for each pair (E', e') in I, E' is a member of the community C;
- for each path p in $T^{int}(E)$ from the root of $T^{int}(E)$ to a node x, and for each pair (E', e') appearing in p, with e' different from `this`, the projection of p on e' is a path in the external execution tree $T^{ext}(E')$ of E' from the root of $T^{ext}(E')$ to a node y, and moreover, if x is final in $T^{int}(E)$, then y is final in $T^{ext}(E')$.

□

Definition 6 (Delegation) A service E of a community C correctly delegates actions to other services of C if the internal execution tree $T^{int}(E)$ of E is coherent with C. □



(a) Internal execution tree of service E_0

(b) External execution trees of E_1 and E_2

Figure 3.3: Internal execution tree

Observe that, if a service of a community C is simple, i.e., it does not delegate actions to other service instances, then it is trivially coherent with C. Otherwise, it is composite and hence delegates actions to other service instances. Intuitively, in the latter case, as expressed by the second bullet of Definition 5, the behavior that the composite service entails on each component service (instance) must be "correct" according to the external schema of the component service (instance) itself.

Definition 7 (Well-formed Community) A service community is *well-formed* if each service in the community is well-formed, and the internal execution tree of each service in the community is coherent with the community. □

Example 5 *Figure 3.3(a) shows (a portion of) an (infinite) internal execution tree*[5]*, conforming to the external execution tree of service* E_0 *shown in Figure 3.2, where all the actions are delegated to services of the community, specifically to* E_1 *and* E_2*, whose (infinite) external execution trees are (partially) shown in Figure 3.3(b). In particular, the execution of* `search_by_author`

[5]In the figure, each action is delegated to exactly one instance of a service schema. Hence, for simplicity, we have denoted a label $(a, \{(E_i, e_i)\})$ simply by (a, E_i, e_i), for $i = 1, 2$.

action and of its subsequent `listen` *action are delegated to instance* e_1 *of service* E_1*, and the execution of* `search_by_title` *action and of its subsequent* `listen` *action to instance* e_2 *of service* E_2*.* □

3.4 Service Instances

In order to be executed, a deployed service has to be activated, i.e., necessary resources need to be allocated. A service instance represents a service running and interacting with its client.

From an abstract point of view, a running instance corresponds to an execution tree with a highlighted node, representing the current node, i.e., the point reached by the execution at a certain moment.

Definition 8 (Service instance)
A service instance is characterized by:

- an *instance identifier*,
- an *external view* of the instance, which is an external execution tree with a current node,
- an *internal view* of the instance, which is an internal execution tree with a current node,
- the current node on the tree which denotes the state of the service.

□

Note that the path from the root of the tree to the current node is the run of the service so far, while the execution (sub-)tree having as root the current node describes the remaining behavior once the current node is reached. Observe also that the current node on the external view of the instance coincides with the current node on the internal view.

Example 6 *Figure 3.4 shows an external view of instance* e *of the service* E_0 *of Figure 3.2. The current node and the sequence of actions executed up to it on the execution tree are shown in thick lines. It represents an execution by a client that has searched for an* `mp3` *file by specifying the author of the song, and has selected and listened to a returned song. The client has reached a node where it is necessary to choose whether* (i) *performing another* `search_by_author`, (ii) *performing a* `search_by_title`*, or* (iii) *terminating the service (since the current node is final).* □

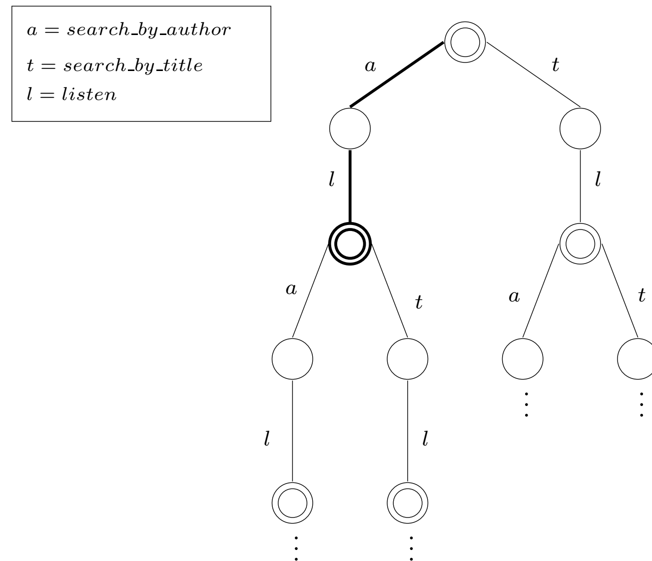


Figure 3.4: External view of a service instance.

The internal view of a service instance additionally maintains information on which service instances execute which actions. At each point of the execution there may be several other active instances of services that cooperate with the current one, each identified by its instance identifier.

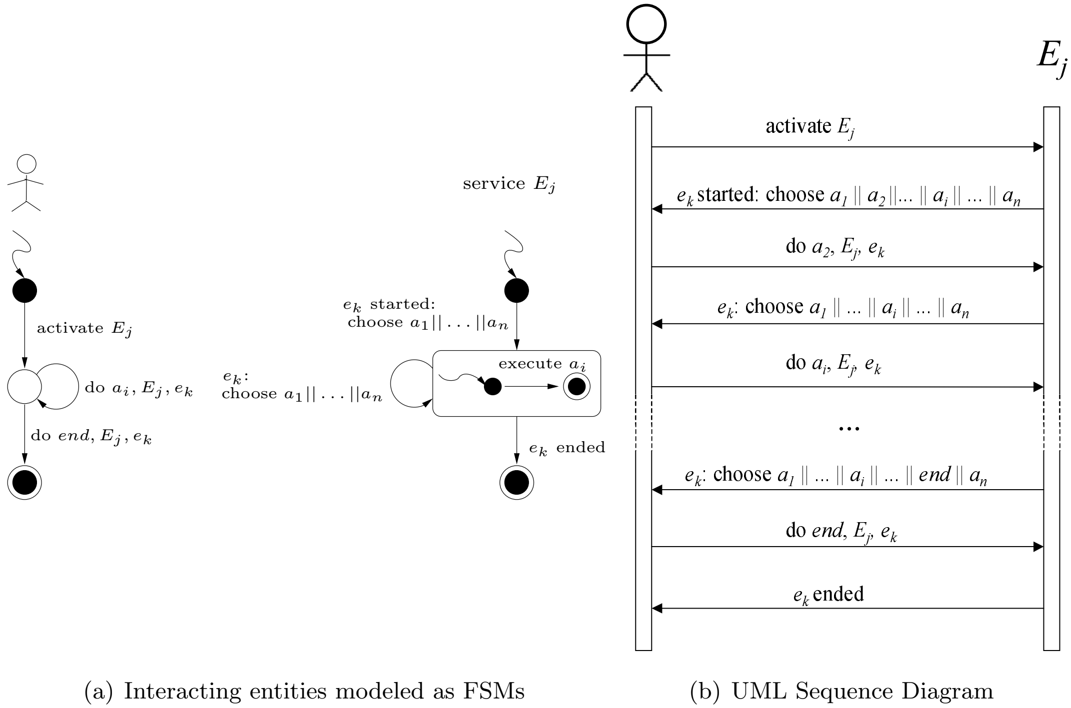
3.4.1 Running a Service Instance

In Section 3.1 we have briefly discussed the steps that a client should perform in order to execute a service, namely:

1. activation of the service instance,
2. choice of the invokable actions
3. termination of the service instance,

where step (*2*) can be performed zero or more times, and steps (*1*) and (*3*) only once. Each of these steps is constituted by sub-steps, consisting in executing commands and in sending acknowledgments, each of them being executed by a different entity (either the client or the service).

In what follows we describe the correct sequence of interactions between a client and a service in our framework. As already stated, in general the client may be either a human user or another service, however, for the sake of simplicity, in what follows we refer to the client using the "he" pronoun.



(a) Interacting entities modeled as FSMs (b) UML Sequence Diagram

Figure 3.5: Conceptual Interaction Protocol

Figure 3.5 shows the conceptual interaction protocol: in particular, Figure 3.5(a) displays the behavior of each entity, represented as a UML State Diagram [67]; Figure 3.5(b) presents the interaction among the entities captured as UML Sequence Diagram [67]. We briefly recall that a sequence diagram shows the explicit sequence of communications between interacting entities, where the vertical bar represents the time proceeding down the page.

Activation. This step is needed to create the service instance. The client invokes the activation command, specifying the service to interact with. The syntax of this command is:

$$\textbf{activate } E_j$$

where E_j is the service being activated. When this command is invoked, a new instance e_k of service E_j is created and all the necessary resources for the execution of e_k are allocated. Additionally, each service instance creates a copy of both the internal and the external execution tree characterizing the service schema it belongs to. The current node on the execution tree associated to e_k is the root.

As soon as e_k is ready to execute, it responds to the client with the message

$$e_k \textbf{ started: choose } a_1||\ldots||a_i||\ldots||a_n$$

The purpose of this message is threefold. First, the client has an acknowledgment that the invoked service has been activated and that the interactions may correctly start. Second, the client is informed about the instance identifier he will interact with (e.g., e_k). Third, the client is asked to choose the action to execute among $a_1, \ldots, a_n$.

Choice. This step represents the interactions carried on between the client and the service instance. Each service instance is characterized, wrt the client, by its external execution tree, and all the actions are offered according to the information encoded in such a tree. Therefore, according to its external execution tree, the service instance e_k proposes to its client a set of possible actions, e.g., $a_1, \ldots, a_n$, and asks the client to choose the action to execute next among $a_1, \ldots, a_n$. The syntax of this command is:

$$e_k\textbf{: choose } a_1||a_2||\ldots||a_i||\ldots||a_n$$

where || is the choice symbol.

According to his goal, the client makes his choice by sending the message

$$\textbf{do } a_i, E_j, e_k$$

In this way, the client informs the instance e_k of service E_j that he wants to execute next the action a_i. Once e_k has received this message, it executes action a_i. The client is completely unaware about how the execution of a_i is achieved: he only knows when it ends, i.e., when the service asks him to make another choice. This lack of observability is shown in Figure 3.5 (a) by the composite state that contains a State Diagram modeling the execution of a_i (action *execute* a_i, performed by e_k). The role of E_j and e_k becomes especially clear if we consider that the client could be a composite service. When a composite service E delegates an action to a component service (e.g., E_j), it needs to activate a new service instance (e.g., e_k). Therefore, on one side, E acts as "server" towards its client; on the the other side, E interacts with the *external view* of the instance of the component service, since E is a client of the latter, and E chooses which action is to be invoked on which service (either itself or a component service) according to its internal execution tree.

Termination. Among the set of invokable actions there is a particular action, **end**, which, if chosen, allows for terminating the interactions. Therefore, if the current node on the external execution tree is a final node, the service proposes a choice as:

$$e_k\text{: } \textbf{choose } end||a_1||a_2||\dots||a_i||\dots||a_n$$

and if the client has reached his goal, he sends the message:

$$\textbf{do } end, E_j, e_k$$

The purpose of this action is to de-allocate all the resources associated with the instance e_k of service E_j. As soon as this is done, the service informs its client of it with the message:

$$e_k\text{: } \textbf{ended}$$

Further examples of interactions can be found in [26].

3.5 Client Specification as Target Service

In our framework, a client willing to achieve a certain goal exploiting the service technology, specifies his request to (a system offering the functionalities of) the service community in terms of the service he would like to use (i.e., activate and interact with). In general, of this service he may indicate functional and non-functional properties of its schema, features regarding implementation and deployment aspects, such as preferred platform, desired transport layer, transaction policies, etc. As remarked in Section 3.1, in this thesis we deal only with the functional aspects of a service schema, therefore, we require that the client specifies (at least) the external schema of the service he desires to use.

Definition 9 (Target Service) A *target service* E is any client request that specifies (at least) the external schema T_{ext} of the service the client would like to activate and interact with. □

Note that the target service is *virtual*, in the sense that it represents only a specification of the client's need. Indeed, a target service cannot be executed as it is, i.e., by activating an instance of it, since there is no underlying implementation of it: in order to be executed, a target service should be "realized". Thus, the community can be used to realize a virtual service as a new service whose execution completely delegates actions to other members of the community. The target service is realized not simply by selecting a member (i.e., a schema from which to activate an instance) of the community, to which delegate target service actions, but more generally by suitably "composing" parts of services in the community.

Finally, note that not all client requests are target services: in Chapter 6 we will see examples of client specifications that do not specify external schemas, since they contain "empty spots", that the composition synthesis process is left free to "fill in".

3.6 Composition Synthesis

When a client requests a target service from a service community, there may be no service in the community that can deliver it directly. However, it may still be possible to synthesize a new composite service, which suitably delegates action execution to the services of the community, and when suitably orchestrated, provides the user with the service he requested.

Definition 10 (Composition) Let C be a well-formed service community and let T_{ext} be the external schema of a target service E expressed in terms of the alphabet Σ of C. A *composition* of E wrt C is an internal schema E^{int} such that:

1. $T(E^{int})$ conforms to T_{ext},

2. $T(E^{int})$ delegates all actions to the services of C (i.e., `this` does not appear in $T(E^{int})$), and

3. $T(E^{int})$ is coherent with C.

□

Definition 11 (Composition Existence) Given the service community C and the external execution tree T_{ext} of a target service E, as in Definition 10, the problem of *composition existence* is the problem of checking whether there exists a composition of E wrt C. □

Observe that, since for now we are not placing any restriction of the form of E^{int}, the problem of composition existence corresponds to check if there exists an internal execution tree T_{int} for E such that *(i)* T_{int} conforms to T_{ext}, *(ii)* T_{int} delegates all actions to the services of C, and *(iii)* T_{int} is coherent with C.

Definition 12 (Composition Synthesis) Given the service community C and the external execution tree T_{ext} of a target service E, as in Definition 10, the problem of *composition synthesis* is the problem of synthesizing an internal schema E^{int} for E that is a composition of E wrt C. □

Intuitively, the composition synthesis produces a new, composite service, which can be possibly stored in the service community for re-use by other clients. It is characterized by:

- an external schema, which is the external execution tree T_{ext} of the target service E, and

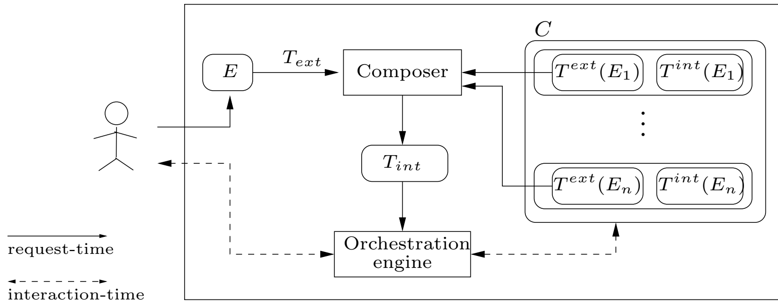


Figure 3.6: Conceptual model for a Service Composition System

- an internal schema, which is the composition T_{int} of E wrt the service community C.

Figure 3.6 shows the abstract, conceptual model of a *Service Composition System* which returns possibly composite services realizing the client requests, exploiting the available services of a community C. Note that all services in C are characterized in terms of both an internal and an external schema (i.e., execution tree). In Section 7.4 we will discuss the actual architecture of the composition tool that we implemented, and which is based on the architecture in Figure 3.6. According to Figure 3.6, a client requests a target service E to a Service Composition System, by providing the specification of its external schema T_{ext}, and expects to execute an instance of E. To do so, first the *Composer* module takes in input the external execution tree T_{ext} contained in the client specification and synthesizes an internal schema T_{int} which is a composition of E wrt C, according to Definition 10. Then, the composition T_{int} is (expressed in a suitable formalism/language and) orchestrated by an *orchestration engine* that activates and interacts with the (instances of the) services in the community, so as to fulfill the client's needs.

The orchestration engine is also in charge of *(i)* offering the correct set of actions to the client, according to the external schema of the target service, *(ii)* delegating the action chosen by the client to the service that offers it, and *(iii)* terminating the execution of (instances of) services in the community.

All this is hidden to the client, who interacts only with the service Composition System and is not aware that a composite service is being executed instead of a simple one.

In the figure we denote as *request-time* the moment in which the client presents his specification to the system and the target service is realized. Indeed, it is more natural to denote with the term *design-time* the moment in which the services in the community are designed. Therefore, the design time is conceptually different from the request time. However, this does not imply that the design time is antecedent to the request time since also the realized

target service can join (i.e., can be stored in) the community. Finally, we call *interaction-time* the moment in which the client interacts with an instance of the realized target service, orchestrated by the orchestration engine.

3.7 Discussion

In this chapter we have presented a general formal framework where services are characterized in terms of their behavioral description, abstractly represented as an (execution) tree. Services are, basically, programs, i.e., processes: trees have been used in the years in the research areas of process verification, and of process synthesis to abstractly denote programs and to study their properties (see the discussion in Section 2.3.2).

We identify two points of view for representing service behavior, i.e., internal and external to the services. A similar approach can be found in [69], where the authors talk about *(i) internal middleware* (of a service) to denote the middleware supporting the internal operations that allow message exchanges, and about *(ii) external middleware* to denote the middleware supporting the operations provided to the clients, such as those supported by SOA (see Section 1.2.2), to find a service and invoke it. Therefore, they take a perspective which is less abstract than ours and more oriented towards the practical applications. The two approaches are therefore complementary: we provide their approach with a conceptual substrate and they "justify" ours with an applicability perspective.

In the framework presented in this thesis, the service community is fixed, during both request-time and interaction-time, i.e., the services that are available during the synthesis of the composite service are also available during the execution of it. However, the service environment is highly dynamic: existing services may become obsolete very often and new services become available on a daily basis, especially in very dynamic *e*-market places, or in *e*-Business applications. Therefore, it may happen that a component service involved in the composition becomes unavailable, and new services become available, during the composite service execution. It is interesting to analyze and tackle the issues presented by a dynamic service community, which lead to the need of on-the-fly dynamic re-configuration, i.e., how to automatically re-configure the composite, running service by "substituting" a new component service, which is "compatible" (i.e., it exports at least the same behavior of the obsolete service) with the obsolete component one, and such that the behavior of the composite service is unaffected by this substitution. Open issues regard the notions of substitutability, adaptivity, compatibility which have not been deeply addressed in the service literature (see Section 2.2) and for which no agreed understanding and definition exist. Also, no techniques for automati-

cally achieving them have been developed. However, results from other areas, such as software component [155, 37] or Petri Nets [104] can be applied.

Finally, note that in this chapter we have introduced the concept of service instance, as an active occurrence of a service. In the rest of the thesis, wlog, we will assume that at most one active instance exists at a time, for each service; if more active instances correspond to the same external schema, we duplicate the external schema for each instance. We can do this because in our framework, the various active instances of a same schema run independently from the others. However, it could be of interest the case when more active instances corresponding to the same schema exist, and interact one with the other.

Chapter 4

Service Behavioral Description as FSM and Automatic Composition

In this chapter, we study and analyze the general framework presented in Chapter 3 in the case where behavioral descriptions (i.e., execution trees) of services admit a concise representation using a finite number of states. In this specific setting, we devise *sound, complete and terminating* techniques both to check for the existence of a composition, and to return a composition. The composition produced is finite state, and hence, as a collateral result of our synthesis technique, we show that if a composition exists then there exists one which is indeed finite state. The synthesis technique is based on reducing the problem of checking the existence of a composition into checking satisfiability of a formula expressed in Deterministic Propositional Dynamic Logic (DPDL), a well-known logic of programs developed to verify properties of program schemas [92]. We also analyze the computational complexity of the proposed algorithms. Specifically, our technique gives us an EXPTIME upper bound in worst-case computational complexity for the composition synthesis problem. While assessing that such bound is in fact tight is still open, we conjecture that the problem is indeed EXPTIME-hard. From a more practical point of view, it is easy to find cases in which the composition must be exponential in the size of the component services and the client specification, hence exponentiality is inherent to the problem. To the best of our knowledge, our work is the first attempt to provide a provably correct technique for the automatic synthesis of service composition, in a framework where the behavior of services is explicitly specified.

In Appendix A, we present an alternative representation of finitely representable services, using Situation Calculus [124], and discuss a (sound, com-

plete and terminating) technique for checking the existence and synthesizing a composition.

4.1 Services with Behavioral Description as Finite State Machines

In the previous chapter, we have not referred to any specific formalism for expressing service schemas. In what follows, we consider services whose schema (both internal and external) can be represented using only a *finite number of states*, i.e., using (deterministic) Finite State Machines (FSMs).

Several papers in the service literature adopt FSMs as the basic model of exported behavior of services [40, 30, 89, 69, 17]. Also, FSMs constitute the core of State Diagrams, which are one of the main components of UML [67] and are becoming a widely used formalism for specifying the dynamic behavior of entities.

Observe that the class of services that can be captured by FSMs represent an interesting set of services, that are able to carry on rather complex interactions with their clients, performing useful tasks. Indeed, many concrete, existing services can be abstractly represented as FSM. In order to show the usefulness and applicability of our approach, we discuss an abstract representation of a service already deployed and running as FSM.

Example 7 *Figure 4.1 shows an abstract representation as FSM of the service* Orbitz[1]*, that allows for arranging a travel. It offers a lot of functionalities, such as booking a car, a cruise, a flight, a hotel room. It also allows to choose among a large variety of customizable vacancies packages, i.e., combinations of hotel and car, hotel and flight, etc. Note, that the FSM in the figure shows only a part of Orbitz functionalities. Indeed, the purpose of this example is not to provide a detailed, and possibly difficult to read, abstract representation of the* Orbitz *service, but to make clear the applicability of our approach and in particular the use of (deterministic) finite state machines as a valid formal tool to abstractly represent existing services.*

In the initial state, the various functionalities that Orbitz provides are presented to the client, so, depending on his needs, he chooses among: `find_hotel&car`, `find_hotel`, `find_hotel&flight`, `find_car`, `find_cruise`, `sign_in` *if a member and* `register` *if not a member.*

[1]Strictly speaking, Orbitz is not a service, but a web portal (`http://www.orbitz.com`). The main difference between them is that web portals are oriented to humans, while services are oriented to applications. However, we agree with [17], on the fact that by understanding the behavior of web portals (i.e., the operations it provides through a browser and their semantics), it is possible to extrapolate the behavior of an “equivalent” service. This was exactly our approach in deriving the behavioral description of Orbitz as FSM.

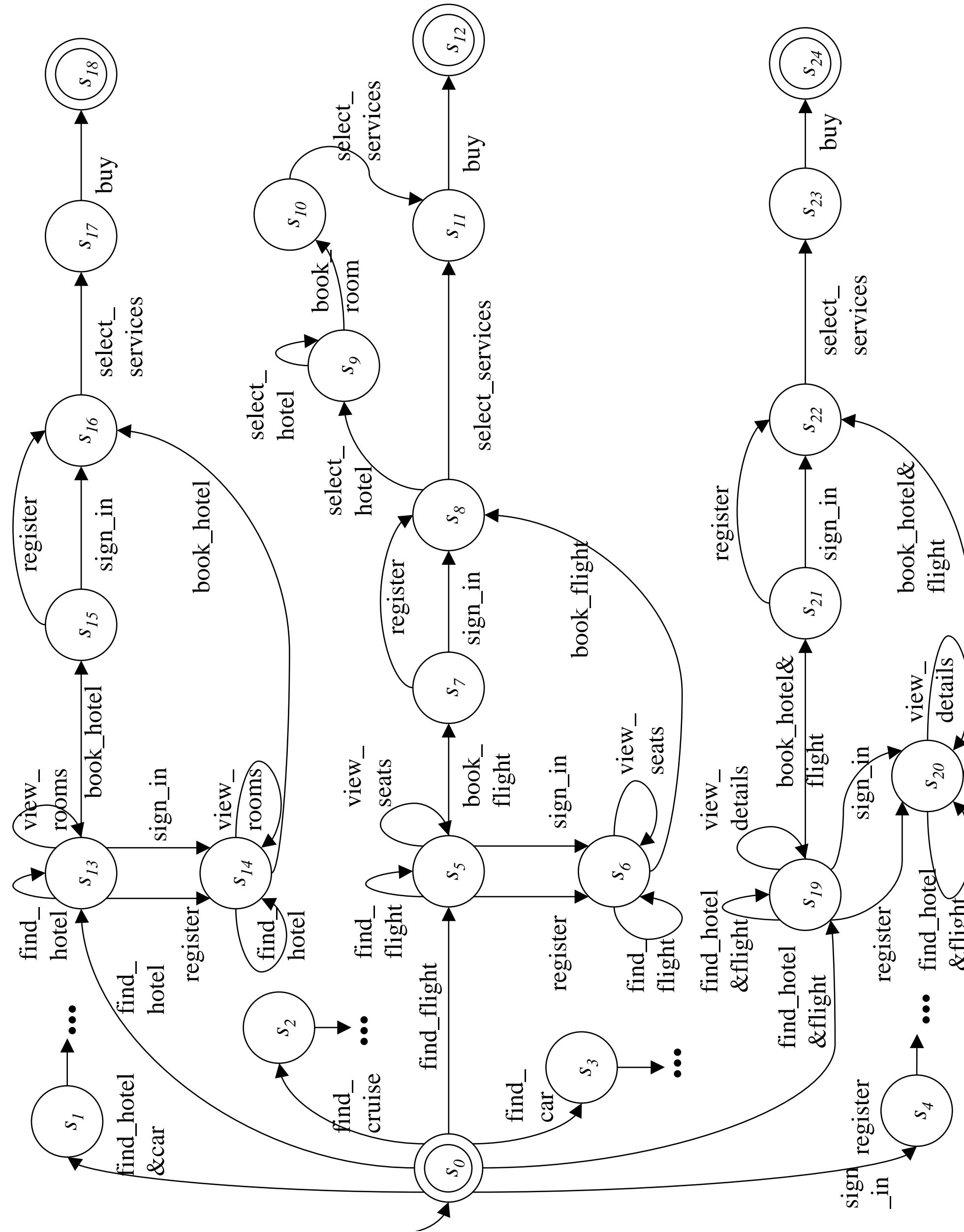


Figure 4.1: The Orbitz service modeled as Finite State Machine.

Note that at each step the client may always choose to end the current booking and to start another one. This corresponds to have, from almost[2] *each state, an arrow pointing to states $s_1, \ldots, s_5, s_{13}, s_{19}$ labeled with the suitable operation (e.g.,* `find_hotel&car` *for the arrow pointing to s_1,* `find_hotel` *for the arrow pointing to s_{13}, etc.). For sake of readability, such arrows are not shown in the figure. Also, from each state that follows a* `register` *or* `sign_in` *action it is possible to execute action* `sign_out`*, leading the computation back to state s_0.*

Assume that a client of such service wants to buy a plane ticket. Thus, he chooses to execute action `find_flight`*. Then, he has a set of choices: he can* (i) *execute action* `find_flight` *again,* (ii) *select a flight and view the various seats, in order to book later a precise seats on the plane (*`view_seats`*),* (iii) `sign_in` *if a member and* `register` *if not a member,* (iv) *book the selected flight (*`book_flight`*). Note that the service allows a client to perform first either the (*`book_flight`*) or the operations* `sign_in` *or* `register` *, but when execution arrives at the point represented by state s_8, the client must be logged in. At this point, the service offers the client with a list of hotels near the client destination: the client may choose and select one (possibly making several choices) (*`select_hotel` *action) and book a room (*`book_room` *action). Then, the client chooses one or more services (*`select_services` *action) such as booking a slot in the parking near the airport, booking a ticket on the city tour bus, etc. Finally, the client accepts* Orbits *conditions and* `buy`*s what he booked.* □

In the study we report here, we make the simplifying assumption that the number of instances of a service in the community that can be involved in the internal execution tree of another service is bounded and fixed a priori. In fact, wlog we assume that it is equal to one. If more instances correspond to the same external schema, we simply duplicate the external schema for each instance. Considering that the number of services in a community is finite, this implies that the overall number of instances orchestrated in executing a service is finite and bounded by the number of services belonging to the community. Within this setting, we show how to solve the problem of composition existence, and how to synthesize a composition that is a FSM. Instead, how to deal with an unbounded number of instances remains open for future work[3].

The fact that external schemas can be represented with a finite number of states means that we can factorize the sequence of actions executed up to a certain point into a finite number of states, which are sufficient to determine the future behavior of the service.

[2]The FSM is deterministic.

[3]Intuitively, it is very likely that the problems of composition existence and synthesis are undecidable in this case.

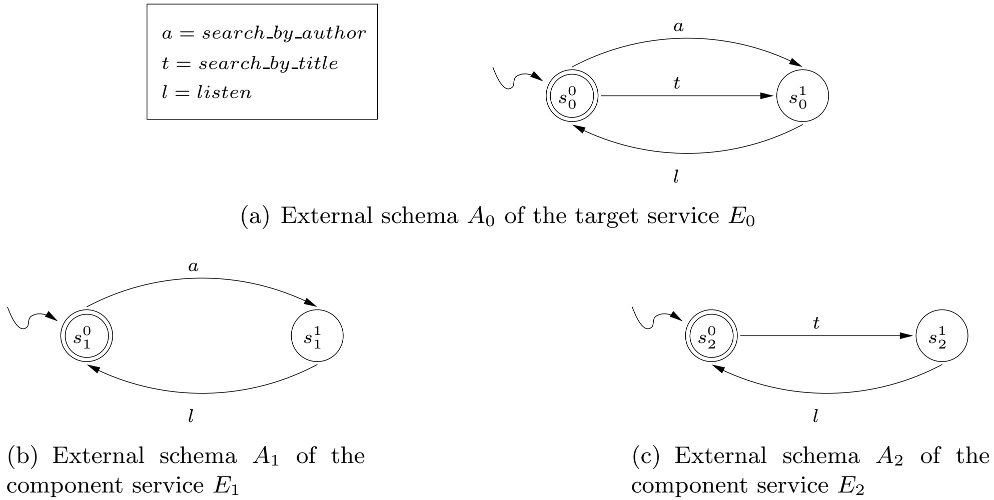
Definition 13 ((FSM) External Schema) Let C be a service community, whose alphabet of actions is Σ. Let E be a service in C. The external schema of E is a FSM $A_E^{ext} = (\Sigma, S_E, s_E^0, \delta_E, F_E)$, where:

- Σ is the alphabet of the FSM, which is the alphabet of the community;
- S_E is the set of states of the FSM, representing the finite set of states of the service E;
- s_E^0 is the initial state of the FSM, representing the initial state of the service;
- $\delta_E : S_E \times \Sigma \to S_E$ is the (partial) transition function of the FSM, that given a state s and an action a returns the state resulting from executing a in s;
- $F_E \subseteq S_E$ is the set of final states of the FSM, i.e., the states where the interactions with E can be legally terminated.

□

In what follows, for sake of clarity, we denote the FSM external schema of a service in C by means of a mapping A^{ext} from E to A_E^{ext}, where E is the set of abstract names of the services in the community C (so, in what follows, instead of writing A_E^{ext}, we will write $A^{ext}(E)$, where $E \in$ E). We use the mapping as a means of associating additional formal structure to the (names of) services in C, i.e., as a way to enforce the fact that the FSM external schema is associated to the service E.

Example 8 *Figure 4.2(a) shows the external schema of the target service E_0 described in Example 4, specified by the client as a FSM A_0. Figures 4.2(b) and 4.2(c) show the external schemas, represented as FSMs A_1 and A_2, respectively associated to component services E_1 and E_2. In other words, A_1 and A_2 are the external schemas of the services that should be composed in order to obtain a new service that behaves like E_0. In particular, E_1 allows for searching for a song by specifying its author(s) (action `search_by_author`) and for listening to the song selected by the client (action `listen`). Then, it allows for executing these actions again. E_2 behaves like E_1, but it allows for retrieving a song by specifying its title (action `search_by_title`).*
E_1 and E_2 belong to the same community of services C. For sake of simplicity, we assume that C is composed by E_1 and E_2 only, and therefore, the alphabet of actions of C is $\Sigma = \{\texttt{search_by_author}, \texttt{search_by_title}, \texttt{listen}\}$. According to our setting, also the actions in A_0 belong to Σ. □



(a) External schema A_0 of the target service E_0

(b) External schema A_1 of the component service E_1

(c) External schema A_2 of the component service E_2

Figure 4.2: Composition of services

The FSM $A^{ext}(E)$ is an external schema in the sense that it specifies an external execution tree $T^{ext}(E)$. Specifically, given $A^{ext}(E)$, we can define the tree $T(A^{ext}(E))$ inductively on the level of nodes in the tree, by making use of an auxiliary function $\sigma(\cdot)$ that associates to each node of the tree a state in the FSM. We proceed as follows:

- ε, as usual, is the root of $T(A^{ext}(E))$ and $\sigma(\varepsilon) = s_E^0$;
- if x is a node of $T(A^{ext}(E))$, and $\sigma(x) = s$, for some $s \in S_E$, then for each a such that $s' = \delta_E(s, a)$ is defined, $x \cdot a$ is a node of $T(A^{ext}(E))$ and $\sigma(x \cdot a) = s'$;
- x is final iff $\sigma(x) \in F_E$.

It is easy to see that with the above construction one can derive a mapping $T^{ext}(E)$ from $A^{ext}(E)$. This is equivalent to say that the tree $T(A^{ext}(E))$ obtained by unfolding $A^{ext}(E)$ coincides with the external execution tree $T^{ext}(E)$, since $T^{ext}(E) = T(E^{ext}) = T(A_E^{ext}) = T(A^{ext}(E))$.

Example 9 *Figure 4.3 shows (a portion of) the external execution tree $T(A_0)$ defined from A_0 by the mapping σ: each node of the tree is labeled with the*

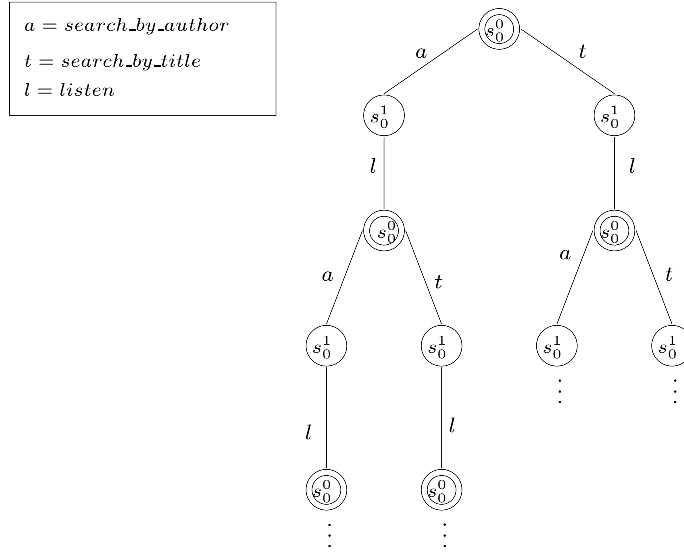


Figure 4.3: External execution tree $T(A_0)$ for the target service E_0

state of A_0 that σ associates to it. The mapping σ is defined as follows.

$$\begin{array}{l}
\sigma(\varepsilon) = s_0^0 \\
\sigma(a) = \sigma(t) = s_0^1 \\
\sigma(a \cdot l) = \sigma(t \cdot l) = s_0^0 \\
\sigma(a \cdot l \cdot a) = \sigma(a \cdot l \cdot t) = \sigma(t \cdot l \cdot a) = \sigma(t \cdot l \cdot t) = s_0^1 \\
\sigma(a \cdot l \cdot a \cdot l) = \sigma(a \cdot l \cdot t \cdot l) = \sigma(t \cdot l \cdot a \cdot l) = \sigma(t \cdot l \cdot t \cdot l) = s_0^0 \\
\ldots
\end{array}$$

σ maps over s_0^1 the nodes of the tree that represent strings ending either by a or by t; it maps over s_0^0 the root and the nodes of the tree associated to strings ending by l. Note that $T(A_0)$ coincides with the external execution tree $T^{ext}(E_0)$ of Figure 3.2. That is, $T^{ext}(E_0)$ has a finite representation as a FSM.
The external execution trees $T(A_1)$ and $T(A_2)$ for the FSMs A_1 and A_2, respectively, can be defined similarly and coincide with the external execution trees of Figure 3.3(b). Finally, note that in general there may be several (equivalent) FSMs that specify the same execution tree. □

Since we have assumed that each service in the community can contribute to the internal execution tree of another service with at most one instance, in specifying internal execution trees we do not need to distinguish between services and service instances. Hence, when the community C is formed by n services $E_1, \ldots, E_n$, it suffices to label the internal execution tree of a service

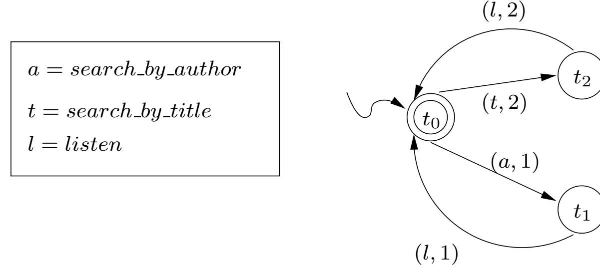


Figure 4.4: MFSM M_0 internal schema for the target service E_0

E by the action that caused the transition and a subset of $[n] = \{1, \ldots, n\}$ that identifies which services in the community have contributed in executing the action. The empty set $\emptyset$ is used to (implicitly) denote `this`. We define internal schemas that have a finite number of states as follows.

Definition 14 ((MFSM) Internal Schema) Let C be a service community, whose alphabet of actions is Σ. Let E be a service in C. Its internal schema is a Mealy FSM (MFSM) $A_E^{int} = (\Sigma, 2^{[n]}, S_E^{int}, s_E^{0\ int}, \delta_E^{int}, \omega_E^{int}, F_E^{int})$, where:

- $\Sigma, S_E^{int}, s_E^{0\ int}, \delta_E^{int}, F_E^{int}$, have the same meaning as in Definition 13;
- $2^{[n]}$ is the output alphabet of the MFSM, which is used to denote which service(s) executes each action;
- $\omega_E^{int} : S_E^{int} \times \Sigma \rightarrow 2^{[n]}$ is the output function of the MFSM, that, given a state s and an action a, returns the subset of services in C that executes action a when service E is in state s; if such a set is empty then `this` is implied; we assume that the output function ω_E^{int} is defined exactly when δ_E^{int} is so.

□

In what follows, for sake of clarity, we denote the MFSM internal schema of a service in C by means of a mapping A^{int} from E to A_E^{int}, where E is the set of abstract names of the services in the community C (so, in what follows, instead of writing A_E^{int}, we will write $A^{int}(E)$, where $E \in$ E). We use the mapping as a means of associating additional formal structure to the (names of) services in C, i.e., as a way to enforce the fact that the MFSM internal schema is associated to the service E.

Example 10 *Figure 4.4 shows a possible internal schema for the target service E_0. It is represented as a MFSM M_0. The output function ω^{int} is defined as follows:*

$$\omega^{int}(t_0, a) = \{1\} \qquad \omega^{int}(t_0, t) = \{2\}$$
$$\omega^{int}(t_1, l) = \{1\} \qquad \omega^{int}(t_2, l) = \{2\}$$

□

The MFSM $A^{int}(E)$ is an internal schema in the sense that it specifies an internal execution tree $T^{int}(E)$. Given $A^{int}(E)$ we, again, define the internal execution tree $T(A^{int}(E))$ by induction on the level of the nodes, by making use of an auxiliary function $\sigma^{int}(\cdot)$ that associates each node of the tree with a state in the MFSM, as follows:

- ε is, as usual, the root of $T(A^{int}(E))$ and $\sigma^{int}(\varepsilon) = s_E^{0\ int}$;
- if x is a node of $T(A^{int}(E))$, and $\sigma^{int}(x) = s$, for some $s \in S_E^{int}$, then for each a such that $s' = \delta_E^{int}(s, a)$ is defined, $x \cdot a$ is a node of $T(A^{int}(E))$ and $\sigma^{int}(x \cdot a) = s'$;
- if x is a node of $T(A^{int}(E))$, and $\sigma^{int}(x) = s$, for some $s \in S_E^{int}$, then for each a such that $\omega_E^{int}(s, a)$ is defined (i.e., $\delta_E^{int}(s, a)$ is defined), the edge $(x, x \cdot a)$ of the tree is labeled by $\omega_E^{int}(s, a)$;
- x is final iff $\sigma^{int}(x) \in F_E^{int}$.

It is easy to see that with the above construction one can derive a mapping $T^{int}(E)$ from $A^{int}(E)$. This is equivalent to say that the tree $T(A^{int}(E))$ obtained by unfolding $A^{int}(E)$ coincides with the internal execution tree $T^{int}(E)$, since $T^{int}(E) = T(E^{int}) = T(A_E^{int}) = T(A^{int}(E))$.

Example 11 *Figure 4.5 shows a portion of the internal execution tree $T(M_0)$ defined from M_0, shown in Figure 4.4. Each node of the tree is labeled with the state of M_0 that σ^{int} associates to it. The mapping σ^{int} is defined as follows.*

$$\sigma^{int}(\varepsilon) = t_0$$
$$\sigma^{int}(a) = t_1$$
$$\sigma^{int}(t) = t_2$$
$$\sigma^{int}(a{\cdot}l) = \sigma^{int}(t{\cdot}l) = t_0$$
$$\sigma^{int}(a{\cdot}l{\cdot}a) = \sigma^{int}(t{\cdot}l{\cdot}a) = t_1$$
$$\sigma^{int}(a{\cdot}l{\cdot}t) = \sigma^{int}(t{\cdot}l{\cdot}t) = t_2$$
$$\sigma^{int}(a{\cdot}l{\cdot}a{\cdot}l) = \sigma^{int}(a{\cdot}l{\cdot}t{\cdot}l) = \sigma^{int}(t{\cdot}l{\cdot}a{\cdot}l) = \sigma^{int}(t{\cdot}l{\cdot}t{\cdot}l) = t_0$$
$$\ldots$$

σ^{int} maps over t_1 the nodes of the tree that represent strings ending by a, and over t_2 the nodes that represent strings ending by t; it maps over t_0 the root and the nodes of the tree associated to strings ending by l.

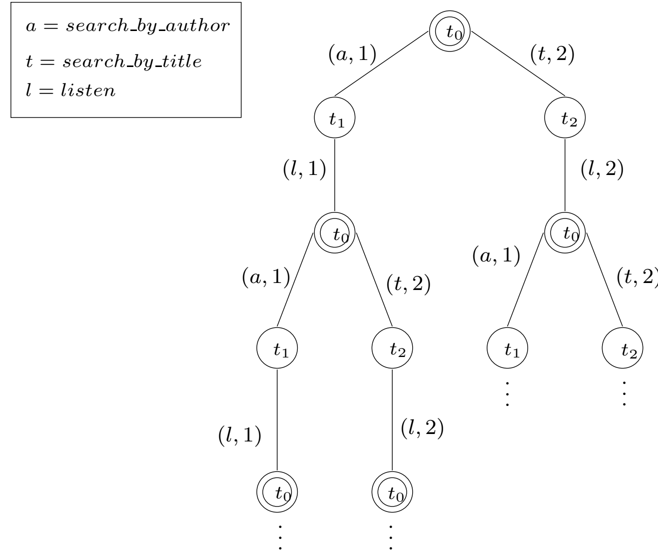


Figure 4.5: Internal execution tree $T(M_0)$ for the target service E_0

Note that $T(M_0)$ is equal to the internal execution tree $T^{int}(E)$ of Figure 3.3(a) (up to renaming the labels (E_i, e_i) with i). That is, $T^{int}(E)$ has a finite representation as a MFSM. Therefore, M_0 is a specification of an internal execution tree that conforms to the external execution tree specified by the FSM A_0 of Figure 4.2(a). Finally, note that in general, an external FSM and its corresponding internal MFSM may have different structures, i.e., the MFSM internal schema cannot be obtained by simply labeling the FSM external schema with services in the community. □

Given a service E whose external schema is an FSM and whose internal schema is an MFSM, checking whether E is well formed, i.e., whether the internal execution tree conforms to the external execution tree, can be done using standard finite state machine techniques. Similarly for checking the coherency of E wrt a community of services whose external schemas are FSMs. In this thesis, we do not go into the details of these problems, and instead we concentrate on composition.

4.2 Preliminaries on Deterministic Propositional Dynamic Logic

Propositional Dynamic Logics (PDLs) are a family of modal logics specifically developed for representing and reasoning about computer programs [66, 92].

They capture the properties of the interaction between programs and propositions that are independent of the domain of computation. In this section, we provide a brief overview of a logic of this family, namely Deterministic Propositional Dynamic Logic (DPDL), which we will use in the rest of the thesis. More details can be found in [81].

Syntactically, a DPDL formula is constituted by expressions of two sorts: *actions*, also called *programs*, and propositions. Arbitrary propositions, also called *formulas*, and programs, denoted by ϕ and r respectively, are built by starting from a set $\mathcal{P}$ of atomic propositions and a set $\mathcal{A}$ of *deterministic* atomic actions as follows:

$$\begin{array}{lcl} \phi & \longrightarrow & \texttt{true} \mid \texttt{false} \mid P \mid \neg\phi \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid \langle r\rangle\phi \mid [r]\phi \\ r & \longrightarrow & a \mid r_1 \cup r_2 \mid r_1; r_2 \mid r^* \mid \phi? \end{array}$$

where P is an atomic proposition in $\mathcal{P}$, a is an atomic action in $\mathcal{A}$. We also use the abbreviation

$$\phi_1 \rightarrow \phi_2 \qquad \text{for} \qquad \neg\phi_1 \lor \phi_2$$

Intuitively, DPDL formulas are composed from atomic propositions by applying arbitrary propositional connectives ($\neg, \land, \lor$) and the modal operators $\langle r\rangle\phi$ and $[r]\phi$. The meaning of the latter two is, respectively, that there exists an execution of r reaching a state where ϕ holds, and that all terminating executions of r reach a state where ϕ holds. As far as arbitrary programs, $r_1 \cup r_2$ means "choose non deterministically between r_1 and r_2"; $r_1; r_2$ means "first execute r_1 then execute r_2"; r^* means "execute r a non deterministically chosen number of times (zero or more)"; $\phi?$ means "test ϕ: if it is true proceed else fail".

The main difference between DPDL (and modal logics in general) and classical logics relies on the use of modalities. A modality is a connective which takes a formula (or a set of formulas) and produces a new formula with a new meaning. Examples of modalities are $\langle r\rangle$ and $[r]$. For instance, the classical logic operator $\neg$ is a connective, which takes a formula p and produces a new formula $\neg p$. The only difference is that in classical logic, the truth value of $\neg p$ is uniquely determined by the value of p, instead modalities are not truth-functional. Because of modalities, the semantics of DPDL formulas (and modal logics) is defined over a structure, namely a Kripke structure.

The semantics of DPDL is based on the notion of *deterministic* Kripke structure. A deterministic Kripke structure is a triple of the form $\mathcal{I} = (\Delta^\mathcal{I}, \{a^\mathcal{I}\}_{a\in\mathcal{A}}, \{P^\mathcal{I}\}_{P\in\mathcal{P}})$, where $\Delta^\mathcal{I}$ denotes a non-empty set of states (also called worlds); $\{a^\mathcal{I}\}_{a\in\mathcal{A}}$ is a family of partial *functions* $a^\mathcal{I} : \Delta^\mathcal{I} \rightarrow \Delta^\mathcal{I}$ from elements of $\Delta^\mathcal{I}$ to elements of $\Delta^\mathcal{I}$, each of which denotes the state transitions caused by an atomic program a; $P^\mathcal{I} \subseteq \Delta^\mathcal{I}$ denotes all the elements of $\Delta^\mathcal{I}$ were P is true.

The basic semantic relation "a formula ϕ holds at a state s of a structure $\mathcal{I}$", is written $\mathcal{I}, s \models \phi$, and is defined by induction on the form of ϕ:

$\mathcal{I}, s \models \texttt{true}$	always
$\mathcal{I}, s \models \texttt{false}$	never
$\mathcal{I}, s \models P$	iff $s \in P^{\mathcal{I}}$
$\mathcal{I}, s \models \neg\phi$	iff $\mathcal{I}, s \not\models \phi$
$\mathcal{I}, s \models \phi_1 \wedge \phi_2$	iff $\mathcal{I}, s \models \phi_1$ and $\mathcal{I}, s \models \phi_2$
$\mathcal{I}, s \models \phi_1 \vee \phi_2$	iff $\mathcal{I}, s \models \phi_1$ or $\mathcal{I}, s \models \phi_2$
$\mathcal{I}, s \models \langle r \rangle \phi$	iff there is s' such that $(s, s') \in r^{\mathcal{I}}$ and $\mathcal{I}, s' \models \phi$
$\mathcal{I}, s \models [r]\phi$	iff for all s', $(s, s') \in r^{\mathcal{I}}$ implies $\mathcal{I}, s' \models \phi$

where the family $\{a^{\mathcal{I}}\}_{a \in \mathcal{A}}$ is systematically extended so as to include, for every program r, the corresponding function $r^{\mathcal{I}}$ defined by induction on the form of r:

$$
\begin{array}{lcl}
a^{\mathcal{I}} : \Delta^{\mathcal{I}} \to \Delta^{\mathcal{I}} & & \\
(r_1 \cup r_2)^{\mathcal{I}} & = & r_1^{\mathcal{I}} \cup r_2^{\mathcal{I}} \\
(r_1; r_2)^{\mathcal{I}} & = & r_1^{\mathcal{I}}; r_2^{\mathcal{I}} \\
(r^*)^{\mathcal{I}} & = & (r^{\mathcal{I}})^* \\
(\phi?)^{\mathcal{I}} & = & \{(s, s) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid \mathcal{I}, s \models \phi\}
\end{array}
$$

The set of subformulas of a DPDL formula ϕ that play some role in establishing the truth-value of ϕ is given by the Fischer-Ladner closure [66].

A structure $\mathcal{I} = (\Delta^{\mathcal{I}}, \{a^{\mathcal{I}}\}_{a \in \mathcal{A}}, \{P^{\mathcal{I}}\}_{P \in \mathcal{P}})$ is called a *model* of a formula ϕ if there exists a state $s \in \Delta^{\mathcal{I}}$ such that $\mathcal{I}, s \models \phi$. A formula ϕ is *satisfiable* if there exists a model of ϕ, otherwise the formula is *unsatisfiable*. A formula ϕ is *valid* in the structure $\mathcal{I}$ if for all $s \in \Delta^{\mathcal{I}}$, $\mathcal{I}, s \models \phi$. We call *axioms* formulas that are used to select the interpretations of interest. Formally, a structure $\mathcal{I}$ is a model of an axiom ϕ, if ϕ is valid in $\mathcal{I}$. A structure $\mathcal{I}$ is a model of a finite set of axioms Γ if $\mathcal{I}$ is a model of all axioms in Γ. An axiom is satisfiable if it has a model and a finite set of axioms is satisfiable if it has a model. We say that a finite set Γ of axioms *logically implies* a formula ϕ, written $\Gamma \models \phi$, if ϕ is valid in every model of Γ. It is easy to see that satisfiability of a formula ϕ as well as satisfiability of a finite set of axioms Γ can be reformulated by means of logical implication, as $\emptyset \not\models \neg\phi$ and $\Gamma \not\models \texttt{false}$ respectively.

DPDL enjoys two properties that are of particular interest. The first is the *tree model property*, which says that every model of a formula can be unwound to a (possibly infinite) tree-shaped model (considering domain elements as nodes and partial functions interpreting actions as edges). The second is the *small model property*, which says that every satisfiable formula admits a finite model whose size (in particular the number of domain elements) is at most exponential in the size of the formula itself.

Reasoning in DPDL (and, in general, in PDLs) has been thoroughly studied from the computational point of view. In particular, the following theorem holds [13]:

Theorem 15 *Satisfiability in DPDL is EXPTIME-complete.* □

4.3 Automatic Service Composition

In this section we address the problem of checking the existence and of synthesizing a composite service in the FSM-based framework introduced above. We show that if a composition exists then there is one such that the internal schema is constituted by a MFSM, and we show how to actually synthesize such a MFSM, when one exists. The basic idea of our approach consists in reducing the problem of composition into satisfiability of a suitable formula of Deterministic Propositional Dynamic Logic (DPDL).

4.3.1 Checking Existence of a Composition

In this section we show how to solve the problem of composition existence, both in the general case and in the context of our running example.

Given the target service E_0 whose external schema is a FSM A_0 and a community of services formed by n component services $E_1, \ldots, E_n$ whose external schemas are FSM $A_1, \ldots, A_n$ respectively, we build a DPDL formula Φ as follows. As set of atomic propositions $\mathcal{P}$ in Φ we have *(i)* one proposition s_j for each state s_j of $A_j, j = 0, \ldots, n$, denoting whether A_j is in state s_j; *(ii)* propositions $F_j, j = 0, \ldots, n$, denoting whether A_j is in a final state; and *(iii)* propositions $\mathsf{moved}_j, j = 1, \ldots, n$, denoting whether (component) A_j performed a transition. As set of atomic actions $\mathcal{A}$ in Φ we have the actions in Σ (i.e, $\mathcal{A} = \Sigma$).

Example 12 *The set $\mathcal{P}$ of atomic propositions is defined as follows:*

$$\mathcal{P} = \{s_0^0, s_0^1, s_1^0, s_1^1, s_2^0, s_2^1, F_0, F_1, F_2, \mathsf{moved}_1, \mathsf{moved}_2\}$$

with the above meaning.

The set $\mathcal{A}$ of deterministic atomic actions, which by construction coincides with the alphabet of the community, is: $\mathcal{A} = \Sigma = \{a, t, l\}$, where:

- *a denotes action* `search_by_author`
- *t denotes action* `search_by_title`
- *l denotes action* `listen`*.*

□

Universal assertions are encoded by the master modality $[u]$.

The DPDL formula Φ, encoding the composition problem, is built as a conjunction of the following formulas.

- Formulas representing $A_0 = (\Sigma, S_0, s_0^0, \delta_0, F_0)$:
  - $[u](s \rightarrow \neg s')$ for all pairs of states $s \in S_0$ and $s' \in S_0$, with $s \neq s'$; these say that propositions representing different states are disjoint (cannot be true simultaneously).
  - $[u](s \rightarrow \langle a \rangle \texttt{true} \wedge [a]s')$ for each a such that $s' = \delta_0(s, a)$; these encode the transitions of A_0.
  - $[u](s \rightarrow [a]\texttt{false})$ for each a such that $\delta(s, a)$ is not defined; these say when a transition is not defined.
  - $[u](F_0 \leftrightarrow \bigvee_{s \in F_0} s)$; this highlights final states of A_0.

Example 13 *To encode the master modality $[u]$, we set*

$$u = (a \cup t \cup l)^*$$

i.e., as the reflexive and transitive closure of the union of all atomic actions in $\mathcal{A}$. In other words, u represents the iteration of a non deterministic choice among all the possible atomic actions. Indeed, we recall that $[u]\phi$, where ϕ is a proposition, asserts that ϕ holds after any regular expression involving a, t, l.

Formulas capturing the external schema A_0 are as follows.

$$[u](s_0^0 \rightarrow \neg s_0^1)$$

This formula states that FSM A_0 can never be simultaneously in the two states s_0^0 and s_0^1. Note that it is equivalent to state $[u](s_0^1 \rightarrow \neg s_0^0)$.

$$\begin{array}{lcl} [u](s_0^0 & \rightarrow & \langle a \rangle \texttt{true} \wedge [a]s_0^1) \\ [u](s_0^0 & \rightarrow & \langle t \rangle \texttt{true} \wedge [t]s_0^1) \\ [u](s_0^1 & \rightarrow & \langle l \rangle \texttt{true} \wedge [l]s_0^0) \end{array}$$

These formulas encode the transitions that A_0 can perform. For example, the first formula asserts that, for all possible sequences of actions, if A_0 is in state s_0^0, the FSM allows for searching an mp3 *file by author, i.e., it*

can execute action a, and it necessarily moves to state s_0^1. Analogously for the other formulas.

$$[u](s_0^0 \;\rightarrow\; [l]\texttt{false})$$
$$[u](s_0^1 \;\rightarrow\; [a]\texttt{false})$$
$$[u](s_0^1 \;\rightarrow\; [t]\texttt{false})$$

These formulas encode the transitions that are not defined on A_0. For example, the first formula asserts that, for all possible sequences of actions, it is never possible to execute action `listen` *when the FSM is in state s_0^0.*

$$[u](F_0 \;\leftrightarrow\; s_0^0)$$

Finally, this formula asserts that s_0^0 is a final state for A_0. □

- Formulas encoding each component FSM $A_i = (\Sigma, S_i, s_i^0, \delta_i, F_i)$:
  - $[u](s \rightarrow \neg s')$ for all pairs of states $s \in S_i$ and $s' \in S_i$, with $s \neq s'$; these again say that propositions representing different states are disjoint.
  - $[u](s \rightarrow [a](\mathsf{moved}_i \wedge s' \vee \neg\mathsf{moved}_i \wedge s))$ for each a such that $s' = \delta_i(s, a)$; these encode the transitions of A_i, conditioned to the fact that the component A_i is actually required to make an a-transition in the composition.
  - $[u](s \rightarrow [a](\neg\mathsf{moved}_i \wedge s))$ for each a such that $\delta_i(s, a)$ is not defined; these say that when a transition is not defined, A_i cannot be asked to execute it in the composition, and therefore A_i does not change state.
  - $[u](F_i \leftrightarrow \bigvee_{s \in F_i} s)$; this highlights final states of A_i.

Example 14 *Formulas capturing the external schema A_1.*

$$[u](s_1^0 \;\rightarrow\; \neg s_1^1)$$

This formula has an analogous meaning as that relative to A_0.

$$[u](s_1^0 \;\rightarrow\; [a](\mathsf{moved}_1 \wedge s_1^1 \vee \neg\mathsf{moved}_1 \wedge s_1^0))$$
$$[u](s_1^1 \;\rightarrow\; [l](\mathsf{moved}_1 \wedge s_1^0 \vee \neg\mathsf{moved}_1 \wedge s_1^1))$$

These formulas encode the transitions which are defined and required to A_1. As an example, the first formula asserts that for all possible sequences of actions, if the FSM A_1 is in s_1^0, then after action a has been executed, necessarily one of the following conditions must hold: either it

is A_1 that performed the transition and therefore it moved to state s_1^1, or the transition has been performed by another FSM, hence A_1 did not move and remained in the current state s_1^0.

$$\begin{array}{lcl} [u](s_1^0 & \rightarrow & [l](\neg \mathsf{moved}_1 \wedge s_1^0)) \\ [u](s_1^0 & \rightarrow & [t](\neg \mathsf{moved}_1 \wedge s_1^0)) \\ [u](s_1^1 & \rightarrow & [a](\neg \mathsf{moved}_1 \wedge s_1^1)) \\ [u](s_1^1 & \rightarrow & [t](\neg \mathsf{moved}_1 \wedge s_1^1)) \end{array}$$

These formulas encode transitions which are not defined. For example, the first formula states that if A_1 is in state s_1^0 and it is asked to execute l, it does not move, and therefore it remains in state s_1^0; this holds for all possible sequences of actions. Note that the situation when the FSM does not move is different from the situation when it loops on a state: indeed, in the latter case the transition is defined whereas in the former it is not.

Finally, the formula

$$[u](F_1 \leftrightarrow s_1^0)$$

asserts that state s_1^0 is final for FSM A_1.

Formulas capturing the external schema A_2.

Such formulas are analogous to the previous ones, therefore, we will just report them, without further comments.

$$\begin{array}{lcl} [u](s_2^0 & \rightarrow & \neg s_2^1) \\ [u](s_2^0 & \rightarrow & [t](\mathsf{moved}_2 \wedge s_2^1 \vee \neg \mathsf{moved}_2 \wedge s_2^0)) \\ [u](s_2^1 & \rightarrow & [l](\mathsf{moved}_2 \wedge s_2^0 \vee \neg \mathsf{moved}_2 \wedge s_2^1)) \\ [u](s_2^0 & \rightarrow & [l](\neg \mathsf{moved}_2 \wedge s_2^0)) \\ [u](s_2^0 & \rightarrow & [a](\neg \mathsf{moved}_2 \wedge s_2^0)) \\ [u](s_2^1 & \rightarrow & [t](\neg \mathsf{moved}_2 \wedge s_2^1)) \\ [u](s_2^1 & \rightarrow & [a](\neg \mathsf{moved}_2 \wedge s_2^1)) \\ [u](F_2 & \leftrightarrow & s_2^0) \end{array}$$

□

- Finally, formulas encoding domain independent conditions:
  - $s_0^0 \wedge \bigwedge_{i=1,\ldots,n} s_i^0$; this says that initially all services are in their initial state; note that this formula is not prefixed by $[u](\cdot)$.
  - $[u](\langle a \rangle \texttt{true} \rightarrow [a] \bigvee_{i=1,\ldots,n} \mathsf{moved}_i)$, for each $a \in \Sigma$; these say that at each step at least one of the component FSM has moved.

– $[u](F_0 \rightarrow \bigwedge_{i=1,\ldots,n} F_i)$; this says that when the target service is in a final state also all component services must be in a final state.

Example 15 *The domain independent conditions that must hold for the overall composition are as follows.*

$$s_0^0 \wedge s_1^0 \wedge s_2^0$$

It asserts that all services start from their initial states.

$$\begin{array}{lcl} [u](\langle a\rangle\texttt{true} & \rightarrow & [a](\mathsf{moved}_1 \vee \mathsf{moved}_2)) \\ [u](\langle t\rangle\texttt{true} & \rightarrow & [t](\mathsf{moved}_1 \vee \mathsf{moved}_2)) \\ [u](\langle l\rangle\texttt{true} & \rightarrow & [l](\mathsf{moved}_1 \vee \mathsf{moved}_2)) \end{array}$$

Each formula expresses that at each step either A_1 or A_2 or both move. For example, the first one asserts that for all possible execution sequences, if a is executed then necessarily either E_1 or E_2 (or both) performed a.

Finally,

$$[u](F_0 \rightarrow F_1 \wedge F_2)$$

states that if the composite service is in a final state, both component services must be in a final state: the composite service may legally terminate only if also all the component services can. □

Note that the DPDL formula Φ, built as above, presents the following properties:

- the Kleene star * is used only in the master modality, i.e., in expressions of the form $[p^*]\Psi$, where Ψ is a DPDL formula which do not contain *;
- the master modality has the form $u = (a_1 \cup \ldots \cup a_n)^*$, where $a_1 \ldots a_n$ are DPDL atomic actions;
- the $\langle\rangle$ modality appears only in front of the atomic proposition `true`;
- both modalities $[]$ (when it is not used as master modality) and $\langle\rangle$ are used in expressions of the form $[a]\Psi$ or $\langle a\rangle\Psi$, where Ψ is a DPDL formula and a is an atomic DPDL action.

We will make use of such observations in Chapter 7.

Lemma 16 *If there exists a composition of E_0 wrt $E_1, \ldots, E_n$, then the DPDL formula Φ, constructed as above, is satisfiable.*

Proof. Suppose that there exists some internal schema (without restriction on its form) ${E_0}^{int}$ which is a composition of E_0 wrt $E_1, \ldots, E_n$. Let $T_{int} = T({E_0}^{int})$ be the internal execution tree defined by ${E_0}^{int}$.

Then for the target service E_0 and each component service E_i, $i = 1, \ldots n$, we can define mappings σ and σ_i from nodes in T_{int} to states of A_0 and A_i, respectively, by induction on the level of the nodes in T_{int} as follows.

- base case: $\sigma(\varepsilon) = s_0^0$ and $\sigma_i(\varepsilon) = s_i^0$.
- inductive case: let $\sigma(x) = s$ and $\sigma_i(x) = s_i$, and let the node $x \cdot a$ be in T_{int} with the edge $(x, x \cdot a)$ labeled by (a, I), where $I \subseteq [n]$ and $I \neq \emptyset$ (notice that `this` may not occur since T_{int} is specified by a composition). Then we define
$$\sigma(x \cdot a) = s' = \delta_0(s, a)$$
and
$$\sigma_i(x \cdot a) = \begin{cases} {s_i}' = \delta_i(s_i, a) & \text{if } i \in I \\ s_i & \text{if } i \notin I \end{cases}$$

Once we have σ and σ_i in place we can define an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \{a^{\mathcal{I}}\}_{a \in \Sigma}, \{P^{\mathcal{I}}\}_{P \in \mathcal{P}})$ for Φ as follows:

- $\Delta^{\mathcal{I}} = \{x \mid x \in T_{int}\}$;
- $a^{\mathcal{I}} = \{(x, x \cdot a) \mid x, x \cdot a \in T_{int}\}$, for each $a \in \Sigma$;
- $s^{\mathcal{I}} = \{x \in T_{int} \mid \sigma(x) = s\}$, for all propositions s corresponding to states of A_0;
- $s_i^{\mathcal{I}} = \{x \in T_{int} \mid \sigma_i(x) = s_i\}$, for all propositions s_i corresponding to states of A_i;
- $\mathsf{moved}_i^{\mathcal{I}} = \{x \cdot a \mid (x, x \cdot a) \text{ is labeled by } I \text{ with } i \in I\}$, for $i = 1, \ldots, n$;
- $F_0^{\mathcal{I}} = \{x \in T_{int} \mid \sigma(x) = s \text{ with } s \in F_0\}$;
- $F_i^{\mathcal{I}} = \{x \in T_{int} \mid \sigma_i(x) = s_i \text{ with } s_i \in F_i\}$, for $i = 1, \ldots, n$.

Since T_{int} is a composition of E_0 wrt $E_1, \ldots, E_n$, it is easy to check that the interpretation $\mathcal{I}$ built as above, is a (tree-like) model for Φ and that, therefore, Φ is satisfiable.

□

Lemma 17 *Any model of the DPDL formula Φ, constructed as above, denotes a composition of E_0 wrt $E_1, \ldots, E_n$.*

Proof. Suppose Φ is satisfiable. For the tree model property, there exists a tree-like model for Φ: let $\mathcal{I} = (\Delta^{\mathcal{I}}, \{a^{\mathcal{I}}\}_{a\in\Sigma}, \{P^{\mathcal{I}}\}_{P\in\mathcal{P}})$ be such a model. From $\mathcal{I}$ we can build an internal execution tree T_{int} for E_0 as follows.

- the nodes of the tree are the elements of $\Delta^{\mathcal{I}}$; actually, since $\mathcal{I}$ is tree-like we can denote the elements in $\Delta^{\mathcal{I}}$ as nodes of a tree, using the same notation that we used for internal/external execution tree;
- nodes x such that $x \in F_0^{\mathcal{I}}$ are the final nodes;
- if $(x, x \cdot a) \in a^{\mathcal{I}}$ and for all $i \in I$, $x \cdot a \in moved_i^{\mathcal{I}}$ and for all $j \notin I$, $x \cdot a \notin moved_j^{\mathcal{I}}$, then $(x, x \cdot a)$ is labeled by (a, I).

It is straightforward to show that: *(i)* T_{int} conforms to $T(A_0)$, *(ii)* T_{int} delegates all actions to the services of $E_1, \ldots, E_n$, and *(iii)* T_{int} is coherent with $E_1, \ldots, E_n$. Since we are not placing any restriction on the kind of specification allowed for internal schemas, it follows that there exists an internal schema E_{int} that is a composition of E_0 wrt $E_1, \ldots, E_n$. □

Theorem 18 *The DPDL formula Φ, constructed as above, is satisfiable if and only if there exists a composition of E_0 wrt $E_1, \ldots, E_n$.*

Proof. Straightforward, from Lemma 16 and 17. □

By construction, the size of the DPDL formula Φ is polynomially related to A_0 and $A_1, \ldots, A_n$. Hence, from the EXPTIME-completeness of satisfiability in DPDL and from Theorem 18 we get the following complexity result.

Theorem 19 *Checking the existence of service composition can be done in EXPTIME.* □

We do not have a tight lower bounds for the complexity of the composition existence problem, however, we argue that exponentiality is inherent to the problem, since, from a practical point of view, it is easy to find cases in which the composition must be exponential in the size of the component services and the client specification.

4.3.2 Synthesizing a Composition

In the previous section we have shown that we are able to check the existence of a composition by checking satisfiability of a DPDL formula Φ encoding the FSM external schema of the target service, FSM external schemas of the services in the community and a number of domain independent conditions. In this section we extend our technique to actually synthesize a composition

which is a FSM. Specifically, we present an algorithm that returns a composition, if one exists, and returns a special symbol (**nil**), denoting that no composition exists, otherwise.

Intuitively, by Theorem 18, if Φ is satisfiable then it admits a model, which is exactly the the composition, we want to synthesize. Conversely, if Φ is not satisfiable, no model exists, therefore, the component FSM $A_1, \ldots, A_n$ cannot be composed in order to build an internal schema for the target FSM A_0. Note that Theorem 18 says nothing about compositions which are finite state machines. However, because of the small model property, from the PDPL formula Φ one can always obtain a model which is at most exponential in the size of Φ. From such a model one can extract an internal schema for E_0 that is a composition of E_0 wrt $E_1, \ldots, E_n$, and which has the form of a MFSM.

Definition 20 (Mealy Composition) Given a finite model $\mathcal{I}_f = (\Delta^{\mathcal{I}_f}, \{a^{\mathcal{I}_f}\}_{a\in\Sigma}, \{P^{\mathcal{I}_f}\}_{P\in\mathcal{P}})$, we define *Mealy composition* an MFSM $A_c = (\Sigma, 2^{[n]}, S_c, s_c^0, \delta_c, \omega_c, F_c,)$, built as follows:

- $S_c = \Delta^{\mathcal{I}_f}$;
- $s_c^0 = d_0$ where $d_0 \in (s_0^0 \wedge \bigwedge_{i=1,\ldots,n} s_i^0)^{\mathcal{I}_f}$;
- $s' = \delta_c(s, a)$ iff $(s, s') \in a^{\mathcal{I}_f}$;
- $I = \omega_c(s, a)$ iff $(s, s') \in a^{\mathcal{I}_f}$ and for all $i \in I$, $s' \in moved_i^{\mathcal{I}_f}$ and for all $j \notin I$, $s' \notin moved_j^{\mathcal{I}_f}$;
- $F_c = F_0^{\mathcal{I}_f}$.

□

As a consequence of this, we get the following results.

Theorem 21 *If there exists a composition of E_0 wrt $E_1, \ldots, E_0$, then there exists a Mealy composition whose size is at most exponential in the size of the FSM external schemas A_0, $A_1, \ldots, A_n$ of E_0, $E_1, \ldots, E_n$ respectively.*

Proof. By Theorem 18, if A_0 can be obtained by composing $A_1, \ldots, A_n$, then the DPDL formula Φ constructed as in Section 4.3.1 is satisfiable. In turn, if Φ is satisfiable, for the small-model property of DPDL there exists a model $\mathcal{I}_f$ of size at most exponential in Φ, and hence in A_0 and $A_1, \ldots, A_n$. From $\mathcal{I}_f$ we can construct a MFSM A_c as in Definition 20. The internal execution tree $T(A_c)$ defined by A_c satisfies all the conditions required for A_c to be a composition, namely: *(i)* $T(A_c)$ conforms to $T(A_0)$, *(ii)* $T(A_c)$ delegates all actions to the services of $E_1, \ldots, E_n$, and *(iii)* $T(A_c)$ is coherent with $E_1, \ldots, E_n$. □

```
AUTOMATIC SERVICE COMPOSITION
 1  INPUT:  A_0              /* FSM external schema of target service */
 2          A_1 ... A_n      /* FSM external schema of services in the community */
 3
 4  OUTPUT: if (a composition of A_0 wrt A_1 ... A_n exists)
 5             then return a Mealy composition of A_0 wrt A_1 ... A_n
 6             else return nil
 7
 8  begin
 9   Φ := FSM2DPDL(A_0, A_1, ... A_n);
10   I_f := DPDLTableau(Φ);
11   if (I_f == nil)
12      then return nil
13      else A_c := Extract_MFSM(I_f);
14           C_min := Minimize(A_c);
15           return C_min;
16  end
```

Figure 4.6: The algorithm for synthesizing a Mealy composition

Theorem 22 *Any finite model of the DPDL formula* Φ *denotes a Mealy composition of* E_0 *wrt* $E_1, \ldots, E_n$.

Proof. By construction, observing that the construction of the Mealy composition from a finite model is semantic-preserving. □

Figure 4.6 shows our algorithm for synthesizing a Mealy composition, which consists of the following steps. First (`line 9`), the DPDL formula Φ is built, exploiting the `FSM2DPDL` function, as a conjunction of formulas encoding: *(i)* the FSM external schema of the target service requested by the client, *(ii)* the FSM external schemas of the (available) services of the community, and *(iii)* domain independent conditions. In other words, it encodes all the FSM external schemas of (real and virtual) services participating in the composition. Essentially, such an encoding aims at characterizing which service in the community "moves" in correspondence with each transition of the target service, so that general domain independent conditions are satisfied. The novelty and peculiarity of our approach to service composition is exactly this: we delegate to one or more services in the community the execution of *each* action which is present in the transition system specified by the client, since only in a second moment it is known which actions will be chosen by the client for execution (and the composite service should be able to execute *any* action chosen by the client). Satisfiability of Φ is then checked (`line 10`, function `DPDLTableau`) exploiting standard tableau algorithms [52, 8] that return a fi-

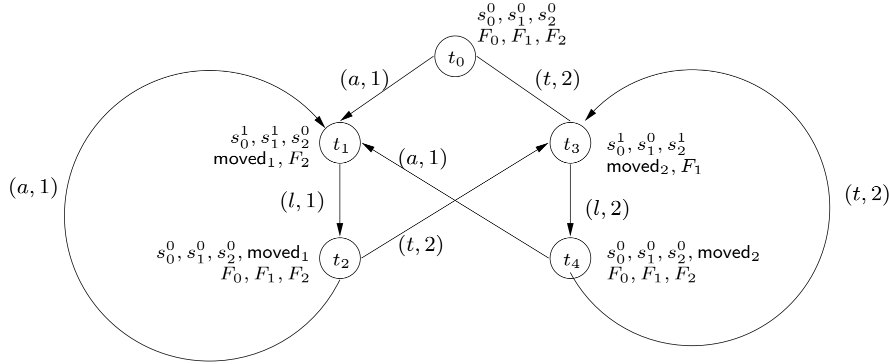


Figure 4.7: Finite model $\mathcal{I}_f$ for Φ.

nite model, if and only if one exists. If Φ is not satisfiable, no model exists, and our algorithm returns **nil** (`line 12`). Otherwise, from a finite model a Mealy composition is built, (function `Extract_MFSM`, `line 13`), according to Definition 20. Intuitively, the transformation from a finite model $\mathcal{I}_f$ to a Mealy Machine A_c consists in discarding from each state of $\mathcal{I}_f$ the information about the current state of (the FSM external schema of) each component service, therefore keeping in A_c only the information about which service is in a final state and which one "moves". Note that, in general, after this transformation, some states of A_c can be redundant, since they contain the same information: in other words, a final step minimizing A_c can be performed (`line 14`, function `Minimize`), and the minimal Mealy composition C_{min} is returned (`line 15`). In Section 7.4, we present a prototype tool that implements such steps.

Example 16 *Let Φ be the DPDL formula encoding A_0, A_1, A_2 and the domain independent conditions, built as in Section 4.3.1. Let $\mathcal{I}_f$ be the finite model (i.e., the Kripke Stricture) obtained using a tableau technique for DPDL, and shown in Figure 4.7. $\mathcal{I}_f$ is defined as $\mathcal{I}_f = (\Delta^{\mathcal{I}_f}, \{a^{\mathcal{I}_f}\}_{a\in\Sigma}, \{P^{\mathcal{I}_f}\}_{P\in\mathcal{P}})$, where:*

$\Delta^{\mathcal{I}_f} = \{t_0, t_1, t_2, t_3, t_4\}$
$a^{\mathcal{I}_f} = \{(t_0, t_1), (t_2, t_1), (t_4, t_1)\}$
$t^{\mathcal{I}_f} = \{(t_0, t_3), (t_2, t_3), (t_4, t_3)\}$
$l^{\mathcal{I}_f} = \{(t_1, t_2), (t_3, t_4)\}$
$(s_0^0)^{\mathcal{I}_f} = \{t_0, t_2, t_4\}$
$(s_0^1)^{\mathcal{I}_f} = \{t_1, t_3\}$
$(s_1^0)^{\mathcal{I}_f} = \{t_0, t_2, t_3, t_4\}$
$(s_1^1)^{\mathcal{I}_f} = \{t_1\}$
$(s_2^0)^{\mathcal{I}_f} = \{t_0, t_1, t_2, t_4\}$
$(s_2^1)^{\mathcal{I}_f} = \{t_3\}$
$\mathsf{moved}_1^{\mathcal{I}_f} = \{t_1, t_2\}$
$\mathsf{moved}_2^{\mathcal{I}_f} = \{t_3, t_4\}$
$F_0^{\mathcal{I}_f} = \{t_0, t_2, t_4\}$
$F_1^{\mathcal{I}_f} = \{t_0, t_2, t_3, t_4\}$
$F_2^{\mathcal{I}_f} = \{t_0, t_1, t_2, t_4\}$

Each state t_i of the model is associated with the atomic propositions in $\mathcal{P}$ that hold in that state, according to $\mathcal{I}_f$. For example, consider state t_0:

$\mathcal{I}_f$ imposes that $s_0^0 \wedge s_1^0 \wedge s_2^0 \wedge F_0 \wedge F_1 \wedge F_2$ holds in t_0. Therefore, we take t_0 as initial state for the MFSM. However, since the same propositions hold also in t_2 and t_4, we could have as well as chosen either t_2 or t_4 as initial state. For sake of readability, in Figure 4.7 we have associated to each state of $\mathcal{I}_f$ simply the list of atomic propositions that are true. Additionally, note that the DPDL encoding does not pose any constraint on the value of moved_i predicates in the initial state of the model: their value has been arbitrarily chosen to be `false`.[4]

Given $\mathcal{I}_f = (\Delta^{\mathcal{I}_f}, \{a^{\mathcal{I}_f}\}_{a\in\Sigma}, \{P^{\mathcal{I}_f}\}_{P\in\mathcal{P}})$ for Φ, we define a Mealy composition $A_c = (\Sigma, 2^{[2]}, S_c, s_c^0, \delta_c, \omega_c, F_c)$, representing the internal schema of the target service, as follows:

- $S_c = \{t_0, t_1, t_2, t_3, t_4\}$;
- $s_c^0 = t_0$, *where* $t_0 \in (s_0^0 \wedge s_1^0 \wedge s_2^0)^{\mathcal{I}_f}$.
- δ_c *is defined as:*

$$\begin{array}{llll} \delta_c(t_0,a) = t_1 & \delta_c(t_1,l) = t_2 & \delta_c(t_2,a) = t_1 & \delta_c(t_4,a) = t_1 \\ \delta_c(t_0,t) = t_3 & \delta_c(t_3,l) = t_4 & \delta_c(t_2,t) = t_3 & \delta_c(t_4,t) = t_3 \end{array}$$

- ω_c *is defined as:*

$$\begin{array}{llll} \omega_c(t_0,a) = \{1\} & \omega_c(t_1,l) = \{1\} & \omega_c(t_2,a) = \{1\} & \omega_c(t_4,a) = \{1\} \\ \omega_c(t_0,t) = \{2\} & \omega_c(t_3,l) = \{2\} & \omega_c(t_2,t) = \{2\} & \omega_c(t_4,t) = \{2\} \end{array}$$

- $F_c = \{t_0, t_2, t_4\}$.

This example shows also that the finite state machine associated to the finite model of Φ is in general not minimal. Indeed, the minimal MFSM C_{min} is shown in Figure 4.8. Note that C_{min} coincides with the MSFM shown in Figure 4.4 which, as shown in Example 11, is an internal schema for the target service E_0 of our running example. □

Before concluding the section, we first discuss our technique on a slightly more complex example of composition synthesis. Then, we show an example of non-existence of a composition.

[4] Note also the model for the DPDL formula Φ is deterministic, as it should be. Non determinism could have been introduced by the operator $\langle\rangle$. However, we are guaranteed that no atomic action a connects a state s_1 with two different (target) states s_2 and s_3, because $\langle\rangle$ operates only on atomic actions a and it appears only in front of the atomic proposition `true`. Indeed, if a related s_1 with s_2 and s_3, such target states would actually be the same, since s_2 and s_3 are associated with the same atomic proposition `true`.

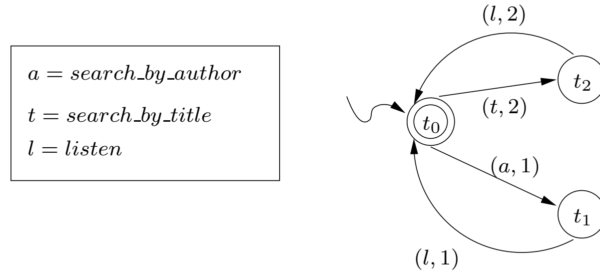


Figure 4.8: Minimal MFSM C_{min} associated to $\mathcal{I}_f$.

Example 17 *Figure 4.9 shows a traveling service composition scenario. Figure 4.9 (a) shows a community $C_\mathcal{T}$ composed by four services, having the following behavior:* (i) `S1: book _airtravel` *allows for booking (zero or more times) first a plane ticket and then a limousine to arrive to the airport;* (ii) `S2: book_accommodation` *allows for choosing whether to reserve a hotel room or a flat in a residence, and in the second case it allows for listing the facilities near the chosen residence (e.g., supermarkets), finally it allows for booking a shuttle zero or more times, to/from the chosen accommodation;* (iii) `S3: book_traintravel` *allows for booking (zero or more times) first a train ticket and then a limousine to arrive to the train station*[5]*;* (iv) `S4: register_event` *allows for registering to an event. Such an event may be a tourist event, e.g., Venice film festival, a conference, etc.*

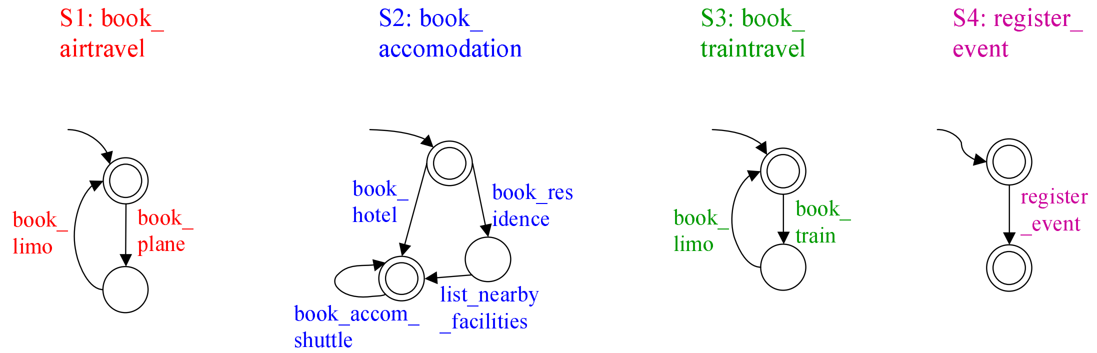
Figure 4.9(b) shows the FSM external schema $A_\mathcal{T}$ of a target service requested by a client, that wants to book a plane or a train ticket and a hotel room (note, the client does not desire to book a residence flat) in any order; then he wants to book a limousine to go to the departure place, a shuttle for traveling around, starting from the arrival destination (at least once, possibly more times) and finally to register to the event for which he makes the travel.

By applying our composition technique, we get the MFSM $M_\mathcal{T}$ shown in Figure 4.9(c). It is easy to see that $M_\mathcal{T}$ is indeed a Mealy composition of $A_\mathcal{T}$ wrt $C_\mathcal{T}$:

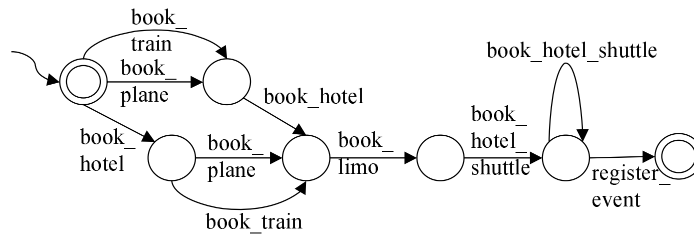
1. *$M_\mathcal{T}$ conforms to $A_\mathcal{T}$, since the FSM obtained from $M_\mathcal{T}$ by dropping the part of the labeling denoting services, is equivalent to (i.e., accepts the same language as) $A_\mathcal{T}$.*

2. *$M_\mathcal{T}$ is coherent with $C_\mathcal{T}$, since* (i) *all actions are in the alphabet of $C_\mathcal{T}$, and* (ii) *all accepting runs of $M_\mathcal{T}$ determine accepting runs over each one of the component services. Therefore,*

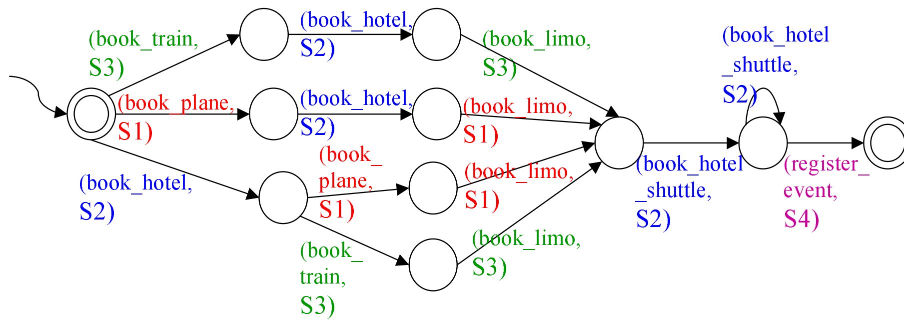
[5]Note that both `S1: book_airtravel` and `S2: book_traintravel` provide the same action `book_limo`.



(a) service community $C_{\mathcal{T}}$

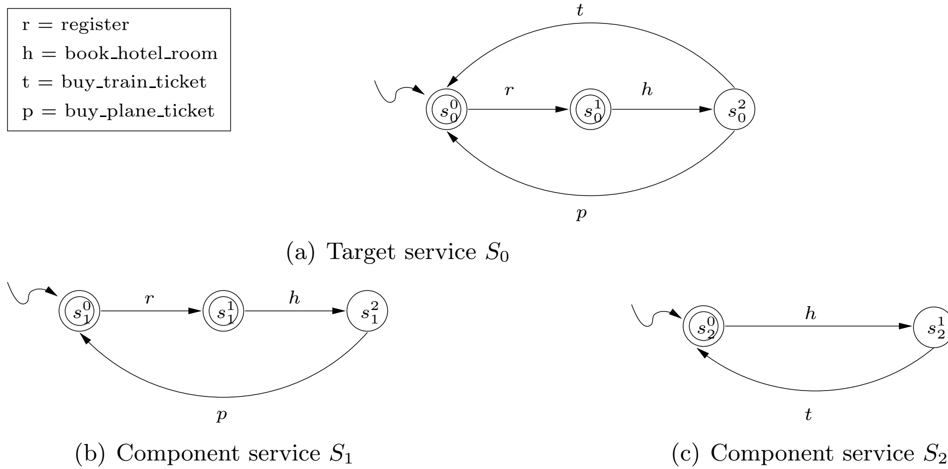


(b) FSM external schema $A_{\mathcal{T}}$ of the target service



(c) Mealy composition $M_{\mathcal{T}}$

Figure 4.9: Traveling service Composition Scenario



(a) Target service S_0

(b) Component service S_1

(c) Component service S_2

Figure 4.10: Non-existence of a composition

3. *$M_{\mathcal{T}}$ correctly delegates all actions to the services of $C_{\mathcal{T}}$ (note also that* `this` *does not appear in $M_{\mathcal{T}}$).*

Finally, note that, the form of the $M_{\mathcal{T}}$ is different from the form of $A_{\mathcal{T}}$: this happens because the `book_limo` *action is provided by two different services. In other words, in general, in order to compute a Mealy composition it does not suffices to label the transitions of the FSM external schema of the target service.* □

Example 18 *Figure 4.10 (a) shows a service for conference participation: first, the client registers to a conference (action* `registration`*), he books a hotel room (action* `book_hotel_room`*), finally, he chooses whether to buy a plane ticket (action* `buy_plane_ticket`*) or a train ticket (action* `buy_train_ticket`*). Then, the client can either end the interactions or execute all of them again. Note that he can also choose to execute the first action only. Such a service represents the target service. Figure 4.10 (b) and (c) show services S_1 and S_2, respectively, constituting the community of services. S_1 allows for executing actions* `registration`*, followed by* `book_hotel_room`*, and* `buy_train_ticket`*, possibly in a repeated way. S_2 allows for executing the sequence of actions* `book_hotel_room`*, and* `buy_plain_ticket`*, possibly repeatedly.*

In this example, no composition exists*: indeed, the annotation of* `book_hotel_room` *depends on which action is executed* next*, since actions* `buy_train_ticket` *and* `buy_plane_ticket` *are executed by different services and states s_1^2 and s_2^1 in Figure 4.10 (b) and (c), respectively, are not final, therefore none of S_1 and S_2 can terminate on these states. Therefore, our composition algorithm returns* **nil**.

In [71] the authors propose a framework in which they are able to compute composition also in this case. Their idea is to let the client choose in advance (e.g., in state s_0^1) which action to execute after `book_hotel_room`, *by adding the so-called look-ahead. In this way, the action delegation is done by considering not only the history of past activities, but also the sequence of future activities. Therefore, the framework presented in [71], although inspired by the one discussed in this thesis, has profound differences with it: client requests are essentially languages and compositions are finite state automata over* strings. *Therefore, using the taxonomy proposed in Chapter 2, their approach can be classified as "language-based".* □

4.4 Discussion

The contribution of this chapter is an effective technique for automatic service composition. In particular, we specialize the general framework of Chapter 3 to the case where services are specified by means of (deterministic) finite state machines, and we present a technique that, given a specification of a target service, i.e., specified by a client, and a set of available services, *(i)* synthesizes a composite service that uses only the available services, *(ii)* fully captures the target one, and *(iii)* it is still described as a finite state machine. Actually, specific representations of services are often based on finite state formalisms, e.g., in [111] services are represented as statecharts, and in [40] services are modeled as Mealy machines. We claim that indeed most services have a behavior which can be abstractly represented as finite state machines, once it is singled out (in the abstract) the computations that such services can execute.

Our approach to automatic composition has two notable features:

1. The composition is based on the ability of executing the available component services concurrently, and of controlling in a suitable way how such services are interleaved to serve the client.

2. The client request is not a specification of a (single) desired execution, but a set of possibly non terminating executions organized in an execution tree, whose nodes correspond to sequences of transitions executed so far and whose successor nodes represent the choices available to the client to choose from what to do next. In other words, the client specifies the so-called *transition system* of the activities he is interested in doing. The ability of expressing a client specification as a transition systems realizes the natural client requirement that his decisions on which action to execute next depend on the outcome of previously executed actions and of other information which he cannot foresee at the time when he

specifies his requests. If either the available services or the client specification are not expressed as transition systems, the client would not have any influence over the sequence of actions executed by the composite service; instead his choices would be made once and for all before the composition is performed.

Both of these features are quite distinctive of our approach, and set the stage for a quite advanced form of composition: to the best of our knowledge, here we present the first algorithm for automatic composition of services in a framework where both the available services and the client specification are characterized by a behavioral description expressed as finite state machine. Our technique is *sound*, *complete* and *terminating*: if a composition of the available component services realizing the client specification exists, then our composition algorithm terminates returning one such a composition. Otherwise, it terminates reporting the non-existence of a composition. We also study the computational complexity of our technique, and we show that it runs in exponential time with respect to the size of the input state machines. Note that only few proposals in the literature follow our idea that the composition involves the concurrent executions of several services. In particular, the most related ones are [40, 123]: they have in common with our proposal the fact that the services are seen as white-boxes and hence they can be interleaved if needed. The composition deals with suitably controlling such an interleaving so as to realize the client request. As discussed in Section 2.2, most work on composition is based on the idea of *sequentially* composing the available services, which are considered as black boxes, and hence atomically executed [5, 98, 105, 153, 2], while the proposal in [108] lies somehow in between of such approaches. The second feature, i.e., that the client request is a specification of the transition system that the client wants to be able to execute, is, to the best of our knowledge, unique to our proposal. Indeed even [71, 40, 108] actually focus on realizing a single execution fulfilling the client request. Notice that such an execution may depend on conditions to be verified at run time, but not on further choices made by the client. Only the proposal in [123] has some similarities with ours: indeed, there, the client goal is expressed in a specific branching-time logic, that allows to specify alternative paths of execution under the control of the client. Though, their goals are still essentially based on having a main execution to follow, plus some side paths that are typically used to resolve exceptional circumstances.

Chapter 5

Loosely Specified Target Services

In this chapter we extend the framework presented in Chapters 3 and 4, to the case where the target service presents don't care non-determinism. We analyze such a situation and study the issues it raises wrt the problem of composition. Finally, we develop sound, complete and terminating techniques for checking the existence and building a composition.

5.1 Underspecified Target Service

In the framework presented in Chapters 3 and 4 the client request is given in terms of a target service and in particular its external schema, i.e., the behavioral description of the service he would like to interact with. Since the client request is *completely specified*, it determines a *single* target service, that the service community is asked to realize. In the present chapter we remove such a completeness assumption and allow a client to provide a loose request, i.e., that denotes an underspecified external schema. The incompleteness in the external schema of the target service shows up in the form of don't care non-determinism. Such non-determinism is called *angelic*: Section 5.4 explains more about such terminology. For each set of non-deterministic transitions that can be executed at a certain point of the computation, the client "does not care" which transition is executed next, since any one is "good" for him, and he leaves the choice on the next transition to the (system implementing the algorithm for) composition synthesis. Such choices may depend on several factors, for instance, on the availability of services in the community to which actions can be delegated. In general, for each set of non-deterministic transitions, several choices concerning action delegation can be made. Therefore, an *underspecified* client request determines a *set of* target services and the

community is asked to realize one (any one) in such a set. Observe that when a transition in a non-deterministic set is chosen, the (possibly deterministic) transitions following it along its path are of course executed: different choices may lead to different subsequent transitions (see Figure 5.1).

Finally, note that at specific points, determined in the client request, the client get again control over the execution.

The presence of underspecification in the client request denotes a situation which is very common in practice, when the client has no particular constraints on which transitions are executed next. It is also a first attempt to cope with dynamicity in the service community.

Definition 23 (Underspecified External Schema) An underspecified external schema E^{ext} is a *nondeterministic* FSM $\mathcal{A}^{ext}(E) = (\Sigma, S_0, s_0^0, \delta_0, F_0)$, where:

- Σ is the alphabet of the FSM;
- S_0 is the set of states;
- s_0^0 is the initial state;
- $\delta_0 : S_0 \times \Sigma_0 \rightarrow 2^{S_0}$ is a partial function that given a state and an action returns the set of possible successor states;
- $F_0 \subseteq S_0$ is the set of final states.

□

Definition 24 (Underspecified Target Service) Let C be a service community and let Σ be its alphabet. An underspecified target service E_0 is a client request denoting an underspecified external schema ${E_0}^{ext}$. □

Again, in general, a client request may express desired properties regarding functional and non-functional aspects of the target service schema, implementation and deployment features. As discussed in Section 3.5, we require that the client specifies (at least) the external schema of the service he desires to use. However, in what follows, we assume for simplicity that a client service specifies *only* the external schema of a target service and therefore, when clear from the context, we use the term "underspecified target service" to denote the "underspecified external schema (of an underspecified target service)".

The following example illustrate the concept of underspecified target service.

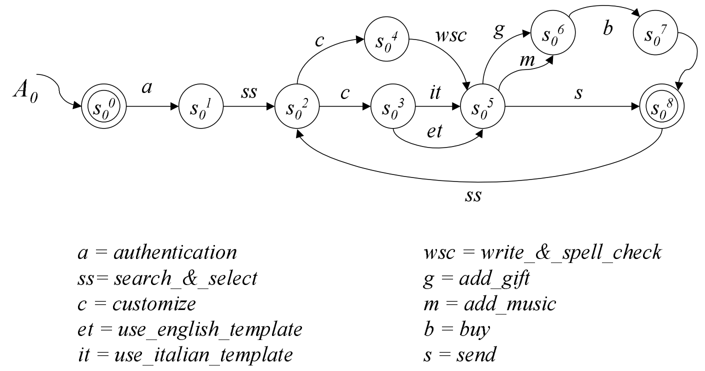


Figure 5.1: Underspecified target service $\mathcal{A}_0$

Example 19 *Figure 5.1 shows an underspecified target service* $\mathcal{A}_0$ *for composing and sending* e-cards. *Specifically, after being authenticated (*`authentication` *action), the client wants to search for an* e-card *and select it (*`search_&_select` *action). At this point the client specifies a nondeterministic choice between two possible paths: along one path he first* `customize`*s an* e-card *(i.e., he selects the stationery for the message, such as font, background, etc.) and then he writes the message from scratch and checks the spelling of it (*`write_&_spell_check` *action); along the other path he* `customizes`*s the* e-card *and then he composes the message by selecting a predefined template, which can be either English or Italian (actions* `use_english_template` *and* `use_italian_template`*, respectively): the choice of which template to use is left to the client. The client "doesn't care" which path is followed and lets the composition synthesis free to specify which sequence of action to execute. Next, there is another choice point, and in this case it is again the client who chooses what to do next: either* (i) *he* `send`*s the* e-card*, or* (ii) *he selects and attaches a gift (*`add_gift` *action) or some music (*`add_music` *action), pays it (*`buy`*) and* `send`*s the* e-card*. Finally, the client chooses whether to stop executing the services or send another* e-card *by performing the action* `search_&_select`*.* □

Observe that in our framework it is not possible to apply to the underspecified external schema of Definition 23 (in general to any external and internal schema represented as FSM) transformations from non-deterministic to deterministic FSM (and vice-versa), since the semantics of it would be completely changed and distorted, and therefore the algorithm for composition synthesis may not work properly. The fact that the c-transitions of Figure 5.1 end into two different states s_0^3 and s_0^4 is not casual, but it expresses a specific constraints of the client. As already discussed in the example, in the un-

derspecified target service of Figure 5.1, the client desires either to follow he "upper" c-transition and then a wsc transition, *or* to follow the "lower" c-transition, and then choose between it or et. Consider now a different situation, resulting when the c-transitions collapse into one, and therefore, state s_0^4 collapses into state s_0^3 and wsc transition originates from s_0^3. Such a (determinized) specification expresses that the client wants first to execute c and then to choose between wsc, it and et. In the two situations the semantics of the client specification is completely different!

A nondeterministic FSM $\mathcal{A}^{ext}(E_0)$, representing an underspecified target service E_0 specifies a *set* $\mathcal{T}(\mathcal{A}^{ext}(E_0))$ of external execution trees, defined as in Definition 2. We recall that for every action a belonging to the alphabet Σ of the community, and that can be executed at the point represented by a node x, there is a (single) successor node $x{\cdot}a$. Specifically, each execution tree in $\mathcal{T}(\mathcal{A}^{ext}(E_0))$ is obtained by unfolding the FSM and while doing so, resolving the nondeterminism by choosing a single successor state for each transition; this generates a (deterministic), possibly infinite tree, whose edges are labeled by Σ_0 and whose nodes corresponding to final states of $\mathcal{A}^{ext}(E_0)$ are annotated as final.

Formally, similarly to what done in Section 4.1, given $\mathcal{A}^{ext}(E_0)$, we define each execution tree $T_i(\mathcal{A}^{ext}(E_0))$ inductively on the level of nodes in the tree, by making use of an auxiliary function $\sigma^i_{ang}(\cdot)$ that associates to each node of $T_i(\mathcal{A}^{ext}(E_0))$ a state in the FSM $\mathcal{A}^{ext}(E_0)$. Intuitively, each different $\sigma^i_{ang}(\cdot)$ corresponds to a different way of resolving the non-determinism in the FSM, by unfolding it. The function $\sigma^i_{ang}(\cdot)$ is defined as follows:

- ε, as usual, is the root of $T_i(\mathcal{A}^{ext}(E_0))$ and $\sigma^i_{ang}(\varepsilon) = s_0^0$;
- let x be a node of $T_i(\mathcal{A}^{ext}(E_0))$, and let $\sigma^i_{ang}(x) = s$, for some $s \in S_0$, then two situations may happen:
  - for each a such that $s' = \delta_0(s, a)$ is *uniquely* defined, i.e., there does not exist another state s'' different from s' such that $s'' = \delta_0(s, a)$, then $x \cdot a$ is a node of $T_i(\mathcal{A}^{ext}(E_0))$ and $\sigma^i_{ang}(x \cdot a) = s'$;
  - for each a such that there exists two different states s' and s'' such that both $s' = \delta_0(s, a)$ and $s'' = \delta_0(s, a)$ are defined, then $x \cdot a$ is a node of $T_i(\mathcal{A}_0)$ and either $\sigma^i_{ang}(x \cdot a) = s'$ or $\sigma^i_{ang}(x \cdot a) = s''$, but not both[1];
- x is final iff $\sigma^i_{ang}(x) \in F_0$.

Analogously to the discussion in Section 4.1, it is easy to see that with the above construction, from $\mathcal{A}^{ext}(E_0)$, one can derive a set $\mathcal{T}^{ext}(E_0)$ of mappings, i.e., of execution trees associated to E_0. This is equivalent to say that

[1]Note that the choice is done arbitrarily.

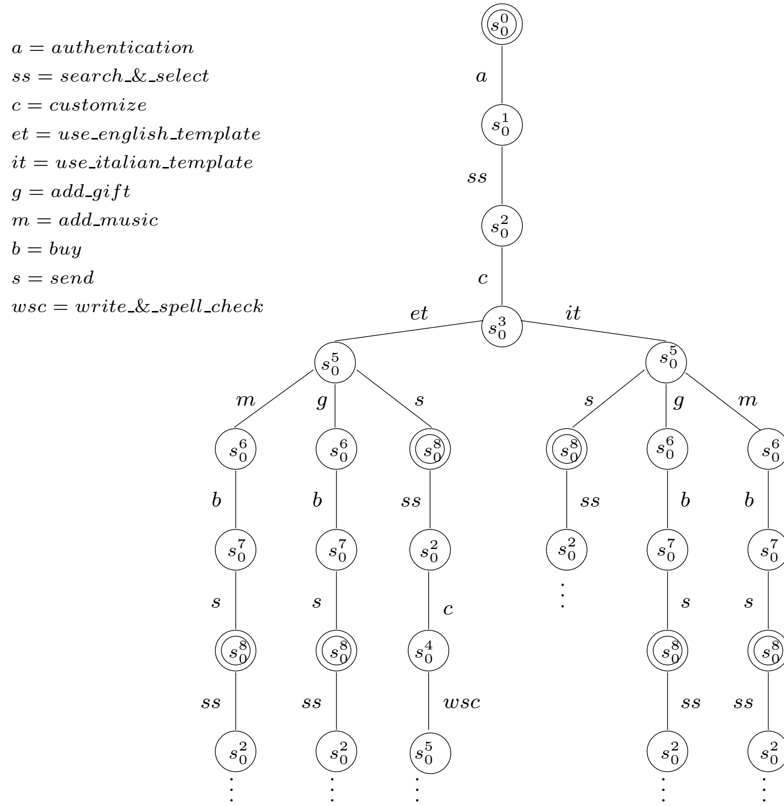


Figure 5.2: External execution tree $T(\mathcal{A}_0)$

each tree $T_i(\mathcal{A}^{ext}(E_0))$, obtained by unfolding $\mathcal{A}^{ext}(E_0)$ and by resolving the non-determinism in a different way, coincides with an external execution tree $T_i^{ext}(E_0)$ in $\mathcal{T}^{ext}(E_0)$.

Example 20 *Figure 5.2 shows (a portion of) $T(\mathcal{A}_0)$, which is an (i.e., one of the many) external execution trees defined from $\mathcal{A}_0$ (cf. Example 19) by a mapping σ_{ang}. Each node in the figure is labeled with the state of $\mathcal{A}_0$ that σ_{ang} associates to it. We will not go over the whole definition of σ_{ang}, because this should be already clear from Example 9 in Section 4.1 and from Figure 5.2 itself. We want to make the following observations:*

1. *For every action o that can be executed at the point represented by a node x, there is a (single) successor node $x{\cdot}o$.*

2. *Action* `customize` *(c in the figure) labels two edges: the first edge ends into a node corresponding to state s_3 in the FSM $\mathcal{A}_0$ and the second edge ends into a node corresponding to state s_4. Since in $\mathcal{A}_0$ there*

is a nondeterministic choice involving c-transitions, this means that in the two cases the non-determinism is resolved differently: indeed, the first occurrence of action `customize` *is followed by a choice between actions* `use_english_template` *and* `use_italian_template`*; the second occurrence is followed by action* `write_&_spell_check`*. We may assume that in the tree odd occurrences of c-transitions are resolved as in the first case, even occurrences as in the second case. However, in general, the choice on which c-transition to execute is done arbitrarily: for this reason, in order to show how to resolve the non-determinism, it has been necessary to exploit the notion of execution tree and use the function* σ^i_{ang}.

□

5.2 The Problem of Service Composition

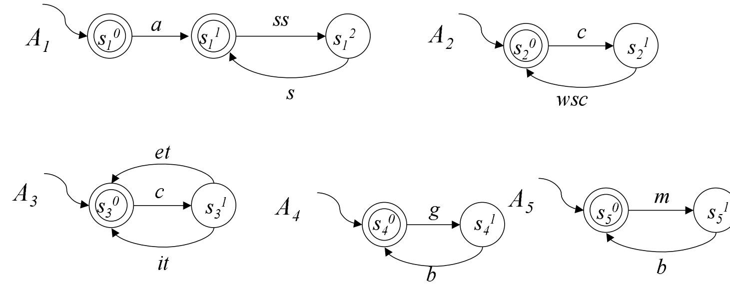
In this new framework, the definition of composition is slightly different from the one reported in Definition 10, since the problem of composition synthesis is concerned with realizing any one of the completely specified target services, which correspond to the underspecified target service provided by the client.

Definition 25 (Composition of Underspecified Target Service) Let C be a well-formed service community and let $\mathcal{E}^{ext}$ be *set* of the external schemas of an underspecified target service E expressed in terms of the alphabet Σ of C. Let $\mathcal{T}(\mathcal{E}^{ext}) = \mathcal{T}^{ext}(E) = \{T^{ext}_i((E))\}$ be the set of external execution trees associated to E. A *composition* of E wrt C is an internal schema E^{int} such that:

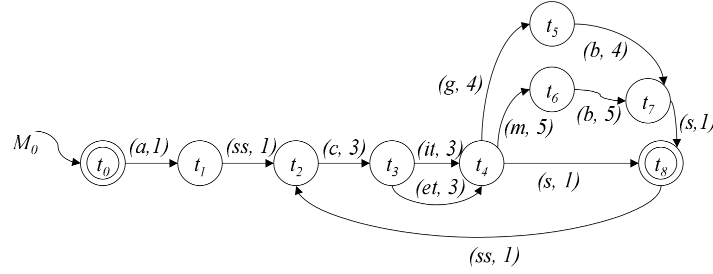
1. $T(E^{int})$ conforms to *any* $T^{ext}_i(E) \in \mathcal{T}^{ext}(E)$,
2. $T(E^{int})$ delegates all actions to the services of C (i.e., `this` does not appear in $T(E^{int})$), and
3. $T(E^{int})$ is coherent with C.

□

Given a community C of services and an underspecified target service E_0, whose underspecified external schema is the FSM $\mathcal{A}^{ext}(E_0)$, the problem of *composition existence* is the problem of checking whether there exists a composition that is coherent with C and that realizes $\mathcal{A}^{ext}(E_0)$. The problem of *composition synthesis* is the problem of synthesizing a composition that is coherent with $\mathcal{C}$ and that realizes $\mathcal{A}^{ext}(E_0)$.



(a) service Community C_{ecard}



a = authentication *wsc = write_&_spell_check*
ss= search_&_select *g = add_gift*
c = customize *m = add_music*
et = use_english_template *b = buy*
it = use_italian_template *s = send*

(b) Composition of $\mathcal{A}_0$ wrt C_{ecard}

Figure 5.3: Composition of the underspecified target service $\mathcal{A}_0$

Since we are considering services that have a finite number of states, we would like also to have a composition that can be represented with a finite number of states, i.e., as a Mealy composition.

Example 21 *Figure 5.3 (a) shows a community of five services* A_1, A_2, A_3, A_4 *and* A_5*.* A_1 *provides actions to* (i) *validate a (registered) client, or to register a new client (*`authentication`*), and then to repeatedly* (ii) `search_&_select` *an* e-card *and* (iii) `send` *it.* A_2 *ad* A_3 *provide functionalities to compose an e-card:* A_2 *repeatedly allows for* `customiz`*ing the layout of a card and for writing a message from scratch and for spell checking it (*`write_&_spell_check`*);* A_3 *repeatedly allows for* `customiz`*ing a card and for selecting a template, which provides help in writing the message: such template can give suggestions in* English *(*`use_english_template`*), or in* Italian *(*`use_italian_template`*). Finally,* A_4 *provides actions to repeatedly add a gift (*`add_gift`*) and pay (*`buy`*)*

it. A_5 provides actions to repeatedly add some music (`add_music`) and pay it (`buy`).

Figure 5.3 (b) shows the Mealy composition M_0 that represents a composition of the underspecified target service $\mathcal{A}_0$ wrt the service community of Figure 5.3 (a). In M_0 the non-determinism of A_0 on the c-transitions is resolved by choosing always s_0^3 as successor state of the c-transitions, i.e., to always having the `customize` action be followed by a choice on the actions `use_english_template` and `use_italian_template`. Therefore, action `write_&_spell_check` is never performed in M_0 and service A_2 is never invoked. Note that M_0 is not the only composition of the underspecified target service $\mathcal{A}_0$. In effect, several ones exists. For example, a first one is identical to M_0, but the non-determinism is resolved by choosing always s_0^4 as successor state of the c-transitions, i.e., by always having the `customize` action be followed by the action `write_&_spell_check`. In this case, the actions `use_english_template` and `use_italian_template` are never executed and service A_3 is never invoked. Other compositions can be obtained by resolving the non-determinism in such a way that "sometimes"[2] s_0^4, "sometimes" s_0^3 are chosen as successor states of the c transition. For example, one such Mealy composition corresponds to a compact representation of the tree of Figure 5.2, described in Example 20, where odd occurrences of c-transitions are resolved by choosing the "upper" c-transition in Figure 5.1, and even occurrences are resolved by choosing the "lower" c-transition □

5.3 Automatic Service Composition Synthesis Technique

In this section, we address the problem of composition existence and synthesis in the FSM-based framework introduced above. The basic idea consists again in reducing the problem of composition existence to satisfiability of a formula written in $DPDL$.

5.3.1 Existence of a Composition

In this section we show how to solve the problem of composition existence in the case when the target service is underspecified. The technique we will discuss is an extension of the technique used in Section 4.3.1. Therefore, in order to avoid useless repetitions, we will only discuss the differences wrt the already presented technique.

Given the underspecified target service E_0, whose (underspecified) external schema is a non-deterministic FSM $\mathcal{A}_0$, and a community of services formed

[2]Provided that the composition is compactly representable.

by n component services $E_1, \ldots, E_n$, whose external schemas are (deterministic) FSM $A_1, \ldots, A_n$ respectively, we build a DPDL formula Φ_{ang} as follows. Analogously to Section 4.3.1, as set of atomic propositions $\mathcal{P}$ in Φ_{ang} we have *(i)* one proposition s_j for each state s_j of $A_j, j = 0, \ldots, n$, denoting whether A_j is in state s_j; *(ii)* propositions $F_j, j = 0, \ldots, n$, denoting whether A_j is in a final state; and *(iii)* propositions $\mathsf{moved}_j, j = 1, \ldots, n$, denoting whether (component) FSM A_j performed a transition. As set of atomic actions $\mathcal{A}$ in Φ_{ang} we have the actions in Σ (i.e, $\mathcal{A} = \Sigma$).

Universal assertions are expressed by the master modality $[u]$. The formula Φ_{ang} is built as a conjunction of the following formulas.

- For the underspecified target service $A_0 = (\Sigma, S_0, s_0^0, \delta_0, F_0)$ we build the formula Φ_{A_0}, as a conjunction of:
  - $[u](s \rightarrow \neg s')$ for all pairs of states $s \in S_0$ and $s' \in S_0$, with $s \neq s'$; these say that propositions representing different states are disjoint.
  - $[u](s \rightarrow \bigvee_{s' \in \delta_0(s,a)} \langle a \rangle \mathtt{true} \wedge [a]s')$ for each a such that $s' = \delta_0(s, a)$; these encode the transitions of A_0. Note the difference wrt the analogous formula of Section 4.3.1. Here, we introduce the operator $\bigvee_{s' \in \delta_0(s,a)}$ to capture all a transitions originating from state s: in the formula in Section 4.3.1 this was not needed, since the FSM denoting the external schema of the target service is deterministic.
  - $[u](s \rightarrow [a]\mathtt{false})$ for each a such that $\delta(s, a)$ is not defined; these say when a transition is not defined.
  - $[u](F_0 \leftrightarrow \bigvee_{s \in F_0} s)$; this highlights final states of A_0.
- For each of the component FSM $A_i = (\Sigma, S_i, s_i^0, \delta_i, F_i)$ we build a formula Φ_{A_i}. Regarding the services in the community, there is no difference between the present framework and the framework in Chapter 4, therefore Φ_{A_i} are built in the same way as in Section 4.3.1.
- Finally, we encode in a formula Φ_{aux} the general structure of the model. In addition to the domain independent conditions of the framework considered in Section 4.3.1, we should have the additional constraint that the obtained composition must be deterministic. However, the DPDL atomic actions are deterministic, therefore, this constraint is implicitly encoded by the DPDL semantics. In other words, also Φ_{aux} is built as in Section 4.3.1.

Note that the DPDL formula Φ built as above presents the same properties as in the previous chapter:

- the Kleene star * is used only in the master modality;

- the master modality has the form $u = (a_1 \cup \ldots \cup a_n)^*$, where $a_1 \ldots a_n$ are DPDL atomic actions;
- the $\langle\rangle$ modality appears only in front of the atomic proposition `true`;
- both modalities $[]$ (when it is not used as master modality) and $\langle\rangle$ are used in expressions of the form $[a]\Psi$ or $\langle a\rangle\Psi$, where Ψ is a DPDL formula and a is an atomic DPDL action.

Again, we will make use of such observations in Chapter 7.

Example 22 *In what follows we discuss the DPDL encoding of the composition problem presented in the running example (underspecified target service of Figure 5.1 and the service community of Figure 5.3(a)).*

The set $\mathcal{P}$ of atomic propositions is:

$$\begin{array}{rcl} \mathcal{P} & = & \{s_0^0, s_0^1, s_0^2, s_0^3, s_0^4, s_0^5, s_0^6, s_0^7, s_0^8, s_1^0, s_1^1, s_1^2, s_2^0, s_2^1, s_3^0, s_3^1, s_4^0, s_4^1, s_5^0, s_5^1, \\ & & F_0, F_1, F_2, F_3, F_4, F_5, \mathsf{moved}_1, \mathsf{moved}_2, \mathsf{moved}_3, \mathsf{moved}_4, \mathsf{moved}_5\} \end{array}$$

where:

- s_j^i, *for* $j = 0, \ldots, 5$ *and* $i = 0, \ldots, 8$*:* s_j^i *is true if and only if* A_j *is in state* s_j^i
- $F_j, j = 0, \ldots, 5$*:* F_j *is true if and only if* A_j *is in a final state;*
- $\mathsf{moved}_j, j = 1, \ldots, 5$*:* moved_j *is true if and only if (component) FSM* A_j *performed a transition*

The set $\mathcal{A}$ of deterministic atomic actions, which by construction coincides with the alphabet Σ of the community, is defined as:

$$\mathcal{A} = \{a, ss, c, et, it, wsc, g, m, b, s\}$$

where:

- c *denotes action* `customize`
- g *denotes action* `add_gift`
- m *denotes action* `add_music`
- b *denotes action* `buy`
- s *denotes action* `send`
- a *denotes action* `authentication`

- *ss denotes action* `search_&_select`
- *et denotes action* `use_english_template`
- *it denotes action* `use_italian_template`
- *wsc denotes action* `write_&_spell_check`

The master modality u is equal to $(a \cup ss \cup c \cup et \cup it \cup wsc \cup g \cup m \cup b \cup s)^*$. We briefly recall that the master modality is used in the formula $[u]\phi$ to assert that the proposition ϕ holds after any regular expression involving the actions in $\mathcal{A}$.

The DPDL formula Φ_{ang} is built as a conjunction of the following formulas.

The **underspecified target service** *$\mathcal{A}_0$ is encoded by the following formulas, with the indicated meaning.*

A_0 cannot be simultaneously in two (or more) states:

$$\begin{array}{llll}
[u](s_0^0 \rightarrow \neg s_0^1) & [u](s_0^1 \rightarrow \neg s_0^2) & [u](s_0^2 \rightarrow \neg s_0^4) & [u](s_0^4 \rightarrow \neg s_0^7) \\
[u](s_0^0 \rightarrow \neg s_0^2) & [u](s_0^1 \rightarrow \neg s_0^3) & [u](s_0^2 \rightarrow \neg s_0^5) & [u](s_0^4 \rightarrow \neg s_0^8) \\
[u](s_0^0 \rightarrow \neg s_0^3) & [u](s_0^1 \rightarrow \neg s_0^4) & [u](s_0^2 \rightarrow \neg s_0^6) & [u](s_0^5 \rightarrow \neg s_0^6) \\
[u](s_0^0 \rightarrow \neg s_0^4) & [u](s_0^1 \rightarrow \neg s_0^5) & \ldots & [u](s_0^5 \rightarrow \neg s_0^7) \\
[u](s_0^0 \rightarrow \neg s_0^5) & [u](s_0^1 \rightarrow \neg s_0^6) & \ldots & [u](s_0^5 \rightarrow \neg s_0^8) \\
[u](s_0^0 \rightarrow \neg s_0^6) & [u](s_0^1 \rightarrow \neg s_0^7) & [u](s_0^3 \rightarrow \neg s_0^8) & [u](s_0^6 \rightarrow \neg s_0^7) \\
[u](s_0^0 \rightarrow \neg s_0^7) & [u](s_0^1 \rightarrow \neg s_0^8) & [u](s_0^4 \rightarrow \neg s_0^5) & [u](s_0^6 \rightarrow \neg s_0^8) \\
[u](s_0^0 \rightarrow \neg s_0^8) & [u](s_0^2 \rightarrow \neg s_0^3) & [u](s_0^4 \rightarrow \neg s_0^6) & [u](s_0^7 \rightarrow \neg s_0^8)
\end{array}$$

A_0 can perform are the following transitions:

$$\begin{array}{l}
[u](s_0^0 \rightarrow \langle a\rangle \texttt{true} \wedge [a]s_0^1) \\
[u](s_0^1 \rightarrow \langle ss\rangle \texttt{true} \wedge [ss]s_0^2) \\
[u](s_0^2 \rightarrow (\langle c\rangle \texttt{true} \wedge [c]s_0^3) \vee (\langle c\rangle \texttt{true} \wedge [c]s_0^4)) \\
[u](s_0^3 \rightarrow \langle it\rangle \texttt{true} \wedge [it]s_0^5) \\
[u](s_0^3 \rightarrow \langle et\rangle \texttt{true} \wedge [et]s_0^5) \\
[u](s_0^4 \rightarrow \langle wsc\rangle \texttt{true} \wedge [wsc]s_0^5)
\end{array}
\qquad
\begin{array}{l}
[u](s_0^5 \rightarrow \langle m\rangle \texttt{true} \wedge [m]s_0^6) \\
[u](s_0^5 \rightarrow \langle g\rangle \texttt{true} \wedge [g]s_0^6) \\
[u](s_0^5 \rightarrow \langle s\rangle \texttt{true} \wedge [s]s_0^8) \\
[u](s_0^6 \rightarrow \langle s\rangle \texttt{true} \wedge [s]s_0^8) \\
[u](s_0^8 \rightarrow \langle ss\rangle \texttt{true} \wedge [ss]s_0^2)
\end{array}$$

Note the third formula in the first column, that represents the non-determinism by or*-ing the terms encoding the two possible transitions. This formula states that for all possible sequences of actions, if $\mathcal{A}_0$ is in state s_0^2, then the FSM allows* either *for customizing the* e-card *(i.e., it can execute action c) and moving to state s_0^3,* or *for executing action c and moving to state s_0^4. Observe the difference wrt encoding a choice on a set of actions, i.e., when several different actions can be executed from a state. In such case, the terms encoding the possible transitions are in* and*, as the three formulas on*

top of the second column[3].

The transitions that A_0 cannot perform are (partially) defined as follows:

$[u](s_0^0 \to [ss]\texttt{false})$ $[u](s_0^1 \to [a]\texttt{false})$ $\ldots$
$[u](s_0^0 \to [c]\texttt{false})$ $[u](s_0^1 \to [c]\texttt{false})$ $[u](s_0^5 \to [a]\texttt{false})$
$[u](s_0^0 \to [it]\texttt{false})$ $[u](s_0^1 \to [it]\texttt{false})$ $[u](s_0^5 \to [ss]\texttt{false})$
$[u](s_0^0 \to [et]\texttt{false})$ $[u](s_0^1 \to [et]\texttt{false})$ $[u](s_0^5 \to [c]\texttt{false})$
$[u](s_0^0 \to [wsc]\texttt{false})$ $[u](s_0^1 \to [wsc]\texttt{false})$ $[u](s_0^5 \to [wsc]\texttt{false})$
$[u](s_0^0 \to [m]\texttt{false})$ $[u](s_0^1 \to [m]\texttt{false})$ $[u](s_0^5 \to [it]\texttt{false})$
$[u](s_0^0 \to [g]\texttt{false})$ $[u](s_0^1 \to [g]\texttt{false})$ $[u](s_0^5 \to [et]\texttt{false})$
$[u](s_0^0 \to [b]\texttt{false})$ $[u](s_0^1 \to [b]\texttt{false})$ $[u](s_0^5 \to [b]\texttt{false})$
$[u](s_0^0 \to [s]\texttt{false})$ $[u](s_0^1 \to [s]\texttt{false})$ $\ldots$

Finally, the final states of $\mathcal{A}_0$ are captured by:

$$[u](F_0 \leftrightarrow (s_0^0 \vee s_0^8))$$

As far as the **services in the community**, *we refer the reader to Section 4.3.1 for a detailed discussion about their encoding in DPDL. In what follows we show the encoding of service A_1 of Figure 5.3(a), which is slightly more complicated than the services in community described in Example 14 of Section 4.3.1.*

The disjointness of states is captured by:

$$[u](s_1^0 \to \neg s_1^1) \qquad [u](s_1^0 \to \neg s_1^2) \qquad [u](s_1^1 \to \neg s_1^2)$$

The transitions of A_1, conditioned to the fact that A_1 is actually required to make the transition in the composition, are:

$$[u](s_1^0 \to [a](\mathsf{moved}_1 \wedge s_1^1 \vee \neg\mathsf{moved}_1 \wedge s_1^0))$$
$$[u](s_1^1 \to [ss](\mathsf{moved}_1 \wedge s_1^2 \vee \neg\mathsf{moved}_1 \wedge s_1^1))$$
$$[u](s_1^2 \to [s](\mathsf{moved}_1 \wedge s_1^1 \vee \neg\mathsf{moved}_1 \wedge s_1^2))$$

The following formulas encode (part of) the transitions that are not defined on A_1 (and that do not make A_1 change state):

$[u](s_1^0 \to [ss](\neg\mathsf{moved}_1 \wedge s_1^0))$ $[u](s_1^1 \to [c](\neg\mathsf{moved}_1 \wedge s_1^1))$
$[u](s_1^0 \to [s](\neg\mathsf{moved}_1 \wedge s_1^0))$ $\ldots$
$[u](s_1^0 \to [c](\neg\mathsf{moved}_1 \wedge s_1^0))$ $[u](s_1^2 \to [a](\neg\mathsf{moved}_1 \wedge s_1^0))$
$\ldots$ $[u](s_1^2 \to [ss](\neg\mathsf{moved}_1 \wedge s_1^0))$
$[u](s_1^1 \to [a](\neg\mathsf{moved}_1 \wedge s_1^1))$ $[u](s_1^2 \to [c](\neg\mathsf{moved}_1 \wedge s_1^0))$
$[u](s_1^1 \to [s](\neg\mathsf{moved}_1 \wedge s_1^1))$ $\ldots$

[3] We remind that by a factorization on the left, such three formulas are equivalent to $[u](s_0^5 \to (\langle m\rangle\texttt{true} \wedge [m]s_0^6) \wedge (\langle g\rangle\texttt{true} \wedge [g]s_0^6) \wedge (\langle s\rangle\texttt{true} \wedge [s]s_0^8))$.

Final states of A_1 *are captured by:* $[u](F_1 \leftrightarrow (s_1^0 \vee s_1^1))$

Finally, **the domain independent conditions** *are captured by the following formulas.*

Initially all services are in their initial state: $s_0^0 \wedge s_1^0 \wedge s_2^0 \wedge s_3^0 \wedge s_4^0 \wedge s_5^0$

At each step at least one of the component FSM has moved:

$$[u](\langle a\rangle \texttt{true} \rightarrow [a](\mathsf{moved}_1 \vee \mathsf{moved}_2 \vee \mathsf{moved}_3 \vee \mathsf{moved}_4 \vee \mathsf{moved}_5))$$
$$[u](\langle ss\rangle \texttt{true} \rightarrow [ss](\mathsf{moved}_1 \vee \mathsf{moved}_2 \vee \mathsf{moved}_3 \vee \mathsf{moved}_4 \vee \mathsf{moved}_5))$$
$$[u](\langle c\rangle \texttt{true} \rightarrow [c](\mathsf{moved}_1 \vee \mathsf{moved}_2 \vee \mathsf{moved}_3 \vee \mathsf{moved}_4 \vee \mathsf{moved}_5))$$
$$[u](\langle it\rangle \texttt{true} \rightarrow [it](\mathsf{moved}_1 \vee \mathsf{moved}_2 \vee \mathsf{moved}_3 \vee \mathsf{moved}_4 \vee \mathsf{moved}_5))$$
$$[u](\langle et\rangle \texttt{true} \rightarrow [et](\mathsf{moved}_1 \vee \mathsf{moved}_2 \vee \mathsf{moved}_3 \vee \mathsf{moved}_4 \vee \mathsf{moved}_5))$$
$$[u](\langle wsc\rangle \texttt{true} \rightarrow [wsc](\mathsf{moved}_1 \vee \mathsf{moved}_2 \vee \mathsf{moved}_3 \vee \mathsf{moved}_4 \vee \mathsf{moved}_5))$$
$$[u](\langle m\rangle \texttt{true} \rightarrow [m](\mathsf{moved}_1 \vee \mathsf{moved}_2 \vee \mathsf{moved}_3 \vee \mathsf{moved}_4 \vee \mathsf{moved}_5))$$
$$[u](\langle g\rangle \texttt{true} \rightarrow [g](\mathsf{moved}_1 \vee \mathsf{moved}_2 \vee \mathsf{moved}_3 \vee \mathsf{moved}_4 \vee \mathsf{moved}_5))$$
$$[u](\langle b\rangle \texttt{true} \rightarrow [b](\mathsf{moved}_1 \vee \mathsf{moved}_2 \vee \mathsf{moved}_3 \vee \mathsf{moved}_4 \vee \mathsf{moved}_5))$$
$$[u](\langle s\rangle \texttt{true} \rightarrow [s](\mathsf{moved}_1 \vee \mathsf{moved}_2 \vee \mathsf{moved}_3 \vee \mathsf{moved}_4 \vee \mathsf{moved}_5))$$

When the target service is in a final state also all component services must be in a final state:

$$[u](F_0 \rightarrow F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge F_5)$$

□

The following Lemma 26 and 27 hold.

Lemma 26 *If there exists a composition of an underspecified target service* E_0 *wrt* $E_1, \ldots, E_n$*, then the DPDL formula* Φ_{ang}*, constructed as above, is satisfiable.* □

Lemma 27 *Any model of the DPDL formula* Φ_{ang}*, constructed as above, denotes a composition of* E_0 *wrt* $E_1, \ldots, E_n$*.* □

Their proof are analogous to the proof of Lemma 16 and 17, considering that *(i)* by Definitions 24 and 25 an underspecified target service denotes a set of external schemas and the composition is an internal schema which realizes anyone of such external schemas and is coherent with the community, and that *(ii)* each function σ^i_{ang} defined in the previous section provides a mapping from states of a non-deterministic FSM representing an underspecified target service to the nodes of an external execution tree.

Therefore, by Lemma 26 and 27 the following theorem holds.

Theorem 28 *The DPDL formula Φ_{ang}, constructed as above, is satisfiable if and only if there exists a composition of E_0 wrt $E_1, \ldots, E_n$.* □

The size of Φ_{ang} is polynomially related to A_0 and $A_1, \ldots, A_n$. Hence, from the EXPTIME-completeness of satisfiability in DPDL and from Theorem 28 we get the following complexity result.

Theorem 29 *Checking the existence of service composition can be done in EXPTIME.* □

5.3.2 Synthesizing a Composition

In the previous section we have shown that we are able to check the existence of a composition by checking satisfiability of a DPDL formula Φ_{ang} encoding the underspecified target service, the services in the community and a number of domain independent conditions. In this section we show how to synthesize a composition which is a FSM. This algorithm is an extension of the algorithm presented in Section 4.3.2, that returns a composition, if one exists, and returns a special symbol (**nil**), denoting that no composition exists, otherwise. All the observations and results discussed in Section 4.3.2 continue to hold. In particular, this is true for the synthesis of a composition which is a finite state machine, and in particular, of a Mealy composition, which is obtained starting from a model which is deterministic by construction. Therefore, also the following theorems hold, whose proofs can be easily obtained from the proofs of analogous Theorems 21 and 22, respectively.

Theorem 30 *If there exists a composition of E_0 wrt $E_1, \ldots, E_0$, then there exists a Mealy composition whose size is at most exponential in the size of the external schemas A_0, $A_1, \ldots, A_n$ of E_0, $E_1, \ldots, E_n$ respectively.*

Theorem 31 *Any finite model of the DPDL formula Φ_{ang} denotes a Mealy composition of E_0 wrt $E_1, \ldots, E_n$.*

Exploiting reasoning methods for DPDL based on model construction, such as tableaux algorithms [39, 52, 7], one can actually construct such a MFSM composition. Also the algorithm for synthesizing a Mealy composition shown in Figure 4.6 can be applied to this new framework. Note that the following functions continue to work with no change in the implementation: *(i)* `DPDLTableau`, for checking satisfiability of the DPDL formula, and building a finite model if it exists or returning (**nil**), otherwise; *(ii)* `Extract_MFSM`, for extracting a Mealy composition; *(iii)* `Minimize` for minimizing the MFSM. The only function that needs to be (slightly) tailored towards the new framework is `FSM2DPDL`, since as discussed in Section 5.3.1, the DPDL encoding must cope with the presence of non-determinism in the underspecified target service.

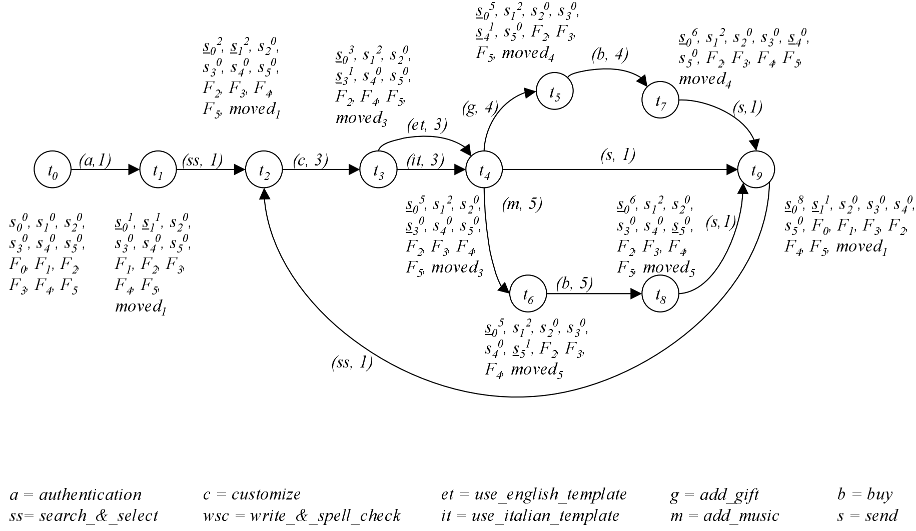


Figure 5.4: Finite model $\mathcal{I}_f$ for Φ_{ang}.

Example 23 *Figure 5.4 shows a finite model (i.e., a Kripke structure) $\mathcal{I}_f$ of the formula Φ_{ang} encoding our running example, returned from a DPDL tableau algorithm. For sake of clarity, for each state we report only a list of the atomic propositions that are true in that state and we underline the propositions denoting states that change their value when an action is performed. It is easy to see that $\mathcal{I}_f$ is indeed a composition for the underspecified target service A_0 of Example 19. $\mathcal{I}_f$ is not minimal: by applying standard minimization techniques one obtains the finite state machine of Figure 5.3 (b). A detailed analysis of $\mathcal{I}_f$ allows us to make the following observations:*

- *Action* `write_&_spell_check`*, performed by service A_2 is never executed, therefore A_2 never moves and it always remain in its initial state s_2^0, which is also final.*

- *Action* `buy` *can be performed both by A_4 and by A_5: each execution of* `buy` *ends in a different state (t_7 and t_8 in the figure), characterized by a different value of the propositions $moved_4$ and $moved_5$: in t_7 $moved_4 =$* `true` *and $moved_5 =$* `false` *because* `buy` *is performed by A_4, while in t_8 $moved_5 =$* `false` *and $moved_5 =$* `true` *because* `buy` *is performed by A_5. Therefore, the minimal FSM shown in Figure 5.3 (b) is not a model of Φ_{ang}, because there the two* `buy` *transitions end in the same state.*

- *State t_1 is not final, despite the fact that all component services are in a final state, because the target service is not in a final state. The final*

states are t_0 and t_9.

□

5.4 Discussion

In this chapter we have studied the problem of service composition in the situation when the client request is underspecified. Here, we follow the approach presented in Chapter 4. In particular,

1. The composition consists again in coordinating the component services in an interleaved and concurrent way, in order to achieve the client request.

2. The client request specifies a target service, i.e., the transition system that the client desires to execute, but presents an interesting enhancement. In Chapters 3 and 4 we assumed that the client request is completely specified: it represents a single target service, while here we consider the case where a client request represents a *a set of* target services. The underspecification that we have studied shows up in the form of "don't care" nondeterminism (angelic nondeterminism) on the next set of transitions of the target service available to the client: the client allows the composition synthesis to resolve nondeterministic choices taking advantage of what the available component services can do at that point of their computation[4].

In this enhanced framework, we have studied the problem of service composition. Our main results is a composition synthesis technique, which is sound, complete and terminating, i.e., if and only if a composition of the available component service realizing the client specification exists, our technique will produce one such a composition. The composition produced is finite state, and hence, as a collateral result of our synthesis technique, we show that if a composition exists then there exists one which is indeed finite state.

Note that to the best of our knowledge, the framework presented in this chapter is unique in the literature: we could deal with "don't care" nondeterminism because the client request specifies a (desired) transition system.

Finally, a brief note on the used terminology. In general, adding nondeterminism to a specification leads to a partial specification which corresponds in fact to a set of complete specifications. The non-determinism introduced in the target service is called *angelic* because it requires to realize

[4]This has to be contrasted with the fact that at the same time the composition synthesis must generate a composition that allows the client to make all choices specified in its transition system.

one (any one) of the completely specified target services corresponding to the target services which the client partially specifies. The term “angelic” is used in contrast to the term “diabolic” which denotes the situation when non-determinism is added to the services in the community. The latter case consists in having a partial specification of the exported behavior of services in the community. In principle, this corresponds to have a huge (possibly infinite) number of complete specifications for each service in the community. Of course, the composite service must be built in such a way to be coherent with *all* possible complete specification corresponding to the partial specification of the services in the community. Therefore, diabolic non-determiminism is far more difficult to handle: it is out of the scope of this thesis and it is left for future research.

Chapter 6

Allowing Services to Take the Initiative

The exported behavior of services, as defined in previous chapters, do not mention the role a service has wrt a given action, i.e., whether the service is "passively" delegated an action or it "takes the initiative" and delegates it: indeed, the external schema only refers to actions being delegated to a services, and that the service (no matters how) can execute. In this chapter, we extend our framework by encoding such information in the exported behavioral description of a service, and by therefore allowing service in the community to synchronize and communicate one with the other. This novelty can be easily applied also to the frameworks of Chapters 4 and 5.

In Chapter 5 we also studied how to automatically synthesize a composition when the client request is underspecified and presents some forms of don't care non-determinism: it represents a set of target services and the composition synthesis is asked to realize any target service in such a set. Here, we focus again on a loose client specification, in which the incompleteness shows up not only as don't care non-determinism, but also as "empty spots" in the specification, denoted by τ actions: they represent specific points in the execution of the composite service in which the client does not want to be directly involved, but of which he wants to be aware.

We study again automatic composition synthesis in such a framework: we devise a sound, complete and terminating algorithm, based on satisfiability in a variant of Propositional Dynamic Logic that solves the composition problem, and we analyze its computational complexity.

6.1 Servant and Initiator Services

In order to perform a given task, a service executes certain actions in coordination with its client or other services. Specifically, each action in the task has a (single) *initiator*, typically the client, which requests the execution of the action possibly passing along information, and one or more *servants*, which are services that respond to the request, possibly exchanging with the initiator further information.

In order to represent the role a service has wrt a given action, we annotate each action symbol as follows: if the service is one of the servants of an action a, then the action appears as ${}^{\gg}a$, conversely if the service is the initiator of a, then the action appears as $a^{\gg}$. Of course, such annotation on the actions is reflected on the internal and external execution trees of each service: Definitions 2 and 3 continue to hold, as well as the notions introduced in Section 3.3. However, observe that for each node of an execution tree, we can have at most one successor node *for each annotated action*, i.e., from a node two a-labeled edges can originate, having different annotations: we assume that the state of the service is entirely determinated by the sequence of annotated actions executed so far, i.e. up to the node associated with the current state of the computation. The alphabet of the community contains non-annotated actions.

Also in this chapter we concentrate on services whose external and internal schemas can be represented using a *finite number of states*, specifically by deterministic finite state machines (FSM), or equivalently, finite deterministic transition systems. Note that the definitions of external and internal schema of a service, represented as FSM can be easily obtained from Definitions 13 and 14, therefore, here we report only the definition of FSM external schema.

Definition 32 ((Annotated) FSM External Schema) Let C be a service community, whose alphabet of actions is Σ. Let E_i be a service in C. The external schema of E_i is a FSM $A_i = (\Sigma^+, S_i, s_i^0, \delta_i, F_i)$, where:

- $\Sigma^+ = \{{}^{\gg}a, a^{\gg} \mid a \in \Sigma\}$ is the alphabet of the FSM;
- S_i is the set of states, representing the finite set of states of the service;
- s_i^0 is the initial state, representing the initial state of the service;
- $\delta_E : S_i \times \Sigma^+ \rightarrow S_i$ is the (partial) transition function that given a state s and an annotated action ${}^{\gg}a$ (or $a^{\gg}$) returns the state resulting from executing the action in s;
- $F_i \subseteq S_i$ is the set of final states, i.e., the states where the service can terminate.

□

The FSM A_i compactly represents the execution tree $T(A_i)$ of the corresponding service: this can be shown by defining a function σ_{tau} that associates to each node of the execution tree a state in the FSM. The definition of σ_{tau} is done inductively on the level of nodes in the tree, and can be easily obtained from the analogous function defined in Section 4.1 (or Section 5.1). Intuitively, given A_i, the execution tree $T(A_i)$ *generated* by A_i is the execution tree containing one node for each sequence of actions obtained by following (in any possible way) the transitions of A_i, and annotating as final those nodes corresponding to the traversal of final states.

The different roles that a service can play with respect to a given action (i.e., either as initiator or as servant) induce a classification between the services of a community:

- *pure-servant* service: it is a service that acts only as servant in all possible sequences of interactions it can be involved in; its associated FSM external schema presents only actions of type $^{\gg}a$; such a service can be directly exploited by a client, as it is able to completely satisfy client requests and execute its tasks (this is the kind of services studied in Chapter 4);

- *pure-initiator* service: it is a service that acts only as initiator in all possible sequences of interactions it can be involved in; its associated FSM external schema presents only actions of type $a^{\gg}$; this kind of service is, for example, the one typically associated to the client, i.e., it represents the target service that the client wants to actually execute.

- *mixed* service: it is a service that acts as initiator in at least one possible sequence of interactions it can be involved in; its associated FSM external schema[1] presents at least one action of type $a^{\gg}$; such a service can be exploited in a composition only if a matching service that acts as servant can be found in the community, to which it can delegate the execution of actions it initiates.

Observe that the fact that a client request is a specification of a pure-initiator service correspond to the idea, almost universally accepted in the Service Oriented Computing literature, that the client cannot be exploited by the services in the community for carrying on their tasks.

[1]In what follows, we omit the terms “external schema” and “internal schema” when clear from the context.

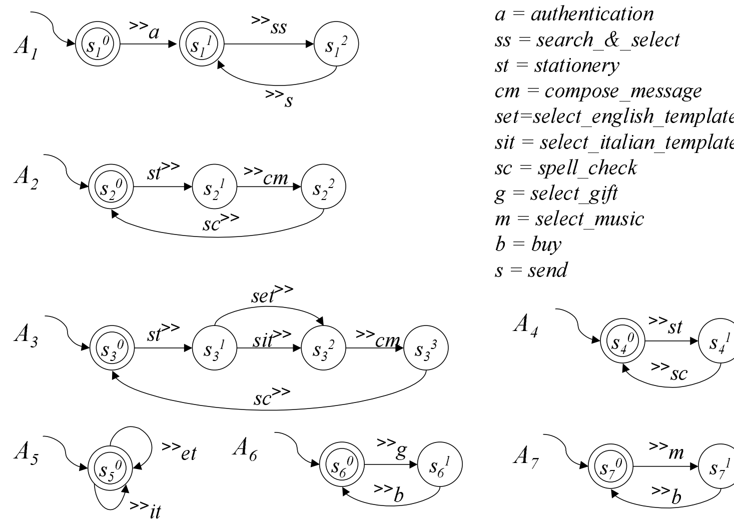


Figure 6.1: services of the community

Example 24 *Consider the* e-card *scenario of Examples 19 and 21. It can be applied to this new framework by:* (ii) *annotating with* $^{\gg}(\cdot)$ *the actions offered by services of the community (shown in Figure 5.3 (a)); and* (ii) *annotating with* $(\cdot)^{\gg}$ *the actions requested by the client in the underspecified target service (shown in Figure 5.1). The services in the community are, therefore, pure-servants and the target service is pure-initiator.* □

Example 25 *Consider again an* e-card *scenario, where the community of services is the one shown in Figure 6.1. It is similar but not equal to the community of Example 21, since we want to show the differences wrt the previous framework and the novel features that this new framework presents. It is constituted by services* $A_1, A_2, A_3, A_4, A_5, A_6$ *and* A_7*.* A_1 *provides the same functionalities as* A_1 *of Example 21: note that* A_1 *is a pure-servant.* A_2 *provides functionalities to compose an* e-card*: it repeatedly allows for selecting the stationery of the message (action* `stationery`*), composing the message (action* `compose_message`*), and performing a spell check (action* `spell_check`*).* A_2 *is servant for the* `compose_message` *action and it is initiator for all the remaining actions.* A_3 *is similar to* A_2*, since it offers the same operations in the same order: in addition, it provides help for composing the message, since between actions* `stationery` *and* `compose_message`*, it allows for selecting either an English template (action* `select_english_template`*), or an Italian (action* `select_italian_template`*), which give suggestions in the corresponding language.* A_3 *acts as servant for the* `compose_message` *action and it is initiator for all the other actions.* A_2 *and* A_3 *are therefore mixed services: they can*

be used in a composition only if the community contains services that act as servants for the actions A_2 and A_3 initiate. A_4 iteratively allows for selecting the `stationery` *and for* `spell_checking` *the text of the* e-card*. A_5 allows for executing actions* `select_english_template` *and* `select_italian_template` *any number of times in any order and it acts as a servant for both of them. Finally, services A_6 and A_7 allow to add respectively, a gift or some music to the* e-card*, and to pay it (note that they offer the same functionalities as services A_4 and A_5 of Example 21, respectively). A_4, A_5, A_6 and A_7 are pure-servant.*
□

6.2 Client Specification

Before turning to the problem of composition, we address the client request. In this thesis, we consider the client request, as the specification of the transition system that the client is interested in being able to execute. In particular, in the framework studied in this chapter, it represents the pure-initiator service that the client wants to realize, possibly presenting advanced forms of underspecification. In Chapter 5 we already studied underspecification in the form of non-determinism on the transitions. In this chapter we enrich the framework with another kind of underspecification. In specified points of the pure-initiator service, specified by the client, we allow that the service itself is interleaved with activities performed by other services in the community in which the client is not involved at all (but of which the client is aware). The client does not explicitly specify such activities, but he only indicates when they are allowed, by introducing in his specification the so-called τ actions. When the client includes a τ action in his request, he is informing the composition system that he is not interested in which sequence of actions is performed. This has two notably consequences: *(i)* the composition system is free to realize each τ action with whatever sequence of actions provided by services in the community, and *(ii)* the client does not want to participate (neither as initiator nor as a servant) in such execution, but still wants to be informed of what is going on. Note that, however, the clients wants to know only which action(s) are executed, not its initiator, nor its servant(s).

In practical situations, when a client is using an already deployed and running service, it is quite common the presence of actions he does not participate in, but that are executed and of which he sees the outcome. An example is again the Orbitz service, described in Example 7 and shown in Figure 4.1. Consider the functionality to find a flight, the client explicitly looks for a flight, but he does not make any request about the hotel or other services, and still the service allows him to select among a list of hotel and of services, depending on the parameters specified by the client when searching for the

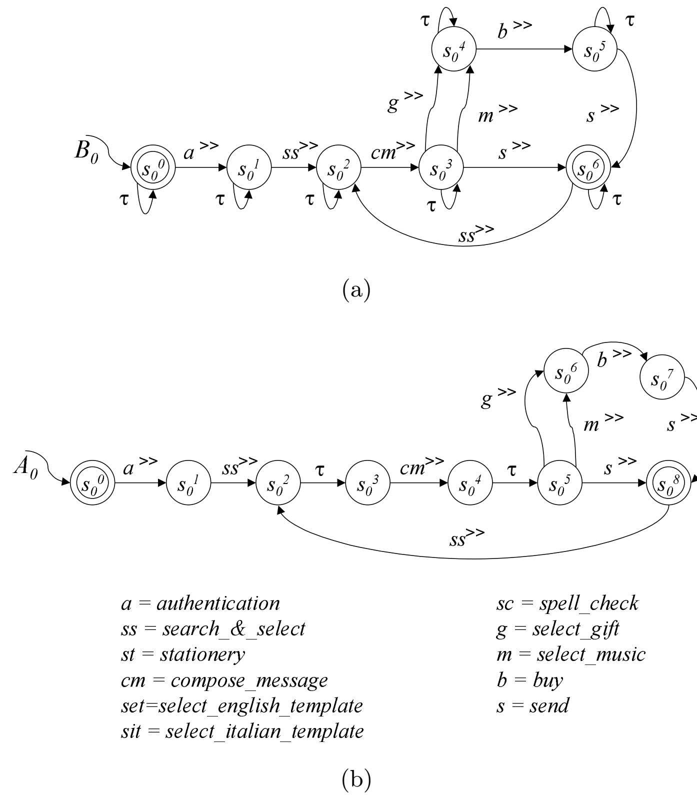


Figure 6.2: τ actions in the client specification

flight. This means that the actions to search for hotel and for services are performed by the services, without involving the client. Another example is provided by services for paying on-line with the credit card: the client does not participate in the actions performed for charging his bank account, but he wants to be aware that his account has been charged with the right amount.

There are two main modalities in which τ actions can be introduced in a client specification. This is shown in the following examples. Since we are in a finite setting, the client specification is expressed as a finite state machine.

Example 26 *Figure 6.2 (a) shows a client specification $\mathcal{B}_0$ that allows for the following interactions. After being authenticated (`authentication` action), the client wants to search for an* e-card *and select it (`search_&_select` action). Then, he desires to compose the message (`compose_message` action). Next, he would like to choose whether* (i) *`send`ing the* e-card, *or* (ii) *selecting and attaching a gift (`add_gift` action) or* (iii) *some music (`add_music` action), paying it (`buy`) and `send`ing the* e-card. *Finally, the client chooses whether to stop executing the services or send another* e-card *by performing the action*

`search_&_select`. *Note that each state of* $\mathcal{B}_0$ *presents a* τ*-labeled loop: this means that the client specification allows the composition system to execute, after each action for which the client is initiator, any sequence of actions, possibly of length zero, for which the client is a pure observator.*

As a slightly further step, one may consider that the client presents to the composition system a request that does not include τ*-actions. Thus, the system may automatically add a* τ*-loop to each state and use the resulting specification to compute a composition.*

Figure 6.2 (b) shows a client specification $\mathcal{A}_0$ *that allows for the same interactions, but where the* τ *actions are explicitly represented: in particular, they do not participate in loops, but they are defined between the pairs of states* (s_0^2, s_0^3) *and* (s_0^4, s_0^5)*. Since all states* s_0^i *of* $\mathcal{A}_0$ *are different, each* τ *action must be realized by* at least *one action, that allows the transition between each pair of states to take place.*

In what follows we consider only the latter situation, since it is more general than the one described at the beginning of the example. □

Once that we have given the intuition of a client specification enriched with τ actions, we are able to give the following formal definition.

Definition 33 (Client Specification) Let C be a service community whose alphabet of actions is Σ. We define a client specification as a nondeterministic FSM $\mathcal{A}_0 = (\Sigma_0, S_0, s_0^0, \delta_0, F_0)$, where:

- $\Sigma_0 = \{a^{\gg} \mid a \in \Sigma\} \cup \{\tau\}$ where τ is a special action that represents a finite sequence of actions in which the client is not the initiator (nor a servant);
- S_0 is the set of states;
- s_0^0 is the initial state;
- $\delta_0 : S_0 \times \Sigma_0 \to 2^{S_0}$ is the (partial) transition function that given a state and an action returns the set of possible successor states;
- $F_0 \subseteq S_0$ is the set of final states.

□

Note that the the client specification may contain don't care non-determinism as introduced in the previous chapter. The non-determinism can involve also τ actions: in this case, when two τ transitions originate from the same state, the client is leaving the composition system free to choose any one transition and replace it with whatever sequence of actions.

Finally, note an important difference wrt the client specification described in the previous chapters. A client specification still represents the service the client wants to exploit. However, by introducing the τ actions, the client specification does *not* denote the external schema of a target service, neither a set of external schemas (of a target service). Indeed, the τ actions represents points in the execution where the client is not directly involved, while an external schema represents the behavioral description of a service from the client point of view, and therefore, it represents exactly (inter)actions between a client and the services that the client *chooses* for execution. Consequently, the notion of composition reported in Definition 10 is not valid in this new framework, in particular because, due to the τ actions, a composition does not (actually, cannot) conform to the client specification, according to Definition 10. Therefore, in the next section we discuss a new notion of composition that copes with the new elements introduced so far, namely *(i)* the role of the client as initiator, *(ii)* the role of services as servant or as initiator, and *(iii)* the underspecification in the client request.

6.3 The problem of Service Composition

In this section, we study the problem of service composition in the new framework, i.e., how to suitably orchestrate (i.e., coordinate the execution of) both the initiator and the servants of each action, using the services in the community, in order to realize the client request.

Definition 34 (Composition Tree) Formally, let $\mathcal{C}$ be a service community whose alphabet of actions is Σ. Let $\mathcal{C}$ be formed by n services $A_1, \ldots, A_n$, and let the client specification be denoted by $\mathcal{A}_0$. A *composition tree* $\mathcal{T}_c$ is a labeled tree $\mathcal{T}_c = (\mathcal{T}, \mathit{fin})$ where $\mathcal{T}$ is a tree over $\Sigma \times [0..n]$[2] (0 stands for the client and $1, \ldots, n$ stand for the services $A_1, \ldots, A_n$, respectively) and *fin* is a boolean labeling function, such that:

- The root ε of the tree represents the fact that no action has been executed yet.
- Each node x in the composition tree $\mathcal{T}_c$ represents the history up to now, i.e., the sequence of actions and their initiator as orchestrated so far.
- For every action a belonging to the alphabet Σ of the community and $\iota \in [0..n]$, $\mathcal{T}_c$ contains at most one successor node $x \cdot (a, \iota)$.
- Some nodes of the composition tree are labeled by *fin* as `true`: such nodes are called *final*. When a node is final, and only then, the orchestration can be stopped.

[2] We use $[i..j]$ to denote the set $\{i, \ldots, j\}$.

- Let the pair $(x, x\cdot(a, \iota))$ be an *edge* of the tree. Each edge $(x, x\cdot(a, \iota))$ of $\mathcal{T}_c$ is labeled by a triple (ι, a, S), where a is the orchestrated action, $\iota \in [0..n]$ denotes the initiator, and $S \subseteq [1..n]$ denotes the nonempty set of services in $\mathcal{C}$ that act as servants. As an example, the label $(0, a, \{1, 3\})$ means that the action a is initiated by the client and served by the services A_1 and A_3.

□

Definition 35 (Composition) Let $\mathcal{C}$, A_0 and $\mathcal{T}_c$ be as in Definition 34. A composition O of A_0 wrt the services in $\mathcal{C}$ is the specification of a *composition tree* $\mathcal{T}(O) = \mathcal{T}_c$. □

Note that the composition tree is the internal execution tree of a synthesized composite service (and the composition is its internal schema). Its (offered[3]) external execution tree $\mathcal{T}_c^{ext}$ is the tree obtained from the composition tree $\mathcal{T}_c$ by replacing:

- with $(a, 0)$ each edge of $\mathcal{T}_c$ labeled by $(0, a, S)$, i.e., such that the initiator of a is the client, and
- with a each edge of $\mathcal{T}_c$ labeled by (ι, a, S), with $\iota \neq 0$, i.e., such that the initiator of a is not the client (but any other service in the community)

In other words, $\mathcal{T}_c^{ext}$ is obtained by projecting out all labels denoting the servants and those initiators which are different from the client.

Given a composition tree $\mathcal{T}(O)$ and a path p (i.e., a sequence of edges) in $\mathcal{T}(O)$ starting from the root and arriving to a node x in $\mathcal{T}(O)$, we call the *projection* of p on a service A_i the sequence of (annotated) actions obtained from p as follows:

1. we remove from p all edges whose label (ι, a, S) is such that $i \notin \{\iota\} \cup S$
2. in the resulting sequence, we replace
   (a) by $a^{\gg}$, each edge labeled by (ι, a, S) where $\iota = i$;
   (b) by ${}^{\gg}a$, each edge labeled by (ι, a, S) where $i \in S$.

Intuitively, point 1 above throws away all edges where service A_i is neither servant nor initiator for action a; points 2(a) and 2(b) deal with edges where A_i is, respectively, the initiator or a servant for action a.

We say that a composition O is *coherent* with a community $\mathcal{C}$ if its tree $\mathcal{T}(O)$ has the following properties:

[3] See Section 3.3.3 for the definition of offered external schema.

- for each edge labeled with (ι, a, S), the action a is in the alphabet of $\mathcal{C}$, and for each service A_i in $\{\iota\} \cup S$, A_i is a member of $\mathcal{C}$;
- for each path p in $\mathcal{T}(O)$ from the root of $\mathcal{T}(O)$ to a node x, and for each service A_i appearing in p, the projection of p on A_i is a (sequence of annotated actions represented by) a node y in the execution tree $\mathcal{T}(A_i)$ of A_i, and moreover, if x is final in $\mathcal{T}(O)$, then y is final in $\mathcal{T}(A_i)$.

Note that from Definition 33, a client specification characterizes some aspects of the composition tree that the client would like to have realized using the services in the community. Of the composition tree, the client specifies *(i)* the actions for which he is the initiator, and *(ii)* the possibility of having activities in which the client himself is not involved.

Observe that the nondeterministic FSM $\mathcal{A}_0$ of Definition 33 specifies a *set* $\mathcal{T}(\mathcal{A}_0)$ of composition trees, and the client requires the orchestrator to realize one (any one) among such trees. Specifically, each composition tree in $\mathcal{T}(\mathcal{A}_0)$ is obtained by

- unfolding the FSM and while doing so, resolving the nondeterminism by choosing a single successor state for each transition (including τ transitions); this generates a (deterministic), possibly infinite tree, whose edges are labeled by Σ_0 and whose nodes corresponding to final states of $\mathcal{A}_0$ are annotated as final;
- replacing each edge labeled by $a^{\gg}$ with an edge labeled by $(0, a, \cdot)$; this means that in the composition the client is the initiator of a;
- replacing each edge labeled by τ with a finite sequence of edges, each one labeled by $(j, a, \cdot)$, where a is some action, and $j \in [1..n]$; this means that for a τ action the composition can contain any finite sequence of interactions initiated by whatever service except by the client;
- choosing for each edge a set of servants, and adding it to the label of the edge.

Note that our framework allows for both a τ transition and a non-τ transition, say labeled by action a, originating from a same state: even if the composition synthesis chooses to realize the τ transition by the a action, the composition remains deterministic, since the two a actions have different initiators (in one case it is the client, in the other it is a service in the community).

We say that a composition O *realizes* a client specification $\mathcal{A}_0$ if $T(O) \in \mathcal{T}(\mathcal{A}_0)$.

Definition 36 (Composition Existence) Let $\mathcal{C}$ be a community of services (consisting of both pure-servant, pure-initiator and mixed services), and let $\mathcal{A}_0$

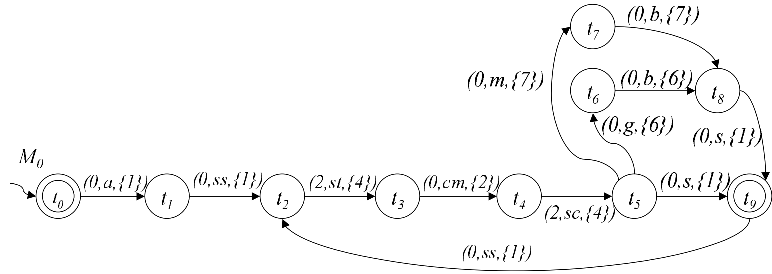
be a client specification. The problem of *composition existence* is the problem of checking whether there exists a composition that is coherent with $\mathcal{C}$ and that realizes $\mathcal{A}_0$. □

Definition 37 (Composition Synthesis) Let $\mathcal{C}$ be a community of services (consisting of both pure-servant, pure-initiator and mixed services), and let $\mathcal{A}_0$ be a client specification. The problem of *composition synthesis* is the problem of synthesizing a composition that is coherent with $\mathcal{C}$ and that realizes $\mathcal{A}_0$. □

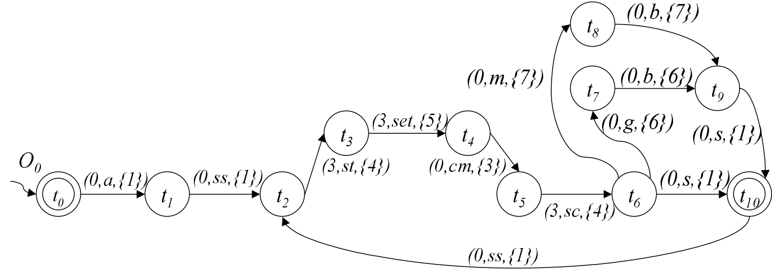
Since we are considering services that have a finite number of states, we would like also to have a composition that can be represented with a finite number of states, i.e., as a Mealy FSM (MFSM) of the form $O = (\Sigma \times [0..n], 2^{[1..n]}, S_c, s_c^0, \delta_c, \omega_c, F_c)$, where:

- $\Sigma \times [0..n]$ is the alphabet of the MFSM, which denotes actions and their initiator;
- S_c, s_c^0, δ_c, F_c are the set of states, the initial state, the transition function, and the final set of states of the MFSM, in analogy with the service FSMs;
- $2^{[1..n]}$ is the output alphabet of the MFSM, which is used to denote which are the servants of each action;
- $\omega_c : S_c \times \Sigma \times [0..n] \to 2^{[1..n]}$ is the output function of the MFSM, which, given a state, an action a, and an initiator for a, returns the set of servant services for a; we assume that the output function ω_c is defined exactly when δ_c is so.

Note that in order to define a composition, we exploited again the notion of labeled tree. We cannot directly define a composition in terms of a finite state machines for several reasons. First of all, if the τ actions appear inside a cycle (or a loop) in the client specification, in principle they can be realized by arbitrary (sequences of) actions, which may be different at each iteration. This implies that in general there may exist composition trees which cannot be compactly represented as finite state machines. Additionally, as discussed in Section 5.1 (and in Example 20) a client specification presenting non-determinism on its transitions specifies a set of execution trees. Finally, the form of the MFSM denoting the composition is in general different from the form of the FSM denoting the client specification (see, e.g., the client specifications in Figures 6.2(a) and 6.2(b) and the compositions in Figure 6.3(a) and Figure 6.3(b)). In particular, the form of the MFSM depends on how actions are delegated to services in the composition tree.



(a) Composition realizing client specifications of Figures 6.2(a) and 6.2(b)



a = authentication	*sc = spell_check*
ss = search_&_select	*g = select_gift*
st = stationery	*m = select_music*
cm = compose_message	*b = buy*
set=select_english_template	*s = send*
sit = select_italian_template	

(b) Another Composition realizing client specifications of Figures 6.2(a) and 6.2(b)

Figure 6.3: MFSM compositions coherent service community of Example 25 and realizing the client specifications of Figure 6.2(a) and Figure 6.2(b)

Example 27 *Figure 6.3 shows two MFSMs which represent possible compositions of the client specifications of Example 26 wrt the service community of Example 25.*

As an example, consider the MFSM[4] M_0 *in Figure 6.3 (b), realizing the client specification of Figure 6.2(b). Such composition specifies the client as initiator of actions* `authentication`*, and* `search_&_select`*, and the service* A_1 *as servant for both of them. Then, the composition realizes the* τ *action by specifying* A_3 *as initiator of the actions* `stationery` *and* `select_english_template`*, which are served respectively by* A_4 *and* A_5 *(note that, correctly, the client is not involved). Next, the composition specifies the*

[4]An edge (s_1, s_2) labeled (ι, a, S) indicates a transition $\delta(s_1, (a, \iota)) = s_2$ with output S, where ι is the initiator of a and S is the set of servants.

client as initiator of the action `compose_message` *with* A_3 *as servant. The second* τ *action is thus encountered and it is realized by action* `spell_check`*, initiated by* A_3 *and served by* A_4*. At this point the client is presented with a set of actions among which he chooses which one to perform next. He is therefore the initiator of:* (i) `send` *(which is served by* A_1*);* (ii) `add_gift`*, followed by* `buy` *(which are served by* A_6*) and further by* `send` *(served by* A_1*);* (iii) `add_music`*, followed by* `buy` *(which are served by* A_7*) and further by* `send` *(served by* A_1*). Finally, the client has to choose whether to stop the service execution or to send another* e-card*: in the latter case, the composition specifies that the client is initiator for the action* `search_&_select` *for which* A_1 *is servant. The MFSM of Figure 6.3(*a*)*[5] *has similar semantics.*

Note that the MFSMs M_0 *and* O_0*, shown in Figures 6.3(*a*) and Figure 6.3(b) respectively, realize not only the specification of Figure 6.2(b), but also the one in Figure 6.2(*a*). Indeed, as far the latter,* M_0 *implements* (i) *the* τ *loop on state* s_0^2 *with the action* `stationery`*, initiated by* A_2 *and served by* A_4*; and* (ii) *the* τ *loop on state* s_0^3 *with the action* `spell_check`*, also initiated by* A_2 *and served by* A_4*. Analogously,* O_0 *realizes* (i) *the* τ *loop on state* s_0^2 *with the sequence of actions* `stationery`*, initiated by* A_3 *and served by* A_4*, followed by* `select_english_template`*, initiated by* A_3 *and served by* A_5*;* (ii) *the* τ *loop on state* s_0^3 *with the action* `spell_check`*, initiated by* A_3 *and served by* A_4*. All other* τ *loops are realized by a zero length sequence of actions by both services.* □

6.4 Composition Synthesis Technique

We address the problem of composition existence and synthesis in the FSM-based framework introduced above. The basic tool we use is reducing the problem of composition existence to satisfiability of a formula written in PDL_{gm}, a variant of PDL [81] equipped with graded modalities [61, 64, 144].

6.4.1 PDL_{gm}

PDL_{gm} formulas ϕ and complex programs r can be built, starting from a set $\mathcal{P}$ of *atomic propositions* and a set $\mathcal{A}$ of *atomic actions*, by applying the constructs whose syntax and semantics has been already discussed in Section 4.2. In addition, the following one is used to capture the graded modalities:

$$(\leq n\langle a\rangle\phi)$$

[5] An other composition is similar to O_0, where the action `select_english_template` is substituted with the action `select_italian_template`, annotated with the same initiator and servant as `select_english_template`.

its semantics is:

$$\{o \mid \sharp\{(o, o') \in a^{\mathcal{I}} \mid o' \in \phi^{\mathcal{I}}\} \leq n\}$$

Intuitively, it denotes all states in the Kripke structure that are connected to at most n states where ϕ holds, through atomic action a.

As for DPDL, also the semantics of PDL_{gm} is based on the notion of Kripke structure (see Section 4.2), which for PDL_{gm} can be in general nondeterministic. The notions of model, formula satisfiability etc., introduced in Section 4.2, can be applied also to PDL_{gm}.

PDL_{gm} formally corresponds to the well-known Description Logic $\mathcal{ALCQ}_{reg}$ [41][6]. Exploiting such a correspondence, PDL_{gm} enjoys two properties that we have exploited so far and that we will exploit in the remaining of the chapters, namely the tree model property and the small model property. We recall that the tree model property says that every model of a PDL_{gm} formula can be unwound to a (possibly infinite) tree-shaped model (considering domain elements as nodes and atomic actions as edges). The small model property says that every PDL_{gm} that is satisfiable, admits a finite model whose size (in particular the number of domain elements) is at most exponential in the size of the formula itself.

6.4.2 Composition Existence

Given the specification of a client service in terms of a nondeterministic FSM $\mathcal{A}_0$ and a community of n services $A_1, \ldots, A_n$, we build a PDL_{gm} formula Φ_{tau}. As set of atomic propositions in Φ_{tau} we have *(i)* one atomic proposition s for each state s of A_j, for $j \in [0..n]$, which intuitively denotes that A_j is in state s;[7] *(ii)* atomic propositions F_j, for $j \in [0..n]$, denoting whether A_j is in a final state; *(iii)* atomic propositions served_j, for $j \in [1..n]$, denoting whether (component) FSM A_j is a servant of a transition; *(iv)* atomic propositions $\mathsf{initiated}_j$, for $j \in [0..n]$, denoting whether FSM A_j is a servant of a transition; *(v)* an atomic proposition Init representing the initial state of the required service; (*vi*) one atomic proposition a for each action $a \in \Sigma$. We have a single atomic action *trans* in Φ_{tau}, such a role will be used to denote state transitions caused by actions.

The formula Φ_{tau} is formed as follows.

- For the client specification $\mathcal{A}_0 = (\Sigma, S_0, s_0^0, \delta_0, F_0)$ we form the formula $[trans^*]\Phi_0$ where Φ_0 is the conjunction of:

[6] In Section 7.2 we will further address this correspondence.

[7] In this paper we are not concerned with compact representations of the states of the FSM. However, we observe that if states are succinctly represented (e.g., in binary format) then, in general, we can exploit such a representation in Φ_{tau} to get a corresponding compact PDL_{gm} formula Φ_{tau} as well.

- $s \rightarrow \neg s'$, for all pairs of states $s, s' \in S_0$; these say that atomic concepts representing different states are disjoint.
- $s \rightarrow \bigvee_{s' \in \delta_0(s, a^{\gg})} \langle trans \rangle (\mathsf{initiated}_0 \wedge a \wedge s')$, for each $a \in \Sigma$ and s with $\delta_0(s, a^{\gg}) \neq \emptyset$; these encode the transitions different from τ.
- $s \rightarrow \bigvee_{s' \in \delta_0(s, \tau)} \langle R_\tau^* \rangle s'$, for each s with $\delta_0(s, \tau) \neq \emptyset$, where R_τ stands for
$$((\leq 1 \langle trans \rangle \neg \mathsf{initiated}_0))?; trans; (\neg \mathsf{initiated}_0)?$$
These encode the τ transitions of $\mathcal{A}_0$; a τ transition is realized through a *single sequence* of actions in which $\mathcal{A}_0$ does not participate; the qualified number restriction is used to ensure that there is a single sequence.
- $s \rightarrow [trans](a \rightarrow \neg \mathsf{initiated}_0)$, for each a such that $\delta(s, a^{\gg})$ is not defined; these say that $a^{\gg}$ is not a possible transition.
- $s \rightarrow [trans]\mathsf{initiated}_0$, if $\delta(s, \tau)$ is not defined; these say when a τ transition is not possible.
- $F_0 \equiv \bigvee_{s \in F_0} s$; this highlights final states of $\mathcal{A}_0$.

• For each component FSM $A_i = (\Sigma, S_i, s_i^0, \delta_i, F_i)$, we form the formula $[trans^*]\Phi_i$, where Φ_i is the conjunction of:

- $s \rightarrow \neg s'$, for all distinct pairs of states $s, s' \in S_i$.
- $s \rightarrow [trans](a \wedge \mathsf{served}_i \rightarrow s')$, for each s and a such that $s' = \delta_i(s, {}^{\gg}a)$; these encode the transitions of A_i, conditioned to the fact that A_i is required to be a servant of a in the composition.
- $s \rightarrow [trans](a \wedge \mathsf{initiated}_i \rightarrow s')$, for each s and a such that $s' = \delta_i(s, a^{\gg})$; these encode the transitions of A_i, conditioned to the fact that A_i is required to be the initiator of a in the composition.
- $s \rightarrow [trans](a \rightarrow \neg \mathsf{served}_i)$, for each s and a such $\delta_i(s, {}^{\gg}a)$ is not defined.
- $s \rightarrow [trans](a \rightarrow \neg \mathsf{initiated}_i)$, for each s and a such $\delta_i(s, a^{\gg})$ is not defined.
- $s \rightarrow [trans](\mathsf{served}_i \vee \mathsf{initiated}_i \vee s)$, for each $s \in S_i$; this encodes that when A_i does not participate to an action, it does not change state.
- $F_i \equiv \bigvee_{s \in F_i} s$; this highlights final states of A_i.

• to encode the general structure of models, we form the formula $[trans^*]\Psi$, where Ψ is the conjunction of:

- $(\leq 1\langle trans\rangle(a \wedge \mathsf{initiated}_0))$, for each action $a \in \Sigma$: this represents that the realized composition is deterministic wrt to the action annotated by the initiator.
- $[trans](\bigvee_{a\in\Sigma} a)$; to represent that each transition is caused by an action.
- $[trans](\bigvee_{i\in[1..n]} \mathsf{served}_i)$; to represent that each transition must have some service as servant.
- $[trans](\bigvee_{i\in[0..n]} \mathsf{initiated}_i)$; to represent that each transition must have an initiator, either a service in the community or the client.
- $[trans](\neg\mathsf{initiated}_i \vee \neg\mathsf{initiated}_j)$, for each $i, j \in [0..n]$ with $i \neq j$; to represent that transitions have a single initiator (but possibly several servants).
- $F_0 \to \bigwedge_{i\in[1..n]} F_i$; this says that when the client specification is in a final state also all component services must be in a final state.
- $\mathsf{Init} \to s_0^0 \wedge \bigwedge_{i\in[1..n]} (s_i^0)$; it represent that initially all services are in their initial state.
- $\mathsf{Init} \to \bigwedge_{a\in\Sigma} (\neg a)$
  $\mathsf{Init} \to \bigwedge_{i\in[0..n]} (\neg\mathsf{initiated}_i)$
  $\mathsf{Init} \to \bigwedge_{i\in[1..n]} (\neg\mathsf{served}_i)$;
  they represent that initially no action has been executed yet.

Finally, we define Φ_{tau} as $\mathsf{Init} \wedge [trans^*]\Phi_0 \wedge \bigwedge_{i=1,\ldots,n}[trans^*]\Phi_i \wedge [trans^*]\Psi$.

In Section 6.4.4 we show the PDL_{gm} encoding of the scenario of Example 26 (in particular, the client service of Figure 6.2(b)) and of Example 25. Note that if τ actions are not present in a client specification, the above encoding represents an alternative way to capture the don't care non-derministic framework of Chapter 5, enriched however with the roles of initiator and servant. Moreover, since the Kleene star * is used only to mimic universal assertions, we do not need graded modalities and we can resort again to use DPDL. We will make use of such observations in Chapter 7.

In what follows, before presenting the theoretical results on composition synthesis we discuss the main differences between the DPDL encoding of Sections 4.3.1 and 5.3.1 and the PDL_{gm} encoding presented above. A first big difference depends of course on the formulas related to τ transitions, for which we do need the graded modalities, and we had to resort to PDL_{gm}, instead of DPDL.

In Sections 4.3.1 and 5.3.1 we encoded service actions as (DPDL) atomic actions while here the service actions are rendered as (PDL_{gm}) atomic propositions and all the transitions are captured by the (unique) atomic action *trans*.

This has consequences not only on the formulas encoding the FSM, but also on the master modality $[u]$. We remind that, intuitively, the master modality is used to encode universal quantification on all the possible sequences transitions, and therefore it is a reflexive and transitive closure over the set of all possible actions. In Sections 4.3.1 and 5.3.1 the set of all possible DPDL actions is given by the union of the service actions: e.g., in Example 13, where the service actions were a, l, t, we set $u = (a \cup t \cup l)^*$. Here, we have only the action *trans*, therefore we simply have $u = trans^*$. Note that because of this, also the universal quantification on the transitions originating from a state are rendered differently: for example, $[a]\cdot$ in Sections 4.3.1 and 5.3.1, corresponds here to $[trans](a \wedge \cdot)$. Also the formulas encoding non-defined transitions on the client specification are inherently different (also because of the presence of roles): the DPDL formula is $[u](s \rightarrow [a]\texttt{false})$, the PDL_{gm} formula is $[trans^*](s \rightarrow [trans](a \rightarrow \neg \mathsf{initiated}_0))$. Finally, by representing service actions as (PDL_{gm}) atomic actions, we are able to state in a simple way that in the initial state of the composition Init, no action is executed.

Another distinction is in the representation of roles. In Sections 4.3.1 and 5.3.1 the services either move or do not move. Here, due to the differentiation between being initiator or servant, a service either moves as initiator or moves as servant, or do not move. This difference becomes clear in the formula encoding component service transitions: in Sections 4.3.1 and 5.3.1, for each defined transition a, we have the formula $[u](s \rightarrow [a](\mathsf{moved}_i \wedge s' \vee \neg \mathsf{moved}_i \wedge s))$. Here, for each defined a transition, we have the two formulas $[trans^*](s \rightarrow [trans](a \wedge \mathsf{served}_i \rightarrow s'))$ and $[trans^*](s \rightarrow [trans](a \wedge \mathsf{initiated}_i \rightarrow s'))$ depending on the fact that the service is servant or initiator for a. Another difference is how it is represented the fact that a component service does not change state when it does not make a transition. In formulas of Sections 4.3.1 and 5.3.1 this information needs to be captured contextually to the definition of non made transition (either because they are not defined or because the service does not move). Here, because of the different roles, it is needed the (separate) formula $[trans^*](s \rightarrow [trans](\mathsf{served}_i \vee \mathsf{initiated}_i \vee s))$. Finally, the DPDL (single, for each action) formula $[u](\langle a \rangle \texttt{true} \rightarrow [a] \bigvee_{i=1,\ldots,n} \mathsf{moved}_i)$ saying that at each step at least one of the component FSM has moved, corresponds to the three formulas $[trans](\bigvee_{a \in \Sigma} a)$, $[trans](\bigvee_{i \in [1..n]} \mathsf{served}_i)$, and $[trans](\bigvee_{i \in [0..n]} \mathsf{initiated}_i)$. The difference in the encoding is due not only to the presence of two propositions served_i and $\mathsf{initiated}_i$ in place of the single proposition moved_i, but also because service actions are captured here as atomic propositions, whereas in Section 4.3.1 they are represented as DPDL atomic actions.

Finally, both here and in Section 5.3.1 we allow for non-determinism in the client specification, this is captured by a qualified number restriction on transitions defined on the client specification $\mathcal{A}_0$.

Lemma 38 *If there exists a composition that is coherent with $A_1, \ldots, A_n$ and that realizes the client specification $\mathcal{A}_0$, then the PDL_{gm} formula*

$$\Phi_{tau} = Init \wedge [trans^*]\Phi_0 \wedge \bigwedge_{i=1,\ldots,n} [trans^*]\Phi_i \wedge [trans^*]\Psi$$

is satisfiable.

Proof. The proof is similar to the proof of Lemma 16, however, for sake of completeness, in what follows we present the full proof.

Let O be a composition that is coherent with $A_1, \ldots, A_n$ and that realizes $\mathcal{A}_0$, and let $T(O)$ be the composition tree specified by O.

We will show that, starting from $T(O)$, we can construct a tree-like model of Φ_{tau} such that Init is satisfied in the root.

First, we define mappings σ_{tau} and σ^i_{tau} from nodes in $T(O)$ to states of A_0 and A_i, respectively, by induction on the level of nodes in $T(O)$, as follows.

- base case: $\sigma_{tau}(\varepsilon) = s_0^0$ and $\sigma^i_{tau}(\varepsilon) = s_i^0$.
- inductive case: let $\sigma_{tau}(x) = s$ and $\sigma^i_{tau}(x) = s_i$, and let the node $x \cdot (a, \iota)$ be in $T(O)$ with the edge $(x, x \cdot (a, \iota))$ labeled by (ι, a, S), where $\iota \in [0, \ldots, n]$ is the initiator for action a and $S \subseteq [1, \ldots, n]$ is the non-empty set of servants. Then we define

$$\sigma_{tau}(x \cdot (a, \iota)) = \begin{cases} s' = \delta_0(s, a) & \text{if } \iota = 0 \\ s & \text{if } \iota \neq i \end{cases}$$

and

$$\sigma^i_{tau}(x \cdot (a, \iota)) = \begin{cases} s_i{}' = \delta_i(s_i, (a, \iota)) & \text{if either } \iota = i \text{ or } i \in S \\ s_i & \text{if neither } \iota = i \text{ nor } i \in S \end{cases}$$

Once we have σ_{tau} and σ^i_{tau} in place we can define an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \{trans^{\mathcal{I}}\}, \{P^{\mathcal{I}}\}_{P \in \mathcal{P}})$ for Φ_{tau} as follows:

- $\Delta^{\mathcal{I}} = \{x \mid x \in T(O)\}$;
- $\mathsf{Init}^{\mathcal{I}} = \{\varepsilon \mid \varepsilon \in T(O)\}$;
- $trans^{\mathcal{I}} = \{(x, x \cdot (a, \iota)) \mid x, x \cdot (a, \iota) \in T(O)\}$ for each $a \in \Sigma$, and $\iota \in [0, \ldots, n]$;
- $a^{\mathcal{I}} = \{x \cdot (a, \iota) \mid (x, x \cdot (a, \iota)) \in T(O) \text{ is labeled by } (\iota, a, S) \text{ for some } S\}$, for each $a \in \Sigma$;

- $s^{\mathcal{I}} = \{x \in T(O) \mid \sigma_{tau}(x) = s\}$, for all propositions s corresponding to states of A_0;
- $s_i^{\mathcal{I}} = \{x \in T(O) \mid \sigma_{tau}^i(x) = s_i\}$, for all propositions s_i corresponding to states of A_i, for $i = 1, \ldots, n$;
- $\mathsf{initiated}_i^{\mathcal{I}} = \{x \cdot (a, \iota) \mid (x, x \cdot (a, \iota)) \in T(O)$ is labeled by (ι, a, S) with $\iota = i$ and for some $S\}$, for $i = 0, \ldots, n$;
- $\mathsf{served}_i^{\mathcal{I}} = \{x \cdot (a, \iota) \mid (x, x \cdot (a, \iota)) \in T(O)$ is labeled by (ι, a, S) with $i \in S\}$, for $i = 1, \ldots, n$;
- $F_0^{\mathcal{I}} = \{x \in T(O) \mid \sigma_{tau}(x) = s$ with $s \in F_0\}$;
- $F_i^{\mathcal{I}} = \{x \in T(O) \mid \sigma_{tau}^i(x) = s_i$ with $s_i \in F_i\}$, for $i = 1, \ldots, n$.

Such a model is essentially obtained from $T(O)$ by annotating the nodes of $T(O)$ with the states of the services, the states of the client specification, and with the initiator and servant(s) of each action.

Since $T(O)$ specifies a composition that is coherent with $A_1, \ldots, A_n$ and that realizes $\mathcal{A}_0$, it is easy to check that the interpretation $\mathcal{I}$ built as above, is a model for Φ_{tau} and that, therefore, Φ_{tau} is satisfiable. □

Lemma 39 *Any model of the PDL$_{gm}$ formula*

$$\Phi_{tau} = Init \wedge [trans^*]\Phi_0 \wedge \bigwedge_{i=1,\ldots,n} [trans^*]\Phi_i \wedge [trans^*]\Psi$$

denotes a composition that is coherent with $A_1, \ldots, A_n$ and that realizes the client specification $\mathcal{A}_0$.

Proof. The proof is similar to the proof of Lemma 17, however, for sake of completeness, in what follows we present the full proof.

If Φ_{tau} is satisfiable, then there exists a tree-like model of Φ_{tau} where Init is satisfied in the root. We will show how to derive, from such a model, a composition tree $T(O)$ that is coherent with $A_1, \ldots, A_n$ and realizes $\mathcal{A}_0$. Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \{trans^{\mathcal{I}}\}, \{P^{\mathcal{I}}\}_{P \in \mathcal{P}})$ be a tree-like model for Φ_{tau}. From $\mathcal{I}$ we can build a composition tree $T(O)$ for A_0 as follows.

- the nodes of the tree are the elements of $\Delta^{\mathcal{I}}$; actually, since $\mathcal{I}$ is tree-like we can denote the elements in $\Delta^{\mathcal{I}}$ as nodes of a tree, using the same notation that we used for composition trees; in particular, each node $x \cdot (a, \iota)$ of the tree is a prefix-closed word in $\Sigma \times \{\iota\}$, where Σ is the alphabet of the service community and therefore of $A_0, A_1, \ldots, A_n$, and $\iota \in [0, \ldots, n]$.

- the node where $\mathsf{Init}^{\mathcal{I}}$ holds is the root.
- nodes x such that $x \in F_0^{\mathcal{I}}$ are the final nodes;
- if $(x, x \cdot (a, \iota)) \in \mathit{trans}^{\mathcal{I}}$ and $x \cdot (a, \iota) \in a^{\mathcal{I}}$, and $x \cdot (a, \iota) \in \mathsf{initiated}_{\iota}^{\mathcal{I}}$ and $j \in S$ for each $x \cdot (a, \iota) \in \mathsf{served}_j^{\mathcal{I}}$ and for all $k \notin S, x \cdot (a, \iota) \notin \mathsf{served}_k^{\mathcal{I}}$, then $(x, x \cdot (a, \iota))$ is labeled by (ι, a, S).

The composition tree $T(O)$ is essentially obtained by extracting the information on initiator and servants from the interpretation of the propositions $\mathsf{initiated}_i$ and served_i.

It is straightforward to show that $T(O)$ is coherent with $A_1, \ldots, A_n$ and realizes $\mathcal{A}_0$. □

Theorem 40 *The PDL_{gm} formula*

$$\Phi_{tau} = \mathit{Init} \wedge [\mathit{trans}^*]\Phi_0 \wedge \bigwedge_{i=1,\ldots,n} [\mathit{trans}^*]\Phi_i \wedge [\mathit{trans}^*]\Psi$$

is satisfiable if and only if there exists a composition that is coherent with $A_1, \ldots, A_n$ and that realizes the client specification $\mathcal{A}_0$.

Proof. Straightforward, from Lemma 39 and 38. □

Observe that, the size of Φ_{tau} is polynomially related to the size of $\mathcal{A}_0$, $A_1, \ldots, A_n$. By Theorem 40 and the EXPTIME-completeness of satisfiability in PDL_{gm} ([41]), we get the following complexity upper bound.

Theorem 41 *Checking the existence of a composition that is coherent with $A_1, \ldots, A_n$ and that realizes a client specification $\mathcal{A}_0$ can be done in EXPTIME.* □

6.4.3 Synthesizing a Composition

In the previous section we have shown how to check the existence of a composition. In this section we show how to synthesize a composition which is a FSM, by presenting a sound, complete and terminating technique.

By Theorem 40, if the formula Φ_{tau} encoding the composition problem is satisfiable, then it admits a model, which is the composition we want to synthesize. Conversely, if Φ_{tau} is not satisfiable, no model exists, therefore, the encoded composition problem admits no solution. However, Theorem 40 considers only compositions which are (possibly infinite) trees. Instead, we are interested on finite compositions. By the small model property of PDL_{gm}, if Φ_{tau} is satisfiable, then it is satisfiable in a model that is at most exponential in the size of Φ_{tau}. From such a finite model one can extract a representation of the composition that has the form of a MFSM.

Definition 42 (Mealy Composition) Given a finite model $\mathcal{I}_f = (\Delta^{\mathcal{I}_f}, \{trans^{\mathcal{I}_f}\}, \{P^{\mathcal{I}_f}\}_{P\in\mathcal{P}})$ we define Mealy composition (wrt the framework considered in this chapter) an MFSM $O = (\Sigma \times [0..n], 2^{[1..n]}, S_c, s_c^0, \delta_c, \omega_c, F_c)$, built as follows:

- $S_c = \Delta^{\mathcal{I}_f}$;
- $s_c^0 = \mathsf{Init}^{\mathcal{I}_f}$;
- $\delta_c(s, (a, \iota)) = s'$ iff $(s, s') \in trans^{\mathcal{I}_f}$, $s' \in a^{\mathcal{I}_f}$, and $s' \in \mathsf{initiated}_\iota^{\mathcal{I}_f}$;
- $\omega_c(s, (a, \iota)) = \{j_1, \dots, j_\ell\}$ iff $(s, s') \in trans^{\mathcal{I}_f}$, $s' \in a^{\mathcal{I}_f}$, $s' \in \mathsf{initiated}_\iota^{\mathcal{I}_f}$, and $s' \in \mathsf{served}_j^{\mathcal{I}_f}$, for exactly those j in $\{j_1, \dots, j_\ell\}$;
- $F_c = F_0^{\mathcal{I}_f}$.

□

As a consequence of this, we get the following results.

Theorem 43 *If there exists a composition that is coherent with $A_1, \dots, A_n$ and that realizes a client specification $\mathcal{A}_0$, then there exists one that is a Mealy composition of size at most exponential in the size of $\mathcal{A}_0, A_1, \dots, A_n$.*

Proof. By Theorem 40, if there exists a composition tree, then the PDL_{gm} formula Φ_{tau} constructed as above is satisfiable. In turn, if Φ_{tau} is satisfiable, for the small-model property of PDL_{gm}, there exists a model $\mathcal{I}_f$ of size at most exponential in Φ_{tau}, and hence in $\mathcal{A}_0$ and $A_1, \dots, A_n$. From $\mathcal{I}_f$ we can construct a MFSM A_c as in Definition 42. Notice that the composition tree generated by A_c essentially corresponds the tree-like model obtained by unwinding $\mathcal{I}_f$. □

Theorem 44 *Any finite model of the PDL_{gm} formula*

$$\Phi_{tau} = Init \wedge [trans^*]\Phi_0 \wedge \bigwedge_{i=1,\dots,n} [trans^*]\Phi_i \wedge [trans^*]\Psi$$

constructed as in the previous section denotes a Mealy composition of that realizes A_0 and that is coherent with $A_1, \dots, A_n$.

Proof. By construction, observing that the construction of the Mealy composition from a finite model is semantic-preserving. □

Exploiting reasoning methods for PDL_{gm} based on model construction, such as tableaux algorithms [39, 52, 7], one can actually construct such a

Mealy composition. Notice that such algorithms need to be able to deal with full reflexive transitive closure, introduced in Φ_{tau} due to τ transitions in the client specification.

Consider again the algorithm for synthesizing a Mealy composition shown in Figure 4.6. Beside the function `Minimize`, for minimizing the MFSM, that is independent from the framework, all the others needs to be tailored towards the new framework: *(i)* `FSM2DPDL` should be substituted with the function $\texttt{FSM2PDL}_{gm}$: it translates the FSMs representing the services in the community and the client specification in the PDL_{gm} encoding, and it must also cope with the presence of both don't care non-determinism and τ actions in the client specification; *(ii)* `DPDLTableau` should be substituted with $\texttt{PDL}_{gm}\texttt{Tableau}$: it checks satisfiability of the PDL_{gm} formula and builds a finite model if it exists or returning (**nil**), otherwise, and it also needs to deal with full reflexive transitive closure and the graded modalities; *(iii)* `Extract_MFSM`: it extracts a Mealy composition, and in additional it should be able to deal with the roles of a service wrt a given action.

In the following section, we show how to encode our running example in a PDL_{gm} formula Φ_{tau}, discuss a model of it and show that it is indeed a Mealy composition for the loose client specification of Figure 6.2(b).

6.4.4 Composition synthesis of our Running Example

In this section, we encode in a PDL_{gm} formula Φ_{tau} the scenario of Example 26 (in particular, the client specification of Figure 6.2(b)) and of Example 25.

The set P of atomic propositions is:

$$\begin{array}{lll} P & = \{ & s_0^0, s_0^1, s_0^2, s_0^3, s_0^4, s_0^5, s_0^6, s_0^7, s_0^8, s_1^0, s_1^1, s_1^2, s_2^0, s_2^1, s_2^2, s_3^0, s_3^1, s_3^2, s_3^3, s_4^0, s_4^1, \\ & & s_5^0, s_6^0, s_6^1, s_7^0, s_7^1, F_0, F_1, F_2, F_3, F_4, F_5, F_6, F_7, \mathsf{served}_1, \mathsf{served}_2, \mathsf{served}_3, \\ & & \mathsf{served}_4, \mathsf{served}_5, \mathsf{served}_6, \mathsf{served}_7, \mathsf{initiated}_0, \mathsf{initiated}_1, \mathsf{initiated}_2, \\ & & \mathsf{initiated}_3, \mathsf{initiated}_4, \mathsf{initiated}_5, \mathsf{initiated}_6, \mathsf{initiated}_7, \mathsf{Init}, a, ss, st, \\ & & cm, set, sit, sc, g, m, b, s\} \end{array}$$

where:

- a denotes action `authentication`
- ss denotes action `search_&_select`
- st denotes action `stationery`
- cm denotes action `compose_message`
- set denotes action `select_english_template`
- sit denotes action `select_italian_template`

- sc denotes action `spell_check`
- g denotes action `add_gift`
- m denotes action `add_music`
- b denotes action `buy`
- s denotes action `send`

The atomic propositions in $\mathcal{P}$ have the following semantics:

- s_j^i, for $j = 0, \ldots, 7$ and $i = 0, \ldots, 8$: s_j^i is true if and only if A_j is in state s_j^i;
- $F_j, j = 0, \ldots, 7$: F_j is true if and only if A_j is in a final state;
- $\mathsf{served}_j, j = 1, \ldots, 7$: served_j is true if and only if (component) FSM A_j is servant for the current transition;
- $\mathsf{initiated}_j, j = 0, \ldots, 7$: $\mathsf{initiated}_j$ is true if and only if FSM A_j (denoting either the client specification, or a service in the community) is initiator for the current transition;
- Init is true if and only if the composition is in the initial state;
- each proposition denoting an action is true if and only if that action is executed in the current transition

The **client specification** $\mathcal{A}_0$ is captured by the formula $[trans^*]\Phi_0$, where Φ_0 is obtained by conjunction of the formulas below. We want to remark that, since (Φ_0, and therefore) all the formulas below are prefixed by $[trans^*]$, the observations that we will make hold for all transitions of $\mathcal{A}_0$ (in particular, for those executed at each iteration of cycles).

$\mathcal{A}_0$ cannot be simultaneously in two (or more) states (propositions representing different states are disjoint):

$$
\begin{array}{llll}
[u](s_0^0 \rightarrow \neg s_0^1) & [u](s_0^1 \rightarrow \neg s_0^2) & [u](s_0^2 \rightarrow \neg s_0^4) & [u](s_0^4 \rightarrow \neg s_0^7) \\
[u](s_0^0 \rightarrow \neg s_0^2) & [u](s_0^1 \rightarrow \neg s_0^3) & [u](s_0^2 \rightarrow \neg s_0^5) & [u](s_0^4 \rightarrow \neg s_0^8) \\
[u](s_0^0 \rightarrow \neg s_0^3) & [u](s_0^1 \rightarrow \neg s_0^4) & [u](s_0^2 \rightarrow \neg s_0^6) & [u](s_0^5 \rightarrow \neg s_0^6) \\
[u](s_0^0 \rightarrow \neg s_0^4) & [u](s_0^1 \rightarrow \neg s_0^5) & \ldots & [u](s_0^5 \rightarrow \neg s_0^7) \\
[u](s_0^0 \rightarrow \neg s_0^5) & [u](s_0^1 \rightarrow \neg s_0^6) & \ldots & [u](s_0^5 \rightarrow \neg s_0^8) \\
[u](s_0^0 \rightarrow \neg s_0^6) & [u](s_0^1 \rightarrow \neg s_0^7) & [u](s_0^3 \rightarrow \neg s_0^8) & [u](s_0^6 \rightarrow \neg s_0^7) \\
[u](s_0^0 \rightarrow \neg s_0^7) & [u](s_0^1 \rightarrow \neg s_0^8) & [u](s_0^4 \rightarrow \neg s_0^5) & [u](s_0^6 \rightarrow \neg s_0^8) \\
[u](s_0^0 \rightarrow \neg s_0^8) & [u](s_0^2 \rightarrow \neg s_0^3) & [u](s_0^4 \rightarrow \neg s_0^6) & [u](s_0^7 \rightarrow \neg s_0^8)
\end{array}
$$

The transitions (different from τ) that $\mathcal{A}_0$ performs as initiator are captured by:

$$\begin{array}{lcl}
s_0^0 & \rightarrow & \langle trans\rangle(\mathsf{initiated}_0 \wedge a \wedge s_0^1)\\
s_0^1 & \rightarrow & \langle trans\rangle(\mathsf{initiated}_0 \wedge ss \wedge s_0^2)\\
s_0^3 & \rightarrow & \langle trans\rangle(\mathsf{initiated}_0 \wedge cm \wedge s_0^4)\\
s_0^5 & \rightarrow & \langle trans\rangle(\mathsf{initiated}_0 \wedge m \wedge s_0^6)\\
s_0^5 & \rightarrow & \langle trans\rangle(\mathsf{initiated}_0 \wedge g \wedge s_0^6)\\
s_0^5 & \rightarrow & \langle trans\rangle(\mathsf{initiated}_0 \wedge s \wedge s_0^8)\\
s_0^6 & \rightarrow & \langle trans\rangle(\mathsf{initiated}_0 \wedge b \wedge s_0^7)\\
s_0^7 & \rightarrow & \langle trans\rangle(\mathsf{initiated}_0 \wedge s \wedge s_0^8)\\
s_0^8 & \rightarrow & \langle trans\rangle(\mathsf{initiated}_0 \wedge ss \wedge s_0^2)
\end{array}$$

Consider the first formula above. Intuitively, it says that if the FSM $\mathcal{A}_0$ is in state s_0^0 then there exists a transition ($\langle trans\rangle$) ending in a state[8] where a has been executed (a is true) by $\mathcal{A}_0$[9] ($\mathsf{initiated}_0$ is true) and $\mathcal{A}_0$ has moved to state s_0^1. Note that since in our example $\mathcal{A}_0$ is deterministic, no formula above contains *or*-ed terms on the right hand side. However, similarly to what noticed in Example 22, since several different actions can be executed from a state, several formula above contain *and*-ed (or better *and*-able) terms. For instance, from state s_0^5 the client can perform any action among m, g or s, therefore the three formulas having s_0^5 on the left hand side, can be factorized into the unique formula $s_0^5 \rightarrow (\langle trans\rangle(\mathsf{initiated}_0 \wedge g \wedge s_0^6)) \wedge (\langle trans\rangle(\mathsf{initiated}_0 \wedge g \wedge s_0^6)) \wedge (\langle trans\rangle(\mathsf{initiated}_0 \wedge s \wedge s_0^8))$.

The τ-transitions that $\mathcal{A}_0$ specifies are captured by:

$$s_0^2 \rightarrow \langle R_\tau^*\rangle s_0^3 \qquad\qquad s_0^4 \rightarrow \langle R_\tau^*\rangle s_0^5$$

where R_τ stands for $((\leq 1\langle trans\rangle\neg\mathsf{initiated}_0))?; trans; (\neg\mathsf{initiated}_0)?$.

The first formula says that if $\mathcal{A}_0$ is in state s_0^2 then there exists a sequence of transitions that satisfies the conditions imposed by R_τ^* and that leads $\mathcal{A}_0$ in state s_0^3. In general, R_τ^* can be explicited as:

$$\begin{array}{l}
((\leq 1\langle trans\rangle\neg\mathsf{initiated}_0))?; trans; (\neg\mathsf{initiated}_0)? \;\cup\; ((\leq 1\langle trans\rangle\neg\mathsf{initiated}_0))?;\\
trans; (\neg\mathsf{initiated}_0)?; ((\leq 1\langle trans\rangle\neg\mathsf{initiated}_0))?; trans; (\neg\mathsf{initiated}_0)? \;\cup\\
((\leq 1\langle trans\rangle\neg\mathsf{initiated}_0))?; trans; (\neg\mathsf{initiated}_0)?; ((\leq 1\langle trans\rangle\neg\mathsf{initiated}_0))?; trans;\\
(\neg\mathsf{initiated}_0)?; ((\leq 1\langle trans\rangle\neg\mathsf{initiated}_0))?; trans; (\neg\mathsf{initiated}_0)? \;\cup\; \ldots
\end{array}$$

Informally, in the context of the first formula above, R_τ^* assume the following meaning:

[8] Both the transition and the state mentioned must exist in each model of the formula Φ_{tau}.

[9] I.e., by the client.

1. Test (?) if there exist at most one transition ($(\leq 1\langle trans\rangle)$) leading into state s_0^3 for which $\mathcal{A}_0$ is not initiator ($\neg\mathsf{initiated}_0$). If the test fails, then the τ transition is realized by a $n-1$-length sequence, if n is the number of times this test is executed.

2. If the test succeeds, then execute such a transition ($trans$)

3. Test (?) if in the current state $\mathcal{A}_0$ is not the initiator ($\neg\mathsf{initiated}_0$). If in the current state $\mathcal{A}_0$ is the initiator, then the τ transition is realized by a sequence of actions of length one. If in the current state $\mathcal{A}_0$ is not the initiator, then the first test is performed again.[10]

Note that in our example the τ transitions of $\mathcal{A}_0$ are realized by a non-zero sequence of actions: the first test does not fail, since τ always connects two different states, and $\mathcal{A}_0$ is not the initiator of any *executed* action (that realizes τ), neither leading towards s_0^3 nor towards s_0^5. Observe that when a τ action is realized by more than one service action, this formula does not specify any requirements about the state of $\mathcal{A}_0$ during the execution of intermediate service actions.

At this point one may wonder under which circumstances a τ action can be realized by a zero length sequence of actions. This happens when there is a state with both a τ loop and an outgoing non-τ transition, as in the client specification $\mathcal{B}_0$ of Figure 6.2 (b). Consider state s_0^1, from which both a τ loop and a ss transition, initiated by $\mathcal{B}_0$ originate. In s_0^1 the first test can non-deterministically fail (if the ss transition is considered) or succeed (if the τ loop is taken).

Note in the general formula, the presence of $\bigvee_{s'\in\delta_0(s,\tau)}$. This means that it is allowed to specify more than one τ transition originating from the same state, i.e., a client specification can be non-deterministic on τ actions. This situation is handled in the same way as for non-τ actions.

The transitions, different from τ, that are not defined on $\mathcal{A}_0$ are captured (in part) by:

$s_0^0 \rightarrow [trans](ss \rightarrow \neg\mathsf{initiated}_0)$	$\ldots$
$s_0^0 \rightarrow [trans](cm \rightarrow \neg\mathsf{initiated}_0)$	$s_0^5 \rightarrow [trans](a \rightarrow \neg\mathsf{initiated}_0)$
$s_0^0 \rightarrow [trans](m \rightarrow \neg\mathsf{initiated}_0)$	$s_0^5 \rightarrow [trans](ss \rightarrow \neg\mathsf{initiated}_0)$
$s_0^0 \rightarrow [trans](g \rightarrow \neg\mathsf{initiated}_0)$	$s_0^5 \rightarrow [trans](cm \rightarrow \neg\mathsf{initiated}_0)$
$s_0^0 \rightarrow [trans](b \rightarrow \neg\mathsf{initiated}_0)$	$s_0^5 \rightarrow [trans](b \rightarrow \neg\mathsf{initiated}_0)$
$s_0^0 \rightarrow [trans](s \rightarrow \neg\mathsf{initiated}_0)$	$s_0^6 \rightarrow [trans](a \rightarrow \neg\mathsf{initiated}_0)$
$s_0^1 \rightarrow [trans](a \rightarrow \neg\mathsf{initiated}_0)$	$s_0^6 \rightarrow [trans](ss \rightarrow \neg\mathsf{initiated}_0)$
$\ldots$	$\ldots$

[10]Note that this second test is needed to capture τ loops.

Intuitively, the first formula of the first column says that if $\mathcal{A}_0$ is in state s_0^1 then for each transition originating from s_0^1, if ss is the executed action, then necessarily it is not initiated by $\mathcal{A}_0$.

The τ transitions that are not defined on $\mathcal{A}_0$ are captured by:

$$\begin{array}{ll} s_0^0 \rightarrow [trans]\mathsf{initiated}_0 & s_0^6 \rightarrow [trans]\mathsf{initiated}_0 \\ s_0^1 \rightarrow [trans]\mathsf{initiated}_0 & s_0^7 \rightarrow [trans]\mathsf{initiated}_0 \\ s_0^3 \rightarrow [trans]\mathsf{initiated}_0 & s_0^8 \rightarrow [trans]\mathsf{initiated}_0 \\ s_0^5 \rightarrow [trans]\mathsf{initiated}_0 & \end{array}$$

Intuitively, the first formula of the first column says that if $\mathcal{A}_0$ is in state s_0^1 then $\mathcal{A}_0$ is the initiator of all transition originating from s_0^1.

Finally, the final states of $\mathcal{A}_0$ are encoded by:

$$F_0 \equiv s_0^0 \vee s_0^8$$

Formulas capturing **the services in the community**. For sake of succinctness, we show the encoding only on the service A_3, which acts both as initiator and as servants for the actions. A_3 is captured by the formula $[trans^*]\Phi_3$, where Φ_3 is obtained by conjunction of the formulas below.

Note again that, since (Φ_3, and therefore) all the formulas below are prefixed by $[trans^*]$, the observations that we will make hold for all transitions of A_3 (in particular, for those executed at each iteration of cycles).

Formulas encoding that A_3 cannot be at the same time in two different states (i.e., no two atomic propositions representing states are true at the same time) are:

$$\begin{array}{lll} [u](s_3^0 \rightarrow \neg s_3^1) & [u](s_3^0 \rightarrow \neg s_3^3) & [u](s_3^1 \rightarrow \neg s_3^3) \\ [u](s_3^0 \rightarrow \neg s_3^2) & [u](s_3^1 \rightarrow \neg s_3^2) & [u](s_3^2 \rightarrow \neg s_3^3) \end{array}$$

In the composition, A_3 can be required to act as servant only for the cm transition. Such transition is encoded as

$$s_3^2 \rightarrow [trans](cm \wedge \mathsf{served}_3 \rightarrow s_3^3)$$

stating that if A_3 is in state s_3^2, for each transition originating from s_3^2 such that A_3 executes action cm as servant, A_3 moves to state s_3^3. Note that in principle A_3 could also be initiator for the same action cm, originating from the same state s_3^2 (but leading in another state).

The transitions of A_3, for which it can be required to act as initiator in the composition, are the following ones:

$$\begin{array}{l} s_3^0 \rightarrow [trans](st \wedge \mathsf{initiated}_3 \rightarrow s_3^1) \\ s_3^1 \rightarrow [trans](sit \wedge \mathsf{initiated}_3 \rightarrow s_3^2) \\ s_3^1 \rightarrow [trans](set \wedge \mathsf{initiated}_3 \rightarrow s_3^2) \\ s_3^3 \rightarrow [trans](sc \wedge \mathsf{initiated}_3 \rightarrow s_3^0) \end{array}$$

Similarly to the previous set of formulas, the first formula states that if A_3 is in state s_3^0, each transition originating from s_3^0 such that st is executed having A_3 as servant makes A_3 move to state s_3^1.

The transitions of A_3 for which A_3 acts as servant and that are not defined are (in part) the following ones:

$s_3^0 \rightarrow [trans](a \rightarrow \neg \mathsf{served}_3)$	...
$s_3^0 \rightarrow [trans](ss \rightarrow \neg \mathsf{served}_3)$	$s_3^2 \rightarrow [trans](a \rightarrow \neg \mathsf{served}_3)$
$s_3^0 \rightarrow [trans](st \rightarrow \neg \mathsf{served}_3)$	$s_3^2 \rightarrow [trans](ss \rightarrow \neg \mathsf{served}_3)$
$s_3^0 \rightarrow [trans](cm \rightarrow \neg \mathsf{served}_3)$	$s_3^2 \rightarrow [trans](st \rightarrow \neg \mathsf{served}_3)$
$s_3^0 \rightarrow [trans](sit \rightarrow \neg \mathsf{served}_3)$	$s_3^2 \rightarrow [trans](sit \rightarrow \neg \mathsf{served}_3)$
$s_3^0 \rightarrow [trans](set \rightarrow \neg \mathsf{served}_3)$	$s_3^2 \rightarrow [trans](set \rightarrow \neg \mathsf{served}_3)$
$s_3^0 \rightarrow [trans](sc \rightarrow \neg \mathsf{served}_3)$	$s_3^2 \rightarrow [trans](sc \rightarrow \neg \mathsf{served}_3)$
...	...

Intuitively, the first formula of the first column states that if A_3 is in state s_3^0, then all transitions that can be executed from s_3^0 leads A_3 into a state such that if a is executed then A_3 has not been servant for a.

The transitions of A_3 for which A_3 acts as initiator and that are not defined are (in part) the following ones:

$s_3^0 \rightarrow [trans](a \rightarrow \neg \mathsf{initiated}_3)$	...
$s_3^0 \rightarrow [trans](ss \rightarrow \neg \mathsf{initiated}_3)$	$s_3^2 \rightarrow [trans](a \rightarrow \neg \mathsf{initiated}_3)$
$s_3^0 \rightarrow [trans](cm \rightarrow \neg \mathsf{initiated}_3)$	$s_3^2 \rightarrow [trans](ss \rightarrow \neg \mathsf{initiated}_3)$
$s_3^0 \rightarrow [trans](sit \rightarrow \neg \mathsf{initiated}_3)$	$s_3^2 \rightarrow [trans](st \rightarrow \neg \mathsf{initiated}_3)$
$s_3^0 \rightarrow [trans](set \rightarrow \neg \mathsf{initiated}_3)$	$s_3^2 \rightarrow [trans](cm \rightarrow \neg \mathsf{initiated}_3)$
$s_3^0 \rightarrow [trans](sc \rightarrow \neg \mathsf{initiated}_3)$	$s_3^2 \rightarrow [trans](sit \rightarrow \neg \mathsf{initiated}_3)$
...	...

The meaning of this set of formulas is similar to the previous set. For example, the first formula of the first column says that states that if A_3 is in state s_3^0, then all transitions that can be executed from s_3^0 leads A_3 into a state such that if a is executed then A_3 has not been initiator for a.

This set of formulas encodes that when A_3 does not participate to an action, it does not change state:

$$
\begin{array}{l}
s_3^0 \rightarrow [trans](\mathsf{served}_3 \vee \mathsf{initiated}_3 \vee s_3^0) \\
s_3^1 \rightarrow [trans](\mathsf{served}_3 \vee \mathsf{initiated}_3 \vee s_3^1) \\
s_3^2 \rightarrow [trans](\mathsf{served}_3 \vee \mathsf{initiated}_3 \vee s_3^2) \\
s_3^3 \rightarrow [trans](\mathsf{served}_3 \vee \mathsf{initiated}_3 \vee s_3^3)
\end{array}
$$

For example, the first formula says that if A_3 is in state s_3^0, all transitions originating from there leads A_3 in a where either A_3 is a servant or an initiator (when it moves), or it remains in state s_3^0 (when it does not move). Note that if there is a loop in state s_3^0, then both s_3^0 and one between served_3 and $\mathsf{initiated}_3$,

would be true.

Final states are:

$$F_3 \equiv s_3^0$$

The **domain independent conditions** are captured by the formula $[trans^*]\Psi$, where Ψ is the conjunction of the formulas below. Again, Ψ and hence all the formulas below are prefixed by $[trans^*]$, therefore the observations that we will make hold for all transitions of $\mathcal{A}_0$ (in particular, for those executed at each iteration of cycles).

The realized composition is deterministic wrt to the action annotated by the initiator.

$$(\leq 1\langle trans\rangle(a \wedge \mathsf{initiated}_0))$$
$$(\leq 1\langle trans\rangle(ss \wedge \mathsf{initiated}_0))$$
$$(\leq 1\langle trans\rangle(st \wedge \mathsf{initiated}_0))$$
$$(\leq 1\langle trans\rangle(cm \wedge \mathsf{initiated}_0))$$
$$(\leq 1\langle trans\rangle(set \wedge \mathsf{initiated}_0))$$
$$(\leq 1\langle trans\rangle(sit \wedge \mathsf{initiated}_0))$$
$$(\leq 1\langle trans\rangle(sc \wedge \mathsf{initiated}_0))$$
$$(\leq 1\langle trans\rangle(m \wedge \mathsf{initiated}_0))$$
$$(\leq 1\langle trans\rangle(g \wedge \mathsf{initiated}_0))$$
$$(\leq 1\langle trans\rangle(b \wedge \mathsf{initiated}_0))$$
$$(\leq 1\langle trans\rangle(s \wedge \mathsf{initiated}_0))$$

For example, the first formula of the first column says that from the current state of the composition, at most one transition must exist for which action a is initiated by $\mathcal{A}_0$ (i.e., by the client). However, this does not prevent from having two a transitions from the same state, having different initiators.

Each transition is caused by an action:

$$[trans](a \vee ss \vee st \vee set \vee sit \vee cm \vee sc \vee m \vee g \vee b \vee s)$$

Each transition must have at least one service as servant ...

$$[trans](\mathsf{served}_1 \vee \mathsf{served}_2 \vee \mathsf{served}_3 \vee \mathsf{served}_4 \vee \mathsf{served}_5 \vee \mathsf{served}_6 \vee \mathsf{served}_7)$$

... and at least one service as initiator (including $\mathcal{A}_0$)

$$[trans] \quad (\mathsf{initiated}_0 \vee \mathsf{initiated}_1 \vee \mathsf{initiated}_2 \vee \mathsf{initiated}_3 \vee \mathsf{initiated}_4 \vee \mathsf{initiated}_5 \vee \mathsf{initiated}_6 \vee \mathsf{initiated}_7)$$

Each action has at most one initiator (i.e., an action is never initiated by two services)

$$[trans](\neg\mathsf{initiated}_1 \vee \neg\mathsf{initiated}_2)$$
$$[trans](\neg\mathsf{initiated}_1 \vee \neg\mathsf{initiated}_3)$$
$$[trans](\neg\mathsf{initiated}_1 \vee \neg\mathsf{initiated}_4)$$
$$[trans](\neg\mathsf{initiated}_1 \vee \neg\mathsf{initiated}_5)$$
$$[trans](\neg\mathsf{initiated}_1 \vee \neg\mathsf{initiated}_6)$$
$$[trans](\neg\mathsf{initiated}_1 \vee \neg\mathsf{initiated}_7)$$
...
$$[trans](\neg\mathsf{initiated}_5 \vee \neg\mathsf{initiated}_6)$$
$$[trans](\neg\mathsf{initiated}_5 \vee \neg\mathsf{initiated}_7)$$
$$[trans](\neg\mathsf{initiated}_6 \vee \neg\mathsf{initiated}_7)$$

When the client specification is in a final state also all component services must be in a final state.

$$F_0 \rightarrow F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge F_5 \wedge F_6 \wedge F_7$$

In the initial state of the composition, all services are in their initial state

$$\mathsf{Init} \rightarrow s_0^0 \wedge s_1^0 \wedge s_2^0 \wedge s_3^0 \wedge s_4^0 \wedge s_5^0 \wedge s_6^0 \wedge s_7^0$$

In the initial state of the composition, no action has been executed yet.

$$\begin{array}{lll} \mathsf{Init} & \rightarrow & (\neg a) \wedge (\neg ss) \wedge (\neg st) \wedge (\neg set) \wedge (\neg sit) \wedge (\neg cm) \wedge (\neg sc) \wedge \\ & & (\neg m) \wedge (\neg g) \wedge (\neg b) \wedge (\neg s) \end{array}$$

$$\begin{array}{lll} \mathsf{Init} & \rightarrow & (\neg \mathsf{initiated}_1) \wedge (\neg \mathsf{initiated}_2) \wedge (\neg \mathsf{initiated}_3) \wedge (\neg \mathsf{initiated}_4) \wedge \\ & & (\neg \mathsf{initiated}_5) \wedge (\neg \mathsf{initiated}_6) \wedge (\neg \mathsf{initiated}_7) \end{array}$$

$$\begin{array}{lll} \mathsf{Init} & \rightarrow & (\neg \mathsf{served}_1) \wedge (\neg \mathsf{served}_2) \wedge (\neg \mathsf{served}_3) \wedge (\neg \mathsf{served}_4) \wedge \\ & & (\neg \mathsf{served}_5) \wedge (\neg \mathsf{served}_6) \wedge (\neg \mathsf{served}_7) \end{array}$$

The PDL_{gm} formula Φ_{tau} that encodes the scenario of Figures 6.2(b) 6.1 is defined as $\mathsf{Init} \wedge [trans^*]\Phi_0 \wedge \bigwedge_{i=1,\ldots,n}[trans^*]\Phi_i \wedge [trans^*]\Psi$.

We conclude the section by showing a model of the PDL_{gm} formula Φ_{tau}.

Figure 6.4 shows an interpretation (i.e., a Kripke structure) $\mathcal{I}_f$ for Φ_{tau} which is returned from a PDL_{gm} tableau algorithm that deals with full reflexive transitive closure. For sake of clarity, for each state we report only a list of the atomic propositions that are true in that state and we underline the propositions denoting states that change their value when an action is performed. Additionally, in Figure 6.4, we decided to label each transition between states by the triple (a, ι, S) where a is an action, ι is its initiator and S the set of servants. We do it in order to help the reader to understand how to obtain a Mealy composition, whose transitions are labeled by (a, ι, S), starting from the PDL_{gm} atomic propositions denoting the performed action, its initiator, and its servants. For sake of exactness, in the figure, the transitions between states should have been labeled by *trans*.

It is easy to verify that the interpretation in Figure 6.4 is indeed a (finite) model of Φ_{tau} since for each state (domain element), all constraints expressed in Φ_{tau} are satisfied. For example, in the initial state, *(i)* Init holds, *(ii)* all the FSMs associated to the component services and to the client specification start from their initial state, *(iii)* no action is executed, therefore no service acts as initiator nor as a servant, *(iv)* since the composition is in a final state,

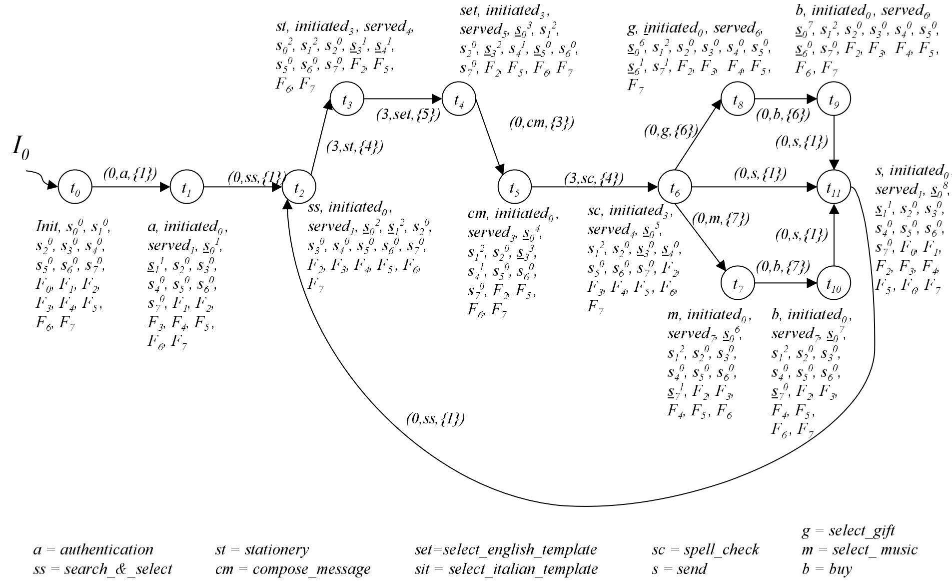


Figure 6.4: Finite model of the PDL_{gm} formula encoding our running example.

also all the component services are in a final state. State t_1 is reached after performing action a, whose initiator is the client and whose servant is A_1: therefore, both the client specification and A_1 moved, respectively, to states s_0^1 and s_1^1, while the other services do not change state.

It is easy to see that $\mathcal{I}_f$ is a Mealy composition for the underspecified client specification $\mathcal{A}_0$ of Figure 6.2 (b). Note that $\mathcal{I}_f$ is not minimal: by applying standard minimization techniques one obtains the finite state machine of Figure 6.3 (b). A detailed analysis of $\mathcal{I}_f$ allows us to make the following observations:

- A_2 never moves and it always remains in its initial state s_2^0, which is also final.
- Action `buy` can be performed both by A_6 and by A_7: each execution of `buy` ends in a different state (t_9 and t_{10} in the figure), characterized by a different value of the propositions served_6 and served_7: in t_9 $\mathsf{served}_6 =$ `true` and $\mathsf{served}_7 =$ `false` because `buy` is performed by A_6, while in t_{10} $\mathsf{served}_6 =$ `false` and $\mathsf{served}_7 =$ `true` because `buy` is performed by A_5. The reader is invited to compare this situation with a similar situation in the DPDL model of Example 23. Therefore, the minimal FSM shown in Figure 6.3 (b) is not a model of the PDL_{gm} formula, because there the two `buy` transitions end in the same state.
- State t_1 is not final, despite the fact that all component services are in

a final state, because the target service is not in a final state. The final states are t_0 and t_{11}.

- In this model s_0^3 holds in t_4 (but not in t_3). In fact the sub-formula of Φ_{tau} encoding τ transitions does not require any specific state to be assigned to the intermediate steps that realize the τ transition, therefore, in the model s_0^3 could hold in t_3, as well.

6.5 Discussion

In this chapter we study the problem of service composition by following the approach of Chapter 4 and 5 (see Sections 4.4 and 5.4 for a discussion on the features of our approach to service composition, developed in Chapters 4 and 5, respectively), but introduce two fundamental extensions:

1. The composition is again based on controlling the concurrent execution of the the available component services, but in addition it allows for *synchronization and communication* between the component services. These aspects are not present in Chapters 4 and 5. Here instead, we introduce the notion of *initiator* and *servant* of an (inter)action, and we require that each action involves one initiator and one or more servants that suitably synchronize and exchange information in order to complete the action. The composition can control who is interacting at each step and allows two component services to interact and synchronize suitably before starting to serve the client, or while serving him. This provides us with a bridge towards the message-based model of services [40], and towards the service communication model that form the basis to standard languages, such as BPEL4WS [4].

2. The client request is again a specification of the transition system that the client would like to execute. Such a specification may contain incomplete information, not only in the form of don't care non-determinism, as in Chapter 5, but also by allowing the activities in which the client is involved (i.e., those described by its transition system) to be interleaved in specified points with activities that are performed by the component services without the client intervention (but of which the client is in any case aware); this allows the client to exploit the synchronization and communication abilities that the component services have (cf. point 1 above) to allow such services to perform some preliminary/extra work before or while serving him.

In such a framework, we again develop sound, complete and terminating technique for checking the existence of a composition and synthesizing one, if possible.

Note that, if a a client specification does not present τ actions, but only don't care non-determinism, the PDL_{gm} encoding does not contain graded modalities, hence it is in fact a DPDL encoding. Therefore, the technique developed in this chapter constitutes an alternative way to compute automatic composition in the framework of Chapter 5, enriched however with the roles of initiator and servant.

In service composition, there is a clear distinction between the role of the client asking for a service, and the role of the software artifact that realizes the service. In our model, such a distinction is reflected in the fact that a client is always considered as an initiator of actions, and not as a servant, while a service is seen mostly as a servant. By blurring such a distinction, one makes the notion of service similar to the notion of agent. Under this perspective, the results presented here become relevant for automatic synthesis of agents. In the future, we aim at investigating this perspective in detail.

Finally, note that in order to check the existence of service composition, instead of reducing such problem to satisfiability of a Proposition Dynamic Logic formula, we could as well exploit results in the research area of program synthesis (see e.g.[82, 94, 93, 60]). We will develop this point as future research (see Section 8.2), keeping also in mind the observations of Section 2.3.2.

Chapter 7

$\mathcal{ESC}$ E-Service Composer

In this chapter we present the tool we developed and that implements our service composition algorithm. We start by making some considerations, from an implementation perspective, on the PDL encodings of the service composition problem, discussed in previous chapters, and we show that instead of implementing a PDL tableau algorithm, we can resort to Description Logics (DLs). Thus, we present $\mathcal{ALU}$ [8], a simple DL equipped with the constructs we need to encode the composition problem. Then, we show how to encode in an $\mathcal{ALU}$ knowledge base the composition problems studied in Chapters 4 and 5. Finally, we present our prototype tool that implements the composition algorithms.

7.1 Considerations about the PDL Encodings from an Implementation Perspective

To the best of our knowledge, no investigation of effective and efficient tableau algorithms for PDLs is available in the literature and consequently no tool that implements PDL based reasoning services is available. Therefore, from a practical point of view, in order to actually synthesize a Mealy composition, we resort to Description Logics (DLs [8]), exploiting the well known correspondence between PDL formulas and DL knowledge bases [53, 132]. DLs are a family of logic used to represent *static* knowledge, i.e., that can be expressed in terms of classes and relationships between them. We use DLs only for implementation purposes, since DL-based tableau algorithms have been widely studied in the literature and several optimizations for effective and efficient implementations have been devised. Note that for each logic of in the family of PDLs, there exists a corresponding DL and vice-versa, for each DL there exists a corresponding PDL. Specifically, $\mathcal{ALC}_{reg}$ is the DL counterpart of the deterministic variant of PDL, namely DPDL, which has been used to encode the frameworks of Chapters 4 and 5, and $\mathcal{ALCQ}_{reg}$ is the DL counterpart of

the PDL variant equipped with graded modalities, namely PDL_{gm}, that encodes the framework of Chapter 6. Consequently, each service framework of Chapters 4, 5 and 6 can be encoded in a DL knowledge base (see [21, 27]).

The deep investigations on DL Tableaux algorithms has lied the basis for the implementation of highly optimized DL-based reasoning systems [84, 78], that we can use to check the *existence* of service compositions. Actually, such systems cannot handle Kleene star *. However, when in PDL formulas Φ encoding the composition problems * is only used to mimic universal assertions, it is still possible to use such systems to check satisfiability of the DL counterpart of Φ, since they have the ability of handling universal assertions. In other words, they can tackle the service composition problems discussed in Chapters 4 and 5. As far as the framework of Chapter 6, * is not only used to encode universal assertions, but also to capture the semantics of τ actions: tableau algorithms in such a case are quite complex and no result exists on how they can be implemented in an optimal and efficient way. Note that, if τ actions are not present, state-of-the-art DL based reasoning systems can be used to tackle also the framework of Chapter 6.

However, state-of-the-art DL reasoning systems cannot be used to *synthesize* a Mealy composition because they do not return a model. Therefore, we implemented from scratch a tableau algorithm for DLs that builds a model[1] (of the DL knowledge base that encodes the specific composition problem) which is a Mealy composition. The tool that we present in this chapter, for the reason presented above, can handle (for the moment) only the service frameworks that do not involve τ actions.

It is also important to note that in Chapters 4 and 5 we did not use the whole DPDL, but a limited fragment, that we call $DPDL_{lim}$, whose syntax is as follows (its semantics is based on a Kripke structure and can be easily obtained from DPDL semantics). Let $\mathcal{A} = \{a_1, \ldots, a_n\}$ be a set of deterministic atomic actions and $\mathcal{P}$ be a set of atomic propositions. An arbitrary $DPDL_{lim}$ formula ψ has the form:

$$\psi := P \wedge [(a_1 \cup \ldots \cup a_n)^*]\phi$$

where P is an atomic $DPDL_{lim}$ proposition and ϕ is built according to the following rules:

$$\phi \quad \longrightarrow \quad P \mid \neg P \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \langle a_1 \rangle \texttt{true} \mid [a_1]\phi_1$$

Note that a model for the $DPDL_{lim}$ formula ψ is deterministic. Non determinism can be introduced only by the operator $\langle\rangle$. However, we are guaranteed about the functionality of $DPDL_{lim}$ atomic actions (i.e., that no

[1] If one exists.

atomic action a connects a state s_1 (of the Kripke structure) with two different (target) states s_2 and s_3) from the fact that *(i)* $\langle\rangle$ operates only on atomic actions a and *(ii)* it appears only in front of the atomic proposition `true`. Indeed, if a related s_1 with s_2 and s_3, such target states would actually be the same, since s_2 and s_3 are associated with the same atomic proposition `true`.

7.2 The Description Logic $\mathcal{ALU}$

Description Logics (DLs [8]) are logics tailored towards representing knowledge in terms of classes and relationships between classes. Formally they are a well behaved fragment of first order logic (FOL) equipped with decidable reasoning. In DLs, the domain of interest is modeled by means of *concepts* and *relationships*, which denote classes of objects and relations, respectively. Generally speaking, a DL is formed by three basic components:

- a *description language*, which specifies how to construct complex concept and relation expressions (also called simply concepts and relations), by starting from a set of atomic symbols and by applying suitable constructors,
- a *knowledge specification mechanism*, which specifies how to construct a DL knowledge base, in which properties of concepts and relations are asserted, and
- a set of *automatic reasoning procedures* provided by the DL.

The set of allowed constructors characterizes the expressive power of the description language. Various languages have been considered by the DL community, and numerous works investigate the relationship between expressive power and computational complexity of reasoning (see [58] for a survey). The research on these logics has resulted in a number of automated reasoning systems, such as FaCT [84, 86] and Racer [79], which have been successfully tested in various application domains (see e.g., [106, 131, 91]).

Among the DLs, in what follows we focus on $\mathcal{ALU}$ [8], which is a simple DL equipped with the constructs we need to encode the composition problems of Chapters 4 and 5.

Let P and A denote atomic roles (binary relations) and atomic concepts respectively. Syntactically, $\mathcal{ALU}$ concepts, denoted by C, are built by starting

from atomic concepts and atomic roles as follows:

$$
\begin{array}{lll}
C & \longrightarrow & A \mid \text{(atomic concept)} \\
 & & \neg A \mid \text{(atomic negation)} \\
 & & C_1 \sqcap C_2 \mid \text{(intersection (of complex concepts))} \\
 & & C_1 \sqcup C_2 \mid \text{(union (of complex concepts))} \\
 & & \forall P.C_1 \mid \text{(value restriction)} \\
 & & \exists P.\mathtt{true} \text{ ((limited) existential quantification)}
\end{array}
$$

In what follows, we make use of the following standard abbreviations:

$$
\begin{array}{rcl}
\mathtt{true} & \text{for} & A \sqcup \neg A \\
\mathtt{false} & \text{for} & \neg\mathtt{true} \\
A \rightarrow C & \text{for} & \neg A \sqcup C
\end{array}
$$

Note that the left hand side of the symbol $\rightarrow$ is an atomic concept.

Let us comment on the constructs of $\mathcal{ALU}$. Only constructs to form concept expressions are present. Among them, there are limited set operators, namely set complement, intersection, and union that are denoted as negation ($\neg$), conjunction ($\sqcap$), and disjunction ($\sqcup$), respectively. However, while conjunction and disjunction can be applied to arbitrary $\mathcal{ALU}$ concepts, negation can be applied only to atomic concepts. The remaining operators allow to capture quantification relations between the objects in two concepts. Also in this case, the allowed forms of quantification are quite limited, due also to the fact that $\mathcal{ALU}$ deals only with atomic roles. The universal quantification, denoted as $\forall P.C_1$, captures all objects that are related through P only to objects of (the possibly complex) concept C_1. The existential quantification over role P, denoted as $\exists P.\mathtt{true}$, allow only the top concept in the scope of P; intuitively, $\exists P.\mathtt{true}$ represents all objects that are related to some object of the domain through P.

From the syntact rules shown above, it stems that $\mathcal{ALU}$ is a very simple DL. However, it is not the simplest DL, which is $\mathcal{AL}$. The only feature that differentiates the two is the union of complex concepts, which $\mathcal{AL}$ does not contemplate.

An $\mathcal{ALU}$ knowledge base (KB) is constituted by a finite set of *inclusion assertions* of the form

$$C_1 \sqsubseteq C_2$$

where C_1 and C_2 are arbitrary $\mathcal{ALU}$ concepts. Intuitively, the symbol $\sqsubseteq$ denotes a subsumption relation between concepts: the above expression is read as "C_2 subsumes C_1", and it denotes that the concept C_2 is more general than (or as general as) the concept C_1. We also use the abbreviation $C_1 \equiv C_2$ for $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$.

As usual in DLs, the semantics of $\mathcal{ALU}$ is specified through the notion of interpretation. An *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of an $\mathcal{ALU}$ KB $\mathcal{K}$ is constituted by an *interpretation domain* $\Delta^{\mathcal{I}}$ and an *interpretation function* $\cdot^{\mathcal{I}}$ that assigns to each atomic concept A a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to each atomic role P a set $P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, such that the following conditions are satisfied.

$$\begin{array}{rcl}
(\neg A)^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \setminus A^{\mathcal{I}} \\
(C_1 \sqcap C_2)^{\mathcal{I}} & = & C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} \\
(C_1 \sqcup C_2)^{\mathcal{I}} & = & C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}} \\
(\forall P.C)^{\mathcal{I}} & = & \{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in P^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\} \\
(\exists P.\mathtt{true})^{\mathcal{I}} & = & \{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in P^{\mathcal{I}}\}
\end{array}$$

To specify the semantics of an $\mathcal{ALU}$ KB we first define when an interpretation satisfies an assertion as follows. An interpretation $\mathcal{I}$ *satisfies* an inclusion assertion $C_1 \sqsubseteq C_2$ if $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$. In other words, the semantics associated to the subsumption relation is based on subsetting.

An interpretation that satisfies all assertions in a KB $\mathcal{K}$ is called a *model* of $\mathcal{K}$. We say that a KB $\mathcal{K}$ is *satisfiable* if there exists a model of $\mathcal{K}$. A concept C is *satisfiable* in a KB $\mathcal{K}$ if there is a model $\mathcal{I}$ of $\mathcal{K}$ such that $C^{\mathcal{I}}$ is non-empty. An assertion α is *logically implied* by $\mathcal{K}$ if all models of $\mathcal{K}$ satisfy α. It can be shown that all these reasoning services, namely KB satisfiability, concept satisfiability in a KB, and logical implication, are mutually reducible (in polynomial time).

One of the distinguishing features of DLs is that they are designed so as to admit reasoning procedures (i.e., to check KB satisfiability, concept satisfiability in a KB, and logical implication) that are sound and complete with respect to the semantics, and decidable. Such procedures are based on tableaux techniques. Intuitively, tableau based algorithms iteratively compute, by subsequent subsumption (i.e., subset) tests, a taxonomy of classes, making explicit all subsumption relationships among the concepts of the knowledge base. Once such step, called *classification*, is performed, reasoning services can take advantage of it to speed up inferences. Several optimizations of the classification step have been devised and have been implemented in the state-of-the-art DL reasoning systems [8].

In what follows, among the reasoning services $\mathcal{ALU}$ is equipped with, we will consider concept satisfiability in a knowledge base, which is easily shown to be EXPTIME-complete for $\mathcal{ALU}$, since concept satisfiability in a knowledge base is already EXPTIME-hard for $\mathcal{AL}$ and is EXPTIME-complete for $\mathcal{ALC}$[2] which includes $\mathcal{ALU}$ (see [8] for details).

[2]The DL $\mathcal{ALC}$ is equal to $\mathcal{ALU}$ with the difference that it admits negation on arbitrary (instead of atomic) concepts and therefore, it also admits full existential quantification, in the form $\exists R.C$.

Finally, $\mathcal{ALU}$ enjoys three properties that are of particular interest for our aims. The first is the *tree model property*, which says that every model of a concept in a knowledge base can be unwound to a (possibly infinite) tree. The second is the *small model property*, which says that every satisfiable concept in a knowledge base admits a finite model of size at most exponential in the size of the concept and the knowledge base itself. The third is the *single successor property* that says that every model of a concept in a knowledge base can be transformed in such a way that in each object there is at most a unique P-successor for each role P. Moreover such a transformation does not increase the size of the model.

7.2.1 The Correspondence between $DPDL_{lim}$ and $\mathcal{ALU}$

In this section we show the correspondence between a $DPDL_{lim}$ formula and an $\mathcal{ALU}$ knowledge base, which we exploit in the rest of the chapter. Note that the correspondence shown here is specific to our setting. A more general correspondence exists between DLs and PDLs, that can be found in [42].

Such a correspondence is based on the similarity between the interpretation structures of the two logics: at the extensional level, states in $DPDL_{lim}$ correspond to individuals (members of $\Delta^{\mathcal{I}}$) in $\mathcal{ALU}$, whereas state transitions correspond to links between two individuals. At the intensional level, propositions correspond to concepts, and atomic programs correspond to roles. The precise correspondence is realized through a (one-to-one and onto) mapping δ from $DPDL_{lim}$ formulas to $\mathcal{ALU}$ concepts, and from $DPDL_{lim}$ (atomic) programs to $\mathcal{ALU}$ roles. The mapping δ is defined inductively as follows:

$$
\begin{array}{rclrcl}
\delta(P) & = & P & \delta(\neg P) & = & \neg\delta(P) \\
\delta(\phi_1 \wedge \phi_2) & = & \delta(\phi_1) \sqcap \delta(\phi_2) & \delta(\phi_1 \vee \phi_2) & = & \delta(\phi_1) \sqcup \delta(\phi_2) \\
\delta([a]\phi) & = & \forall\delta(a).\delta(\phi) & \delta(\langle a\rangle \mathtt{true}) & = & \exists\delta(a).\mathtt{true} \\
\delta(a) & = & a & & &
\end{array}
$$

where P is an atomic $DPDL_{lim}$ proposition and a an atomic $DPDL_{lim}$ action.

Given the above correspondence, all the $DPDL_{lim}$ constructs that we consider can be naturally mapped into their $\mathcal{ALU}$ analogues. However, DLs are normally used to define a knowledge base, while in PDLs no such notion exists. Consequently, the above correspondence is not sufficient to relate the reasoning problems. In [132] the mapping is therefore extended in such a way that a PDL formula can be viewed as a knowledge base formed by a set of assertions, and reasoning on the DPDL formula can be rephrased in terms of reasoning with respect to such knowledge base. In order to show this mapping, we need the following Theorem [92].

Theorem 45 *[92] Let Γ be a set of DPDL formulas $k_1, \ldots, k_n$, where finitely many atomic programs appear, and let ϕ be a DPDL formula. Then, $\Gamma \models \phi$ if and only if $\models [(a_1 \cup \ldots \cup a_n)^*] \bigwedge_{i=1,\ldots,n} k_i \rightarrow \phi$ or, equivalently, if and only if $[(a_1 \cup \ldots \cup a_n)^*] \bigwedge_{i=1,\ldots,n} k_i \wedge \neg\phi$ is unsatisfiable, where $a_1, \ldots, a_n$ are all the atomic programs appearing in $\Gamma \cup \{\phi\}$.*

This theorem builds upon the basic idea that a set of axioms can be "internalized" into a single concept, i.e., it is possible to build a new formula that expresses all the formulas in Γ [132]. This can be done by exploiting the semantics of union and of reflexive and transitive closure on programs, and because DPDL models are connected. In what follows, we will exploit the theorem in the other direction, i.e., that a certain DPDL formula is equivalent to a finite set of DPDL formulas. Note that in particular, the theorem above holds for $DPDL_{lim}$.

From the above theorem, we can extend the (one-to-one and onto) mapping δ to a (one-to-one and onto) mapping δ' from a $DPDL_{lim}$ formula ψ to an $\mathcal{ALU}$ knowledge base $\mathcal{K}$ as follows. Let $\psi = P \wedge [(a_1 \cup \ldots \cup a_n)^*] \bigwedge_{i=1,\ldots,n} k_i$, where *(i)* $k_1, \ldots, k_n$ are $DPDL_{lim}$ axioms of the form $\phi_1^i \rightarrow \phi_2^i$, with ϕ_1^i, ϕ_2^i (possibly complex) $DPDL_{lim}$ formulas, *(ii)* $a_1, \ldots, a_n$ are the atomic $DPDL_{lim}$ actions appearing in ψ, and *(iii)* P is an atomic $DPDL_{lim}$ formula not containing $a_1, \ldots, a_n$.

$$\delta'(\psi) = P \wedge [(a_1 \cup \ldots \cup a_n)^*]\delta'(k_1) \wedge \ldots \wedge \delta'(k_n) = \{\delta'(P), \delta'(k_1), \ldots, \delta'(k_n)\}$$
$$\delta'(k_i) = \delta'(\phi_1^i \rightarrow \phi_2^i) = \delta(\phi_1^i) \sqsubseteq \delta(\phi_2^i)$$
$$\delta'(P) = \texttt{true} \sqsubseteq \delta(P)$$

From the above construction and by Theorem 45, we have the following results:

Theorem 46 *Let $P \wedge [(a_1 \cup \ldots \cup a_n)^*]\Gamma$ be a $DPDL_{lim}$ formula. Then the $DPDL_{lim}$ conjunct $\Gamma \wedge P$ is satisfiable if and only if the $\mathcal{ALU}$ knowledge base $\delta'(\Gamma)$, $\texttt{true} \sqsubseteq \neg\delta(P)$ is unsatisfiable.*

Before concluding the section, we want to observe that the size of the obtained $\mathcal{ALU}$ knowledge base is polynomial in the size of the DPDL formula. Therefore, from the above construction, satisfiability of DPDL formula can be polynomially reduced to satisfiability of $\mathcal{ALU}$ knowledge base (as well as $\mathcal{ALU}$ concept satisfiability) (and vice-versa).

Notice that, although the correspondence between PDLs and DLs has been exploited to provide reasoning methods for DLs, it has also lead to a number of interesting extensions to PDLs in terms of those constructs that are typical of DLs and have never been considered in PDLs. In particular, there is a tight relation between qualified number restrictions and graded modalities in modal logic [144, 145, 61, 64].

7.3 $\mathcal{ALU}$ Encoding of Service Composition

In this section we discuss how to encode in an $\mathcal{ALU}$ knowledge base the composition problems of Chapter 4 and 5. Note that since the latter is a generalization of the composition problem of the former, we only report the $\mathcal{ALU}$ encoding relative to the framework of Chapter 5. Then, we prove its correctness of such encodings.

7.3.1 Encoding in $\mathcal{ALU}$ the Framework of Chapters 4 and 5

Let E_0 be an underspecified target service, whose (underspecified) external schema is a non-deterministic FSM A_0 and let $\mathcal{C}$ be a community of n services $E_1, \ldots, E_n$ whose external schemas are deterministic FSM $A_1, \ldots, A_n$ respectively. Let Σ be the alphabet of actions of $\mathcal{C}$. We build an $\mathcal{ALU}$ knowledge base $\mathcal{K}_\Phi$ as follows.

As set of atomic concepts $\mathcal{A}$ we have *(i)* one concept s_j for each state s_j of $A_j, j = 0, \ldots, n$, denoting that A_j is in state s_j; *(ii)* concepts $F_j, j = 0, \ldots, n$, denoting that A_j is in a final state; and *(iii)* concepts $\mathsf{moved}_j, j = 1, \ldots, n$, denoting that (component) FSM A_j performed a transition; *(iv)* one concept Init to denote the initial situation. As set of atomic roles $\mathcal{P}$ in $\mathcal{K}_\Phi$ we have the actions in Σ (i.e, $\mathcal{P} = \Sigma$).

The knowledge base $\mathcal{K}_\Phi$ is constituted by the following set of assertions:

- For the external schema $A_0 = (\Sigma, S_0, s_0^0, \delta_0, F_0)$ of the target service we assert:
    - $s \sqsubseteq \neg s'$, for all pairs of states $s, s' \in S_0$; these encode the fact that atomic concepts representing different states are disjoint: specifically, if A_0 is in a state s, it cannot stay in another state s' (different from s).
    - $s \sqsubseteq \bigsqcup_{s' \in \delta_0(s,a)} \exists a.\mathtt{true} \sqcap \forall a.s'$, for each a such that $s' = \delta_0(s, a)$; these encode transitions which are defined for A_0: specifically, if A_0 is in state s, there exists at least one a-transition which is defined from s and all such transitions will make A_0 "move" to state s', or to state s'', or to s''', etc., for each state $s', s'', s''', \ldots$, such as $\delta_0(s, a^{\gg}) = s', \delta_0(s, a^{\gg}) = s'', \delta_0(s, a^{\gg}) = s''', \ldots$
    - $s \sqsubseteq \forall a.\mathtt{false}$, for each $\delta_0(s, a)$; these encode transitions which are not defined for A_0: intuitively, if A_0 is in state s, it cannot make any a-transitions.
    - $F_0 \equiv \bigsqcup_{s \in F_0} s$; this highlights final states of A_0.
- For the external schema $A_j = (\Sigma, S_j, s_j^0, \delta_j, F_j)$ of each service in the community we assert:

- $s \sqsubseteq \neg s'$, for all distinct pairs of states $s, s' \in S_j$.
- $s \sqsubseteq \forall a.(\mathsf{moved}_i \sqcap s' \sqcup \neg\mathsf{moved}_i \sqcap s)$, for each $s' = \delta_0(s, a)$; these encode transitions of A_j, conditioned to the fact that A_j "moves": specifically, if A_j is in state s, for each a-transition that is defined from s, either A_j "moves" and goes to state s', or it does not "move" and remains in state s'.
- $s \sqsubseteq \forall a.(\neg\mathsf{moved}_i \sqcap s)$, for each $\delta_i(s, a)$ which is not defined: intuitively, if A_j cannot do a, and a is performed, then A_j does not "move".
- $F_i \equiv \bigsqcup_{s \in F_i} s$; this highlights final states of A_i.

- Finally, to encode the general structure of models, we assert:
  - $\exists a.\mathtt{true} \sqsubseteq \forall a. \bigsqcup_{i \in [1..n]} \mathsf{moved}_i$, for each action a; these encode that at least one of the community service must move at each step
  - $F_0 \sqsubseteq \bigsqcap_{i \in [1..n]} F_i$; this says that when the target service is in a final state also all services in the community must be in a final state.
  - $\mathsf{Init} \sqsubseteq s_0^0 \sqcap \bigsqcap_{i \in [1..n]} (s_i^0)$; initially all services are in their initial state.

Note that it is easy to extend such encoding to the situation where for each action it is specified the role (initiator or servant) each service plays wrt it.

Example 28 *Figure 7.1 shows the $\mathcal{ALU}$ knowledge base, encoding the composition problem of Section 4.3. Because of the correspondence between DPDL and DLs, a finite model of such $\mathcal{ALU}$ knowledge base is exactly the one shown in Figure 4.7. According to the discussion made in Section 7.2.1, the following observations can be made between the $\mathcal{ALC}$ knowledge base $\mathcal{K}_\Phi$ of Figure 7.1 and the DPDL formula Φ of Section 4.3:*

- *the set of DPDL atomic propositions coincides with the set of $\mathcal{ALU}$ atomic concepts and the set of DPDL atomic roles coincides with the $\mathcal{ALU}$ atomic roles.*

- *the operators of negations, conjunction and disjunction on DPDL propositions correspond to analogous operators on $\mathcal{ALU}$ concepts.*

- *the DPDL modalities $[R]\cdot$ and $\langle R \rangle \cdot$ correspond in $\mathcal{ALU}$ to the universal quantification $\forall R.\cdot$ and to the existential quantification $\exists R.\cdot$, respectively, where R is a DPDL atomic action and, equivalently, an $\mathcal{ALU}$ atomic role.*

- *in Φ we have the master modality $[u]$, representing the reflexive and transitive closure of the union of actions in the alphabet of the community (e.g., $u = (a \cup t \cup l)^*$): in $\mathcal{K}_\Phi$, it does not appear explicitly. Additionally, the axioms in $\mathcal{K}_\Phi$ correspond to conjunctions in Φ.*

- *the DPDL $\rightarrow$ symbol corresponds to the $\mathcal{ALU}$ subsumption symbol $\sqsubseteq$*

Finally, in order to denote the initial situation, in the $\mathcal{ALU}$ knowledge base $\mathcal{K}_\Phi$ we introduce the concept Init*, which has no explicit counterpart in the DPDL formula Φ. Indeed, in Φ we choose to denote the initial situation in terms of the conditions holding in it (i.e., $\bigwedge_{i \in [0..n]} s_i^0$): note, however, that we can as well use the atomic concept* Init*, in which case the DPDL subformula would be* $\mathsf{Init} \rightarrow \bigwedge_{i \in [0..n]} s_i^0$. □

7.3.2 Results on Composition Existence and Synthesis in $\mathcal{ALU}$

In this section we prove the correctness of the $\mathcal{ALU}$ encoding by showing that the same results of Chapter 4 and 5 hold. Such results can be proved in two alternative ways, either by repeating the steps of the corresponding DPDL based Theorem or Lemma, or by using the results of such Theorems and Lemmas and exploiting the correspondence between DPDL and DL. In order to avoid useless repetitions, we follow the second approach.

We start by showing that we can reduce the problem of checking the existence of a composition to the problem of checking the satisfiability of the concept Init in the $\mathcal{ALU}$ knowledge base constructed as in Section 7.3.1.

Theorem 47 *The concept* Init *is satisfiable in the $\mathcal{ALU}$ knowledge base $\mathcal{K}$, constructed as in Section 7.3.1, if and only if there exists a composition of A_0 wrt $A_1, \ldots, A_n$.*

Proof. By Theorem 18 (or Theorem 40 there exists a composition of A_0 wrt $A_1, \ldots, A_n$ if and only if the DPDL formula Φ constructed in Section 4.3 (resp. Section 5.3.1) is satisfiable. By Theorem 46 Φ is satisfiable if and only if the concept Init is satisfiable in the $\mathcal{ALU}$ knowledge base built in Section 7.3.1. □

From the EXPTIME-completeness of concept satisfiability in $\mathcal{ALU}$ and from Theorem 47 we get the following complexity result.

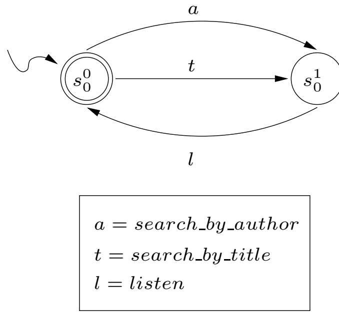
Theorem 48 *Checking the existence of a service composition can be done in EXPTIME.* □

The following theorem shows that we can also build a (possibly non-finite) composition.

Theorem 49 *Any model of an $\mathcal{ALU}$ knowledge base $\mathcal{K}$, built as in Section 7.3.1 is a composition of A_0 wrt $A_1, \ldots, A_n$.*

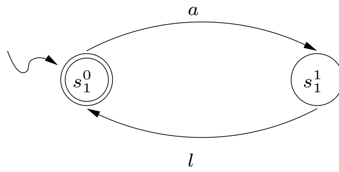
Set $\mathcal{C}$ of atomic concepts is: $\mathcal{C} = \{s_0^0, s_0^1, s_1^0, s_1^1, s_2^0, s_2^1, \mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2, \mathsf{moved}_1, \mathsf{moved}_2, \mathsf{Init}\}$
Set $\mathcal{R}$ of atomic roles is: $\mathcal{A} = \Sigma = \{a, t, l\}$

External schema A_0 of the target service E_0.



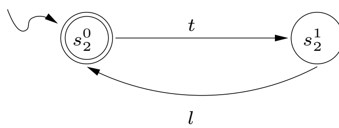
$$
\begin{array}{lcl}
s_0^0 & \sqsubseteq & s_0^1 \\
\\
s_0^0 & \sqsubseteq & \exists a.\mathtt{true} \sqcap \forall a.s_0^1 \\
s_0^0 & \sqsubseteq & \exists t.\mathtt{true} \sqcap \forall t.s_0^1 \\
s_0^0 & \sqsubseteq & \forall l.\mathtt{false} \\
\\
s_0^1 & \sqsubseteq & \forall a.\mathtt{false} \\
s_0^1 & \sqsubseteq & \forall t.\mathtt{false} \\
s_0^1 & \sqsubseteq & \exists l.\mathtt{true} \sqcap \forall l.s_0^0 \\
\\
\mathcal{F}_0 & \equiv & s_0^0
\end{array}
$$

External schema A_1 of component service E_1.



$$
\begin{array}{lcl}
s_1^0 & \sqsubseteq & s_1^1 \\
\\
s_1^0 & \sqsubseteq & \forall a.(\mathsf{moved}_1 \sqcap s_1^1 \sqcup \neg\mathsf{moved}_1 \sqcap s_1^0) \\
s_1^0 & \sqsubseteq & \forall t.(\neg\mathsf{moved}_1 \sqcap s_1^0) \\
s_1^0 & \sqsubseteq & \forall l.(\neg\mathsf{moved}_1 \sqcap s_1^0) \\
\\
s_1^1 & \sqsubseteq & \forall a.(\neg\mathsf{moved}_1 \sqcap s_1^0) \\
s_1^1 & \sqsubseteq & \forall t.(\neg\mathsf{moved}_1 \sqcap s_1^0) \\
s_1^1 & \sqsubseteq & \forall l.(\mathsf{moved}_1 \sqcap s_1^0 \sqcup \neg\mathsf{moved}_1 \sqcap s_1^1) \\
\\
\mathcal{F}_1 & \equiv & s_1^0
\end{array}
$$

External schema A_2 of component service E_2.



$$
\begin{array}{lcl}
s_2^0 & \sqsubseteq & s_1^1 \\
\\
s_2^0 & \sqsubseteq & \forall a.(\neg\mathsf{moved}_2 \sqcap s_2^0) \\
s_2^0 & \sqsubseteq & \forall t.(\mathsf{moved}_2 \sqcap s_2^1 \sqcup \neg\mathsf{moved}_2 \sqcap s_2^0) \\
s_2^0 & \sqsubseteq & \forall l.(\neg\mathsf{moved}_2 \sqcap s_2^0) \\
\\
s_2^1 & \sqsubseteq & \forall a.(\neg\mathsf{moved}_2 \sqcap s_2^0) \\
s_2^1 & \sqsubseteq & \forall t.(\neg\mathsf{moved}_2 \sqcap s_2^0) \\
s_2^1 & \sqsubseteq & \forall l.(\mathsf{moved}_2 \sqcap s_1^0 \sqcup \neg\mathsf{moved}_2 \sqcap s_2^1) \\
\\
\mathcal{F}_2 & \equiv & s_2^0
\end{array}
$$

Domain independent conditions:

$$
\begin{array}{ll}
\mathtt{true} \sqsubseteq \exists a.\mathtt{true} \sqcap \forall a.\mathsf{moved}_1 \sqcup \mathsf{moved}_2 & \mathcal{F}_0 \sqsubseteq \mathcal{F}_1 \sqcap \mathcal{F}_2 \\
\mathtt{true} \sqsubseteq \exists t.\mathtt{true} \sqcap \forall a.\mathsf{moved}_1 \sqcup \mathsf{moved}_2 & \\
\mathtt{true} \sqsubseteq \exists l.\mathtt{true} \sqcap \forall a.\mathsf{moved}_1 \sqcup \mathsf{moved}_2 & \mathsf{Init} \sqsubseteq s_0^0 \sqcap s_1^0 \sqcap s_2^0
\end{array}
$$

Figure 7.1: $\mathcal{ALU}$ encoding of the composition problem of composition problem of Chapter 4

Proof. From Lemma 17 (or Lemma 39) and Theorem 46. □

However, we are interested in compositions which are finite state machines, therefore, we define a Mealy composition starting from a finite model of the $\mathcal{ALU}$ knowledge base, as follows.

Definition 50 (Mealy Composition) Given a finite model $\mathcal{I}_f = (\Delta^{\mathcal{I}_f}, \cdot^{\mathcal{I}_f})$, we define *Mealy composition* an MFSM $A_c = (\Sigma, 2^{[n]}, S_c, s_c^0, \delta_c, \omega_c, F_c,)$, built as follows:

- $S_c = \Delta^{\mathcal{I}_f}$;
- $s_c^0 = Init^{\mathcal{I}_f}$;
- $s' = \delta_c(s, a)$ iff $(s, s') \in a^{\mathcal{I}_f}$;
- $I = \omega_c(s, a)$ iff $(s, s') \in a^{\mathcal{I}_f}$ and for all $i \in I$, $s' \in moved_i^{\mathcal{I}_f}$ and for all $j \notin I$, $s' \notin moved_j^{\mathcal{I}_f}$;
- $F_c = \mathcal{F}_0^{\mathcal{I}_f}$.

□

By the small model property of $\mathcal{ALU}$, we can prove the existence of a Mealy composition.

Theorem 51 *If there exists a composition of A_0 wrt $A_1, \ldots, A_n$ then there exists one that is a Mealy composition of size at most exponential in the size of $A_0, A_1, \ldots, A_n$.*

Proof. From Theorems 47 and 46. □

Finally, the following theorem says that we can also build a Mealy composition, starting from a finite model of the $\mathcal{ALU}$ knowledge base, and applying the construction of Definition 50.

Theorem 52 *Any finite model of the $\mathcal{ALU}$ knowledge base built in Section 7.3.1 is a Mealy composition of A_0 wrt $A_1, \ldots, A_n$.*

Proof. From Theorem 49 and Theorem 46. □

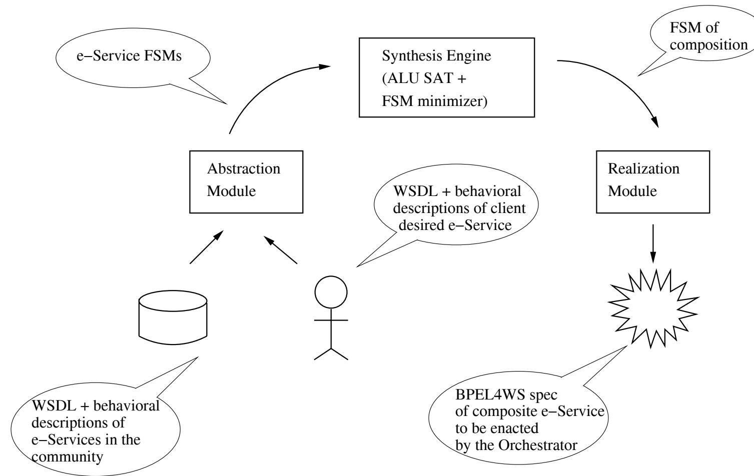


Figure 7.2: The Service Composition Architecture

7.4 The Service Composition Tool $\mathcal{ESC}$

In this section we discuss the prototype tool $\mathcal{ESC}$ that we developed to compute automatic service composition in our framework.

Figure 7.2 shows the high level architecture for $\mathcal{ESC}$. Each service is represented in terms of both its static interface, through a WSDL document, and its behavioral description[3], which can be expressed in any language that allows to express a finite state machine (e.g., Web Service Conversation Language [77], Web Service Transition Language [30], BPEL4WS [4], etc.). We recall that in our framework the focus is on actions that a service can execute; such actions can be seen as the abstractions of the effective input/output messages and operations offered by the service. As an example, Figure 7.3 shows the WSDL interface of service E_0 whose behavior is represented in Figure 7.1.

We start from a repository of services, which implements the community of services, and which can be seen, therefore, as an advanced version of UDDI[142]. The client specifies his target service in terms of a WSDL document and of its behavioral description, again expressed using one of the language mentioned before[4]. Both the services in the repository and the target service are then abstracted into the corresponding FSM (`Abstraction Module`). The `Synthesis Engine` is the core module of $\mathcal{ESC}$. It takes in input such FSMs, produces the $\mathcal{ALU}$ knowledge base $\mathcal{K}_\Phi$, (possibly) builds a model and produces in output the MFSM of the composite service, where

[3]Note that such behavioral description of services specifies the external schema.

[4]The behavioral description of both the client specification and the services in the repository are expressed in the same language.

```
<definitions ...
   xmlns:y="http://new.thiswebservice.namespace"
   targetNamespace="http://new.thiswebservice.namespace">

   <!-- Types -->
   <types>
      <element name="ListOfSong_Type">
         <complexType>
            <sequence>
               <element minOccurs="1"
                        maxOccurs="unbound"
                        name="SongTitle"
                        type="xs:string"/>
            </sequence>
         </complexType>
      </element>
   </types>

   <!-- Messages -->
   <message name="search_by_title_request">
      <part name="containedInTitle" type="xs:string"/>
   </message>
   <message name="search_by_title_response">
      <part name="matchingSongs" xsi:type="ListOfSong_Type"/>
   </message>
   <message name="search_by_author_request">
      <part name="authorName" type="xs:string"/>
   </message>
   <message name="search_by_author_response">
      <part name="matchingSongs" xsi:type="ListOfSong_Type"/>
   </message>
   <message name="listen_request">
      <part name="selectedSong" type="xs:string"/>
   </message>
   <message name="listen_response">
      <part name="MP3fileURL" type="xs:string"/>
   </message>

   <!-- Service and Operations -->
   <portType name="MP3CompositeServiceType">
      <operation name="search_by_title">
         <input message="y:search_by_title_request"/>
         <output message="y:search_by_title_response"/>
      </operation>
      <operation name="search_by_author">
         <input message="y:search_by_author_request"/>
         <output message="y:search_by_author_response"/>
      </operation>
      <operation name="listen">
         <input message="y:listen_request"/>
         <output message="y:listen_response"/>
      </operation>
   </portType>

</definitions>
```

Figure 7.3: WSDL specification of service E_0 whose external schema A_0 is represented in Figure 7.1.

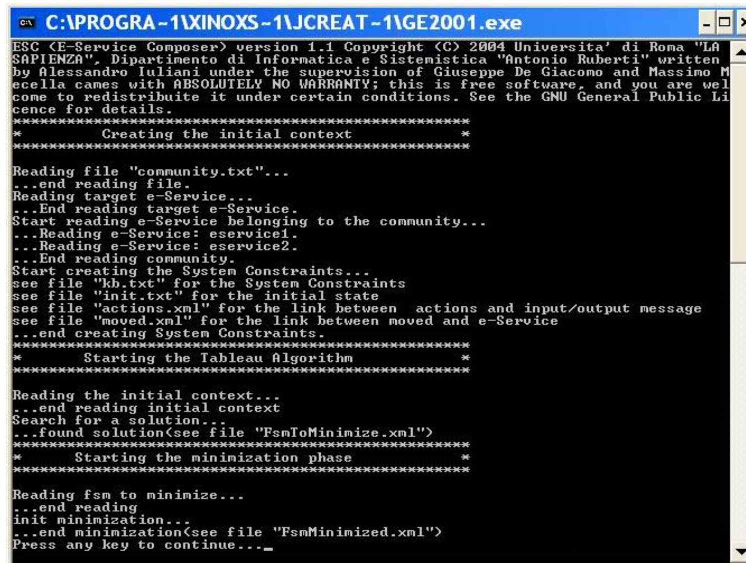


Figure 7.4: A screenshot of $\mathcal{ESC}$ running on the example of Figure 7.1

each action is annotated with (the identifier of) the component service(s) that executes it. Finally, such abstract version of the composite service is realized into a BPEL4WS specification[5] (`Realization Module`), that can be executed by an orchestration engine, i.e., a software module that suitably coordinates the execution of the component services participating to the composition [69].

Figure 7.4 shows a screenshot of the execution of $\mathcal{ESC}$ over the $\mathcal{ALU}$ encoding of Figure 7.1.

We tested our tool on several examples, involving communities containing up to 10 services, each one having roughly 10-20 states: $\mathcal{ESC}$ performs quite nicely, considering that the current release does not implement any relevant optimization.

7.4.1 The Abstraction Module

The implementation of the `Abstraction Module` depends on which language is used to represent the behavioral description of services. In our prototype we use the Web Service Transition Language (WSTL [30]). WSTL is an XML-based description language able to represent the exported behavior of services. It does not address the implementation that actually drives the interactions; rather, it captures what is observable from the point of view of the client. Although simple, it enables the modeling of complex behaviors, such as loops

[5]It represents the internal schema for the target service.

```
<?xml version="1.0" encoding="UTF-8"?>
<Conversation name="target">
        <Transition source="s00IF" target="s01">
                <InputMessage>search_by_author_request</InputMessage>
                <OutputMessage>search_by_author_response</OutputMessage>
        </Transition>
        <Transition source="s00IF" target="s01">
               <InputMessage>search_by_title_request</InputMessage>
               <OutputMessage>search_by_title_response</OutputMessage>
        </Transition>
        <Transition source="s01" target="s00IF">
                <InputMessage>listen_request</InputMessage>
                <OutputMessage>listen_response</OutputMessage>
        </Transition>
</Conversation>
```

Figure 7.5: WSTL specification of FSM A_0 shown in Figure 7.1

and exceptions. WSTL integrates well with existing standards, in particular it is complementary to WSDL since it extends the static interface of WSDL with elements which describe the correct sequence of the exchanged messages. An interesting feature of WSTL is that it has a clear and formal semantics based on finite state transition systems: in particular, the WSTL specification of a service can be easily obtained from the conceptual specification of the service behavior, expressed as FSM, by following the methodology presented in [30]. A detailed description of WSTL goes beyond the scope of this thesis. However, for sake of completeness we show in Figure 7.5 a WSTL specification of the target service A_0 shown in Figure 7.1.

Here, we will not go over the specification, we only make the following observations. State s_0^0 is both initial and final in A_0, therefore in the WSTL specification we adorn its name with I and F, to denote such information. Each transition, which connect a source state to a target state, abstractly represents the input and output messages of an action: `InputMessage` denotes the functionality that the client requests to a service and `OutputMessage` represents the functionality provided by the service. For example, when the client invokes the input functionality `search_by_author_request`, the service "answers" with the output functionality `search_by_author_response`. Note that also here we are focusing on functionalities, therefore, we are not concerned on whether the returned list of songs is empty or it contains at least one element: in all cases the output functionality that service provides when its client requests to search an mp3 by specifying its author is followed by a request to provide the functionality to listen to a song (`listen_request`). Note that the input and output messages of the WSTL file of Figure 7.5 are those listed in the WSDL file of Figure 7.3.

7.4.2 Implementation of the Synthesis Engine Module

The various functionalities of the `Synthesis Engine` are implemented into three Java sub-modules.

- The `FSM2ALU Translator` module takes in input the FSMs produced by the `Abstraction Module`, and translates them into an $\mathcal{ALU}$ knowledge base, following the encoding presented in Section 7.3.1.
- The $\mathcal{ALU}$ `Tableau Algorithm` module implements the standard tableau algorithm for $\mathcal{ALU}$ (cf., Buchheit et al., 1993 [39]). It takes in input the $\mathcal{ALU}$ knowledge base and checks the satisfiability of the Init concept, or, equivalently, it verifies if a composition exists. If this is the case, it returns a model of the knowledge base, which is a finite state machine. Otherwise, it returns the information about unsatisfiability of the knowledge base, i.e., the non-existence of a composition.
- The `FSM Minimizer` module minimizes the model, since it may contain states which are unreachable or unnecessary. Classical, standard minimization techniques can be used, in particular, we implemented the *Implication Chart Method* [125]. The minimized FSM is then converted into a Mealy FSM, where each action is annotated with the service in the repository that executes it (and possibly with the role the service participates in).

Since these three modules are in fact independent, they are wrapped into an additional module, the `Composer Module`, which also provides the external interface.

7.4.3 Implementation of the Realization Module

The `Realization Module`, whose development is currently ongoing, is in charge of producing an executable BPEL4WS file starting from the automatically synthesized MFSM. In the following, we outline the intuitions that are driving our design and development (based on results in [10, 24]):

- Transitions are mapped first, thus deriving transition skeletons, then states are mapped, thus deriving state skeletons, and finally the BPEL4WS file is obtained, by connecting state skeletons on the basis of the MFSM; in such a way the obtained BPEL4WS specification has a structure similar to the one shown in Figure 7.6, i.e., with a `<flow>` operation wrapping all the state skeletons, connected among them by `<link>`s.

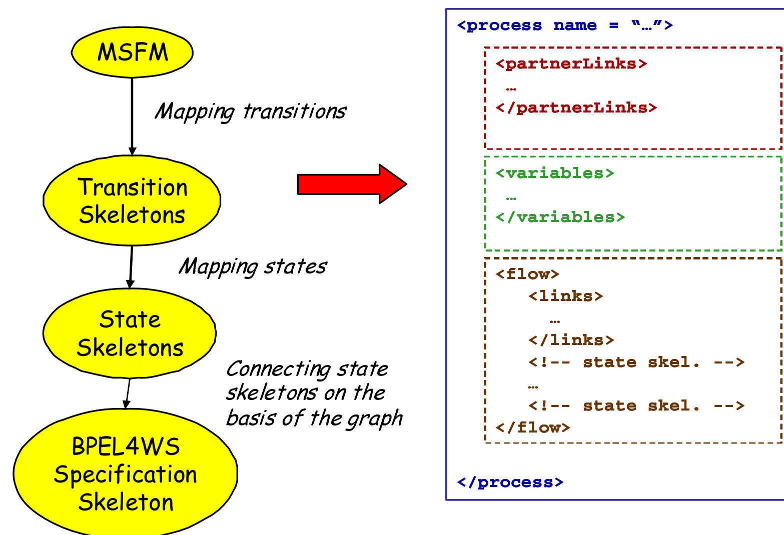


Figure 7.6: Methods for deriving the BPEL4WS file and its structure [10]

- Each transition corresponds to a BPEL4WS pattern (i.e., transition skeleton) consisting of *(i)* an `<onMessage>` operation (in order to wait for the input from the client of the composite service), *(ii)* followed by the invocation to the appropriate component service, and then *(iii)* a final operation for returning the result to the client. Of course both before the component service invocation and before returning the result, messages should be copied forth and back in appropriate variables.

- All the transitions originating from the same state are collected in a `<pick>` operation, having as many `<onMessage>` clauses as transitions originating from the state; this is the state skeleton.

- The above steps for transition and state skeletons work for request/reply interactions; simple modifications are needed for notification/response, one-way and notification-only interactions, that can imply a proactive behaviour of the composite service, possibly guarded by `<onAlarm>` blocks. Figure 7.7 shows the structure of the skeletons.

- Finally, the BPEL4WS file is built visiting the MFSM in depth, starting from the initial state and applying the previous rules. Specifically, all the `<pick>` blocks are enclosed in a surrounding `<flow>`; the dependencies are modeled as `<link>`s: `<link>`s are controlled by specific variables `Si-to-Sj` that are set to `TRUE` iff the transition $S_i \rightarrow S_j$ is executed; each state skeleton has many outgoing `<link>`s as states connected in output, each going to the appropriate `<pick>` block.

```
<onMessage ... >
   <sequence>
      <assign>
         <copy>
            <from variable="input" ... />
            <to variable="transitionData" ... />
         </copy>
      </assign>
      < !-- invocation of the component service -->
      <assign>
         <copy>
            <from variable="transitionData" ... />
            <to variable="output" ... />
         </copy>
      </assign>
      <reply ... />
      </sequence>
</onMessage>
```

(a) Transition skeleton

```
<!-- N transition from state Si -->
<pick name = "Si">
   <!-- transition #1 -->
      <onMessage ...>
      <!-- transition skeleton -->
      </onMessage>
      ...
      <!-- transition #N -->
      <onMessage ...>
      <!-- transition skeleton -->
      </onMessage>
</pick>
```

(b) State skeleton

Figure 7.7: BPEL4WS code skeletons for transitions and states

- The previous step works for acyclic state machines. In the case of a state machine with cycles, the following intuition can be applied: *(i)* identify all the cycles; *(ii)* for each cycle enclose the involved state skeletons inside a `<while>` block controlled by a condition `!exit`, where `exit` is a variable defined ad hoc and it is set to `FALSE` by any transition that "goes out" of the cycle; *(iii)* connect the overall `<while>` block to other state skeletons by appropriate `<link>`s.

  There are some interesting special cases: *(i)* a state S with self-transitions can be represented as a `<pick>` block enclosed in a `<while>` block controlled by a condition `(Vs)` (the variable `Vs` is set to `FALSE` by other non self-transitions); *(ii)* cycles starting from the initial state should not be considered, as they can be represented as the start of a new instance of the BPEL4WS process.

By remarking the fact that the Realization Module is still in the development phase, we present in Figure 7.8 the BPEL4WS pseudo code for the MFSM of the running example.

We foresee the validation of our approach and an engineered implementation of the tool in the context of the MAIS[6] project, which aims at studying and applying the SOC paradigm to Multichannel Adaptive Information Systems.

[6] `http://black.elet.polimi.it/mais/index.php`

```
<process suppressJoinFailure = "no">

   <partnerLinks>
   ...
   <!-- Sono definiti i partner link per il servizio composto
        (MP3CompositeServiceType), per il servizio componente di
        tipo 1 (MP3ServiceType1) e per il servizio componente di
        tipo 2 (MP3ServiceType2)
   -->
   </partnerLinks>

   <variables>
      <variable name="input" messageType="listen_request"/>
      <variable name="output messageType="listen_response"/>
      <variable name="dataIn" messageType="listen_request"/>
      <variable name="dataOut" messageType="listen_response"/>
      ... ...
   </variables>

   <flow>
      <links>
         <link name="start-to-1"/>
         <link name="start-to-2"/>
      </links>

      <pick createInstance = "yes">
      <!-- A new process instance is created in the initial state.
           This resolve the presence of the cycles without the need
           of an enclosing <while>
      -->

         <onMessage="a" ...>
            <sequence>
               <assign><copy>...</copy></assign>
               ...
               <!-- The "a" transition skeleton should
                    set variables: start-to-1 = TRUE
                                   start-to-2 = FALSE
               -->
               <assign><copy>...</copy></assign>
               <reply ... />
            </sequence>
         </onMessage>
         <onMessage="t" ...>
            <sequence>
               <assign><copy>...</copy></assign>
               ...
               <!-- The "t" transition skeleton should
                    set variables: start-to-1 = FALSE
                                   start-to-2 = TRUE
               -->
               <assign><copy>...</copy></assign>
               <reply ... />
            </sequence>
         </onMessage>
         <source linkName="start-to-1"
          transitionCondition = "bpws:getVariableData(start-to-1) = TRUE" />
         <source linkName="start-to-2"
          transitionCondition = "bpws:getVariableData(start-to-2) = TRUE" />
      </pick>

      <pick>
         <onMessage partnerLink="client"
                    portType="MP3CompositeServiceType"
                    operation="listen"
                    variable="input">
            <sequence>
               <assign>
                  <copy>
                     <from variable="input" part="selectedSong"/>
                     <to variable="dataIn" part="selectedSong"/>
                  </copy>
               </assign>
               <invoke partnerLink="service"
                       portType="MP3ServiceType1"
                       operation="listen"
                       inputVariable="dataIn"
                       inputVariable="dataOut" />
               <assign>
                  <copy>
                     <from variable="dataOut" part="MP3FileURL"/>
                     <to variable="output" part="MP3FileURL"/>
                  </copy>
               </assign>
               <reply name="replyOutput"
                      partnerLink="client"
                      portType="MP3CompositeServiceType"
                      operation="listen"
                      variable="output"/>
            </sequence>
         </onMessage>
         <target linkName="start-to-1"/>
      </pick>

      <pick>
         <onMessage ... >
            <sequence>
               <assign><copy>...</copy></assign>
               ...
               <assign><copy>...</copy></assign>
               <reply ... />
            </sequence>
         </onMessage>
         <target linkName="start-to-2"/>
      </pick>

   </flow>
</process>
```

Figure 7.8: BPEL4WS pseudo-code for the MFSM shown in Figure 4.8. The detail of a transition skeleton is shown only for one operation.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis we have defined a formal and comprehensive framework for the characterization and the theoretical investigation of the problem of service composition. In particular, the main contribution wrt research on Service Oriented Computing is in tackling *simultaneously* the following issues:

- presenting a formal framework where services are characterized in terms of their behavioral descriptions and the problem of service composition is precisely defined;
- providing techniques for automatically computing service composition in the case where the behavioral description of services is expressed as finite state machines, and providing a computational complexity characterization of the algorithms for automatic composition;
- presenting $\mathcal{ESC}$, an open source prototype tool that implements our techniques for automatically synthesizing a composition.

This thesis constitutes a first step at least towards *(i)* the achievement of an agreed upon comprehension of what a service is, from an abstract perspective; *(ii)* the definition of a general and common framework that contextualizes services and service composition, and *(iii)* a consolidated formal definition of service composition.

The ultimate goal of Service Oriented Computing is to enable organizations to seamlessly compose business processes and dynamically integrate them with the partners processes, exploiting network technology. Thus, the results on automatic service composition developed in this thesis form a conceptual basis to define how internal business processes can be dynamically integrated with those of other organizations in order to create new value-added services.

8.2 Future Work

The research areas related to services, and in particular, service composition are quite recent and there are many open problems to study and resolve. This thesis is among the first ones that tackle the issue of automatic composition from an abstract perspective and depict a formal framework for the characterization of such a problem. The work done in this thesis can be therefore taken as a starting point for a deeper investigation of several aspects of service composition. In what follows, we highlight some possible future directions. Throughout the thesis we already mentioned some of them, such as *(i)* allowing the service community to export a *partial* behavioral description, and studying the problem of service composition in such a framework (see Section 5.4); *(ii)* allowing several instances of a service to be simultaneously active and possibly interact one with the other (see Section 3.7); *(iii)* extending our $\mathcal{ESC}$ tool to deal also with client specifications containing τ actions (see Chapter 7); *(iv)* dynamic community and on-the-fly re-configuration of composite services (see Section 3.7); *(v)* introduce data in such a way that the problem of service composition is polynomial in the data (see Section 3.3.2); (*vi*) extending our framework towards an agent based perspective, by blurring the distinction between clients and component services (see Section 6.5).

In the next subsection, we discuss additional extensions that we are currently developing.

8.2.1 Simulation

In this thesis we have solved the problem of service composition (where services are represented as FSMs) by reducing it to the problem of satisfiability in PDL/DL. In other words, the algorithm we have provided does not explicitly show how a composite service is built, since its construction can be seen as a "side-effect" of the tableau algorithms that check satisfiability. Therefore, it is interesting to devise techniques to explicitly build the MFSM representing the composite service. This also allows us to reach a hardness result for the problem of service composition synthesis, that we conjecture to be EXPTIME-hard.

Our basic idea is to refer to notions and results in the program synthesis and verification field, focusing, in particular, to the *simulation* problem between two concurrent transition systems. We recall that given two transition systems S and S', S simulates S' (written $S \preceq S'$) if, intuitively, S' presents more behavior than S', and includes the behavior of S' (see e.g. [82]). The simulation problem is defined as, given S and S', to check whether $S \preceq S'$. Thus, it is easy to see [20] that a composition exists if and only if $S \preceq S'$, if S is the FSM (external schema of the) target service, and S' is a "suit-

ably" built cartesian product of (the FSM external schema of) all component services. In [82], the authors prove that the simulation problem for concurrent transition systems is EXPTIME-complete. In particular, they show the hardness by providing an explicit construction: given S' and an alternating Turing machine T of space complexity $s(n)$, they prove that it is possible to synthesize, using a logarithmic amount of space, a concurrent transition system S of size $O(s(n))$ such that $S \preceq S'$ iff T accepts the empty tape. In the program synthesis and verification field, S' represents the specification of a desired program, S represent (a model of) the program that one would like to build. However, we cannot directly use this results because in our service framework the target service S is given and S' should be built!

Currently, we are studying how to solve the opposite problem, i.e., given S' how to synthesize S such that $S \preceq S'$.

8.2.2 FLOWS

FLOWS[1] (First-Order Logic Ontology for Web Service [29]) is a proposal of a language for representing services and in particular their process model (i.e., their exported behavioral description). It has been developed as part of SWSL (Semantic Web Services Language)[2], which is a working group of the Semantic Web Service Initiative. The main purpose of FLOWS is to *overcome the limitations intrinsic in OWL-S*, which is a description logic based language with a well defined semantics, but which is not expressive enough to capture all and only the intended interpretations of the process model. Therefore, in order to resolve ambiguities, process models expressed in OWL-S need to be interpreted by a human, thus not achieving completely the Service Oriented vision, according to which the semantics of a service should be computer-interpretable. Additionally, FLOWS *aims at providing a uniform view of the various frameworks where services are characterized in terms of their process*, expressed in PDL/DL [31], as automata [40], as Petri Nets [118], in Situation Calculus [107].

FLOWS is based on the Process Specification Language (PSL [75, 76]), which is a first order logic ontology explicitly designed for allowing (correct) interoperability among heterogeneous software applications, that exchange information as first-order sentences. Recently, PSL has become an International Standard (ISO 18629). PSL has many interesting features, that make it suitable for modeling service behavior. Indeed, PSL can model both "black box" processes, i.e., activities, and "white box" processes, i.e., complex activities, and allows for explicit quantification over complex activities. In particular, the latter aspect, not shared by several other formalisms, makes it possible to

[1] This is a joint work with Michael Gruninger, Rick Hull and Sheila McIlraith.

[2] `http://www.daml.org/services/swsl/`

express in an explicit manner a broad variety of properties and constraints on composite activities. PSL is constituted by a (finite) set of first order logic theories defining concepts (i.e., relations and functions) needed to formally describe a process and its properties (e.g., temporal constraints, activity occurrences, etc.): they are defined starting from primitive ones, whose meaning is assumed in the ontology. Additionally, it is extensible, in the sense that other theories can be created in order to define other concepts. The fact that PSL is based on FOL, on one side equippes it with a well understood model-theoretic semantics, thus overcoming the OWL-S expressivity problems, but on the other hand it makes it semi-decidable. However, a solution to this drawback consists in identifying several decidable subsets of PSL, which can be exploited in performing automated tasks, such as service composition or verification.

8.2.3 The COLOMBO Framework

COLOMBO[3] is a rich framework for services, that builds on the results presented in this thesis and extends it essentially with data and message types. Data may be internal to services or shared by the service community, therefore we introduce, respectively, the notions of local store and world schema to represent them. In general the data contained in the local store is not a subset of the data in the world schema, since the former may refer to variables that are internal to the services and therefore are not exported. In addition to the world schema, a service community is also characterized by a set of atomic actions that modify the world schema, a finite set of services, and a set of message types, denoting the alphabet of the community. Each service is characterized in terms of its behavior expressed as guarded automata, defined as usual over the alphabet of the community. Differently from the framework presented in this thesis, each service interacts with the other services in the community and with the client in terms of messages and actions. Therefore, the primitive actions it defines are essentially used to send a message (type), receive a message (type), but also to perform an operation, by specifying input parameters, and to assign some value to a location in the local store. The transition relation defines the set of successor states given a state, a guard and a primitive actions. A guard is a FOL formula in the general case, it is a propositional formula in a restricted, decidable framework. Finally, between the various services in the community, a set of (bidirectional) channels can be defined: in general, we envision a queue based communication topology, where the various queues are not part of the services, but are seen as part of the community.

[3]This is a joint work with Giuseppe De Giacomo, Diego Calvanese, Rick Hull, Maurizio Lenzerini and Massimo Mecella.

The problem of service composition, in such a framework, aims at building a new service that realizes a client request, by coordinating the available services. Several solutions to this problems may be found, depending also on the formalism used to express the client request.

8.2.4 Modeling the Client

The Service Oriented Computing paradigm aims at supporting the automatic interoperation and collaboration of available services. Therefore, the work done in the service literature (including the one reported in this thesis) focus on such issues. However, most of the services will be exploited and in particular, automatically composed, so that a human user reaches his goal.

To the best of our knowledge few works in the literature models the client requests and no work at all models a (human) client and his/her interactions wrt a service. Instead, it would be very interesting to study how human-machine interaction models can be applied in the context of services, so that also the degree of service *usability* reaches high levels. According to the definition in [55], a user is "whoever is trying to get the job done using the technology", i.e., by interacting with a system.

Appendix A

Characterizing Services and Service Composition in Situation Calculus

In this chapter we discuss the use of reasoning about actions formalisms, and in particular of Situation Calculus, for representing services and synthesizing compositions. Then, we show the use of description logics reasoning procedures for getting effective algorithms and complexity results to perform composition synthesis. This approach is alternative and equivalent to the one presented in Chapter 4.

A.1 Preliminaries on Situation Calculus

In this section, we present the basic notions of the Situation Calculus, that we will use in the remaining of the chapter. We assume the reader to be familiar with it and refer him/her to [124] for a more detailed insight.

Situation Calculus is a first-order language (with some second order features) for representing dynamic domains. In this formalism, world changes happen under the effects of named *actions*. The history of world changes, denoted simply by a sequence of actions, is called *situation* and it is represented by a first-order term. The *initial situation*, denoted by the special constant S_0, represents the particular situation in which no actions have occurred yet. The situations are arranged in a tree based structure called *situation tree*, whose root is the initial situation. The distinguished binary function $do(a, s)$ denotes the successor situation to s resulting from performing the action a. In general, actions may be parameterized. For example, $pickup(x)$ denotes the action of picking up an object x and $do(pickup(block), s)$ denotes the situation resulting from picking up the block in situation s. Propositions whose truth

values vary from situation to situation are called *fluents*, and are denoted by predicate symbols taking a situation term as their last argument. For example, the fluent $holding(x, s)$ is true when an agent is holding object x in situation s.

Each action in the domain is specified by providing certain types of axioms.

First, it is necessary to state the conditions that must hold so that executing an action is physically possible. These are specified by the the *action precondition axioms*, using the special predicate $Poss(a, s)$. For example,

$$\forall s.Poss(pickup(x), s) \equiv \forall y.\neg holding(y, s) \wedge clear_on_top(x, s)$$

says that the agent can pick up object x if and only if the agent is not holding any other object and x is clear on top, i.e., no other object is on top of x.

Second, it must be stated how the action affects the world. This is done by specifying *effect axioms*, which represent "causal laws" of the application domain. For example,

$$clear_on_top(x, s) \supset holding(x, do(pickup(x, s)))$$

says that picking up an object x causes it to being hold in the hand, provided that no other object was on top of x. However, effects axioms are not sufficient to reason about change, since it is necessary also to provide the *frame axioms*, stating which fluents remain unchanged by executing the action. Of course, the number of the frame axioms is very large, in general, it is of the order of $2\times\mathcal{A}\times\mathcal{F}$, where $\mathcal{A}$ is the number of actions and $\mathcal{F}$ is the number of fluents. Therefore, axiomatizing a domain can be quite complex and theorem proving can become highly inefficient. This problem is called the *frame problem.* In [124] a way to deal with the the frame problem is presented, based on the idea of collecting all the effect axioms about a given fluent and making a completeness assumption, i.e., that they specify all and only the ways that the value of the fluent may change. Thus, performing a syntactic transformation, one obtains a *successor state axiom* for the fluent:

$$\begin{array}{rcl} holding(x, do(a, s)) & \equiv & (a = pickup(x) \wedge clear_on_top(x, s)) \vee \\ & & (holding(x) \wedge a \neq drop(x)) \end{array}$$

This says that an agent holds object x in situation s if and only if x is clear on top and the agent makes the action of picking x up, or the agent was already holding x and did not drop it. Note that this approach yields a solution to the frame problem, i.e., a parsimonious representation for the effects of action.

Within this language, we can formulate domain theories that describe how the world changes as the result of the available actions. One possibility are Reiter's Basic Action Theories, which have the following form [124]:

- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action a. They have the form $\forall s.Poss(a, s) \equiv \Psi_a(s)$, where $\Psi_a(s)$ is a Situation Calculus formula (uniform in s) with s as the only free variable and in which *Poss* does not appear.
- Successor-state axioms, one for each fluent F. They have the form $\forall a, s.F(do(a, s)) \equiv \Phi_F(a, s)$, where $\Phi_F(a, s)$ is a Situation Calculus formula (uniform in s) with a and s as the only free variables.
- Unique names axioms for the primitive actions plus some foundational, domain independent axioms.

A.2 Service Composition in Situation Calculus

In Chapter 3 we have characterized service behavior and composition in general terms by means of execution trees. This abstract view needs to be refined in order to get a finite representation of services that can be concretely manipulated.[1] In Chapter 4 we addressed services whose execution trees (both external and internal) have a finite representation as finite state machines. Here, we also deal with services whose execution trees have a compact representation, and we propose an alternative but equivalent approach based on formalisms developed for Reasoning about Actions to represent services. This allows us to use logical reasoning, in particular, satisfiability, to characterize the problem of service composition, in an analogous way as done in Section 4.3. There are many possible action languages that can be used for representing services (including some tightly related to DL [53, 103]). Here we focus on Reiter's Situation Calculus Basic Action Theories [124], which are widely known and allow us to concentrate on the aspects specific to our problem. Since we aim at actually computing the compositions we will deal with the propositional variant of the Situation Calculus (in which fluents are propositions). As in Section 4.3 we also assume that for each service there is only a fixed finite number of active instances, and, in fact, wlog, we assume that there is only one, so that we can omit the term "instance" when referring to a service. Within this setting, in the next section, we show how to solve the composition problem. Instead, how to deal with an unbounded number of instances remains open for future work.

In order to characterize composition in this setting, we first show how a Basic Action Theory can represent the external execution tree of a service. We represent the external schema of a service E as a Basic Action Theory

[1] Obviously, not all execution trees can be represented in a finite way.

Γ, where each action is represented by a Situation Calculus action. The set of fluents of Γ is constituted by *(i)* a special fluent *Final*, denoting that the service execution can stop in that situation; and *(ii)* one fluent F_a for each action a, with the following meaning: F_a is true in situation s if and only if it is possible to execute a in s. Also, Γ fully specifies the value of each fluent in the initial situation S_0. Technically, this means that we have complete information on the initial situation, and, because of the action precondition and successor-state axioms, we have complete information in every situation.

Observe that the fluents used in Γ have a meaning only wrt to the service community, since they are not attached in any way to the actual service instance the client interacts with. In contrast, actions represent interactions meaningful both to the client and the service instance.

Intuitively, the part of the situation tree [124] formed only by the actions that are possible (as specified by *Poss*) directly corresponds to the external execution tree of the service, where the final nodes are the situations in which *Final* is true. To formally define such an execution tree, we first inductively define a function $n(\cdot)$ from situations to sequences of actions union a special value *undef*:

- $n(S_0) = \varepsilon$;
- $n(do(a, s)) = n(s) \cdot a$ if $n(s) \neq undef$ and $Poss(a, s)$ holds;
- $n(do(a, s)) = undef$ otherwise.

The execution tree $T(\Gamma)$ generated by Γ is defined over the set of nodes $\{n(s) \mid n(s) \neq undef\}$, such that a node $n(s)$ is final if and only if $Final(s)$ holds. It is easy to check that $T(\Gamma)$ is indeed an execution tree. Note also that from such correspondence, also the part of the situation tree formed only by the actions that are possible is finitely representable.

Next, we turn to the problem of characterizing service composition. Let $\Gamma_1, \ldots, \Gamma_n$, be the theories for the component services $E_1, \ldots, E_n$, respectively, and let Γ_0 be the theory for the target service. The basic idea is to represent which services are executed when an action of the target service is performed. We do this by means of special predicates $Step_i(a, s)$, denoting that service E_i executes action a in situation s. Formally, we construct a Situation Calculus theory Γ_C formed by the union of the axioms below:

- Γ_0;
- Γ'_i, for $i = 1, \ldots, n$, where Γ'_i is obtained from Γ_i:
    1. by renaming each fluent F, including *Final*, to F_i;
    2. by renaming *Poss* to $Poss_i$;

3. by modifying the successor-state axioms as follows:
$\forall a, s. F_i(do(a, s)) \equiv (Step_i(a, s) \wedge \Phi_{F_i}(a, s)) \vee (\neg Step_i(a, s) \wedge F_i(s))$;

- $\forall a, s.(Poss(a, s) \wedge \neg Final(s)) \supset \bigvee_{i=1}^{n} Step_i(a, s) \wedge Poss_i(a, s)$;
- $\forall s. Final(s) \supset \bigwedge_{i=1}^{n} Final_i(s)$.

Observe that, because of the last two axioms, which encode domain independent conditions, the resulting theory Γ_C is not a Basic Action Theory. In Γ_C, we do not have anymore complete knowledge on the value of the fluents of the various services. This is due to the new form of the successor-state axioms, which make fluents depend on the predicates $Step_i$, whose value is not determined uniquely by Γ_C. Note however that if we did know such values in every situation, then the value of all the fluents would be determined. Note also that the value of $Step_i$ is constrained by the last two axioms so that, in every situation that is not final for the target service E_0, at least one of the component services steps forward. Finally, the last axiom states that, if E_0 is final, then so are all component services.

It can be shown that Γ_C *(i)* characterizes all the internal execution trees that conform to the external execution tree $T(\Gamma_0)$ generated by Γ_0; *(ii)* delegates all actions to $E_1, \ldots, E_n$; *(iii)* is coherent with $E_1, \ldots, E_n$. More precisely it can be shown that from each model of Γ_C one can construct one such internal execution tree and that on the other hand starting from each such internal execution tree one can construct a model of Γ_C.

This characterization allow us to reduce checking for the existence of a composition to checking satisfiability of a propositional Situation Calculus theory.

Theorem 53 *Let $\Gamma_0, \Gamma_1, \ldots, \Gamma_n$ be the Basic Action Theories representing the services $E_0, E_1, \ldots, E_n$ respectively, and let Γ_C be the theory defined as above. Then, Γ_C is satisfiable if and only if E_0 can be composed using $E_1, \ldots, E_n$.*

Proof. Let $T(\Gamma_0), T(\Gamma_1), \ldots, T(\Gamma_n)$ be the execution trees generated by $\Gamma_0, \Gamma_1, \ldots, \Gamma_n$, respectively.

"$\Leftarrow$" If E_0 can be composed using $E_1, \ldots, E_n$, then by Definition 10 there exists an internal schema ${E_0}^{int}$ of E_0 such that *(i)* $T({E_0}^{int})$ conforms to $T(\Gamma_0)$, *(ii)* $T({E_0}^{int})$ delegates all actions to $E_1, \ldots, E_n$, and *(iii)* $T({E_0}^{int})$ is coherent with $E_1, \ldots, E_n$.

From $T({E_0}^{int})$ and $T(\Gamma_0)$ we can obtain a model $\mathcal{M}$ of Γ_C as follows.

First, from $T(\Gamma_0)$ it is straightforward to build a model $\mathcal{M}_0$ of Γ_0 (it suffices to use the function $n(\cdot)$ from situations to sequences of actions [2] defined above).

[2]Each node x of an execution tree is characterized by the sequence of actions (see Section 3.3.1) leading to x.

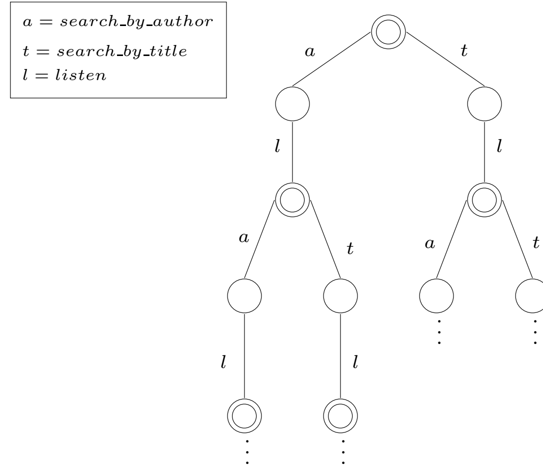


Figure A.1: External execution tree of service E_0

From $T({E_0}^{int})$, we can then extend $\mathcal{M}_0$ to a model $\mathcal{M}'_0$ by adding the truth-value of $Step_i(a, s)$ for every component service E_i, every action a, and every situation s. The truth-value of $Step_i(a, s)$ is defined by the elements (e_i, E_i) of the set $I_{(a,n(s))}$ that labels action a at node $n(s)$ of $T({E_0}^{int})$ with the instance e_i of service E_i executing a (Definition 3): if $E_i \in \pi_1(I_{(a,n(s))})$ (where $\pi_1(\cdot)$ is the projection over the first component) i.e., there exists a pair in $I_{(a,n(s))}$ where E_i is the first component, then $Step_i(a, s) = \texttt{true}$ in situation s. Now consider that $Poss_i$ and all fluents F_i, including $Final_i$, are completely determined once each $Step_i$ is determined, due to the form of precondition and successor-state axioms, respectively. This means that we can further extend $\mathcal{M}'_0$ to a model $\mathcal{M}$ of $\Gamma'_1, \ldots, \Gamma'_n$. Moreover, by construction of $T({E_0}^{int})$, such a model $\mathcal{M}$ satisfies the last two axioms of Γ_C, and hence the whole Γ_C.

"$\Rightarrow$" Let $\mathcal{M}$ be a model of Γ_C. Note that Γ_C is an extension of Γ_0, hence the execution tree generated by Γ_C is $T(\Gamma_0)$. Now, by using the truth values of $Step_i(a, s)$ for every component service E_i, every action a, and every situation s, one can construct the internal execution tree $T({E_0}^{int})$ of E_0. Due to the constraints posed to the interpretation of $Step_1, \ldots, Step_n$ by the theory Γ_C, $T({E_0}^{int})$ is indeed a composition. $\square$

Example 29 *Consider the scenario of Examples 4 and 5, which we report here for sake of readability. Service E_0 allows for searching and listening to* `mp3` *files. In particular, the client may choose to search for a song by specifying either its author(s) or its title (action* `search_by_author` *and* `search_by_title`*, respectively). Then the client selects and listens to a song (action* `listen`*). Finally, the client chooses whether to perform those actions again. Figure A.1*

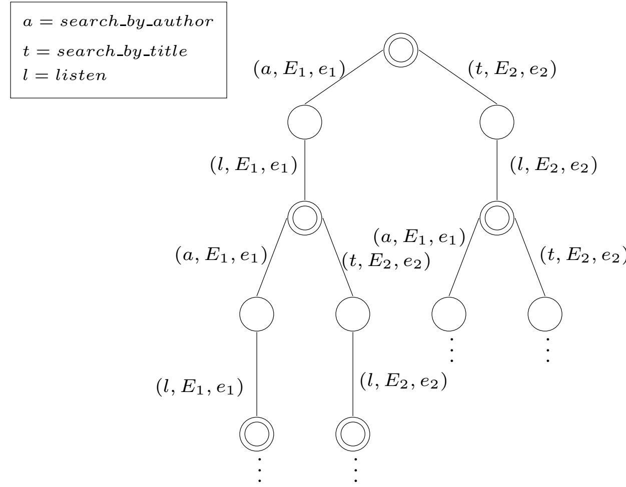


Figure A.2: Internal Execution tree of service E_0

shows the external execution tree of E_0. The community of services E_0 belongs to is constituted by two services, E_1 and E_2: E_1 iteratively allows to (i) `search_by_author` *an mp3 file and to* (ii) `listen` *to it; E_2 iteratively allows to* (i) `search_by_title` *an mp3 file and to* (ii) `listen` *to it. E_0 delegates its actions to E_1 and E_2: this is shown in Figure A.2, which represents the internal execution tree of E_0. In other words, if the external execution tree of E_0 is considered as the target service, and the service community is constituted by E_1 and E_2, it is easy to verify that there exists an internal schema for E_0 which is a composition of E_1 and E_2. This can be checked by representing E_0, E_1, and E_2 as Situation Calculus theories Γ_0, Γ_1, Γ_2, construct from them Γ_C, and check its satisfiability. In what follows, given the simplicity of the running example, we discuss a simplified encoding, where there is only the fluent* Final. *We leave as an exercise to the reader the extension of the provided encoding to the general case. In the encoding we denote action* `search_by_author` *by a, action* `search_by_title` *by t, action* `listen` *by l.*

The Action Theory Γ_C for the running example is constituted by the union of the the following axioms:

- *Basic Action Theory Γ_0 describing the target service:*

$$\begin{array}{lcl} \forall s.Poss(a,s) & \equiv & Final(s) \\ \forall s.Poss(t,s) & \equiv & Final(s) \\ \forall s.Poss(l,s) & \equiv & \neg Final(s) \\ \\ \forall \alpha, s.Final(do(\alpha,s)) & \equiv & (\alpha = r \wedge \neg Final(s)) \vee \\ & & (Final(s) \wedge \alpha \neq a \wedge \alpha \neq t) \end{array}$$

Note that actually, from the execution tree in Figure A.1, it is possible to execute action a and t only from final nodes, while action l can be executed from non-final nodes.

- *Action Theory Γ_1' describing the service E_1:*

$$\begin{array}{lll}\forall s.Poss_1(a,s) & \equiv & Final_1(s)\\ \forall s.Poss_1(t,s) & \equiv & \texttt{false}\\ \forall s.Poss_1(l,s) & \equiv & \neg Final_1(s)\end{array}$$

$$\begin{array}{lll}\forall \alpha, s.Final_1(do(\alpha,s)) & \equiv & (Step_1(\alpha,s) \wedge \alpha = r \wedge \neg Final_1(s)) \vee\\ & & (\neg Step_1(\alpha,s) \wedge Final_1(s) \wedge \alpha \neq a)\end{array}$$

The second axiom states that E_1 cannot execute action t, as it is.

- *Action Theory Γ_2' describing the service E_2 is analogous to Γ_1':*

$$\begin{array}{lll}\forall s.Poss_2(a,s) & \equiv & \texttt{false}\\ \forall s.Poss_2(t,s) & \equiv & Final_2(s)\\ \forall s.Poss_2(l,s) & \equiv & \neg Final_2(s)\end{array}$$

$$\begin{array}{lll}\forall \alpha, s.Final_2(do(\alpha,s)) & \equiv & (Step_2(\alpha,s) \wedge \alpha = r \wedge \neg Final_2(s)) \vee\\ & & (\neg Step_2(\alpha,s) \wedge Final_2(s) \wedge \alpha \neq a)\end{array}$$

- *Domain independent conditions:*

$$\begin{array}{lll}\forall a, s.(Poss(a,s) \wedge \neg Final(s)) & \supset & Step_1(a,s) \wedge Poss_1(a,s) \vee\\ & & Step_2(a,s) \wedge Poss_2(a,s)\\ \forall sper Final(s) & \supset & Final_1(s) \wedge Final_2(s)\end{array}$$

- *Axiom describing the initial situation:*

$$Final(S_0)$$

Satisfiability of Γ_C can be checked using standard techniques. □

A.3 Computing Service Composition

Next we turn to the problem of actually synthesizing a composite service. To do so, we resort to description logics and we re-express Situation Calculus Action Theories as an $\mathcal{ALU}$ [8] knowledge base. We refer the reader to Section 7.2 for an introduction to description logics and $\mathcal{ALU}$.

We define a mapping δ from (uniform) Situation Calculus formulas (wlog in negation normal form) with a free situation variable s to boolean combination of concepts as follows:

$$\begin{array}{rcl}
\delta(F(s)) & = & F, \quad \text{for each fluent } F \\
\delta(Poss(a,s)) & = & Poss_a, \quad \text{(similarly for } Poss_i(a,s)\text{)} \\
\delta(Step_i(a,s)) & = & Step_a_i, \quad \text{for each } i \in 1..n \\
\delta(\neg\varphi(s)) & = & \neg\delta(\varphi(s)) \quad (\varphi \text{ is an atomic proposition}) \\
\delta(\varphi_1(s) \wedge \varphi_2(s)) & = & \delta(\varphi_1(s)) \sqcap \delta(\varphi_2(s)) \\
\delta(\varphi_1(s) \vee \varphi_2(s)) & = & \delta(\varphi_1(s)) \sqcup \delta(\varphi_2(s))
\end{array}$$

Also, we consider an $\mathcal{ALU}$ role for each atomic action in Σ.

Next, we define the $\mathcal{ALU}$ counterpart Δ_C of Γ_C as the following knowledge base.

- to model the situation tree, we add the assertion $\top \sqsubseteq \bigsqcap_{a\in\Sigma} \exists a.\top$, and implicitly take into account the tree model property and the unique successor property;
- to model the initial situation Φ_0, we add the assertion $\mathsf{Init} \sqsubseteq \delta(\Phi_0)$, where Init is a new atomic concept denoting the initial situation;
- for each precondition axiom $\forall s.Poss(a,s) \equiv \Psi_a(s)$, we add the assertion $\delta(Poss(a,s)) \equiv \delta(\Psi_a(s))$; similarly for the modified precondition axioms in $\Gamma'_1, \ldots, \Gamma'_n$;
- for each successor-state axiom $\forall a, s.F(do(a,s)) \equiv \Phi_F(a,s)$, we first instantiate the axiom for each action in Σ and we simplify the equalities on actions. Then, for each instantiated successor-state axiom $F(do(\bar{a},s)) \equiv \Phi_F^{\bar{a}}(s)$ – where $\Phi_F^{\bar{a}}(s)$ is what we obtain from $\Phi_F(a,s)$ once we instantiate it on the action $\bar{a}$ and resolve the equalities on actions – we add the assertion $\forall\bar{a}.F \equiv \delta(\Phi_F^{\bar{a}}(s))$;
- for the last two axioms of Γ_C, we add the assertions $Poss_a \wedge \neg Final \sqsubseteq \bigsqcup_{i=1}^{n} Step_a_i \sqcap Poss_a_i$ and $Final \sqsubseteq \bigsqcap_{i=1}^{n} Final_i$.

Note that, in the above construction, it is necessary to instantiate the successor-state axioms for each action, since, contrary to the Situation Calculus, $\mathcal{ALU}$ does not admit quantification over actions. From the above construction we get the following theorem.

Theorem 54 *The* Init *concept is satisfiable in the* $\mathcal{ALU}$*-counterpart* Δ_C *of* Γ_C *if and only if* Γ_C *is satisfiable.*

Observe that the size of Δ_C is at most equal to the size of Γ_C times the number of actions in Σ. Hence, from the EXPTIME-completeness of concept satisfiability in $\mathcal{ALU}$ knowledge bases and from Theorem 54 we get the following complexity result.

Theorem 55 *Checking the existence of a service composition can be done in EXPTIME.*

Observe that, because of the small model property and the single successor property of Description Logics, if Init is satisfiable in Δ_C one can always obtain a model which is single successor and of size at most exponential. From such a model one can immediately extract a finite (possibly exponential) representation of the internal execution tree constituting the composition. Also from such a representation one can build a Situation Calculus Basic Action Theory (or its counterpart in $\mathcal{ALU}$ if needed) that generates exactly such a internal execution tree.

The results obtained in this section are perfectly compatible with the results in Section 7.2.

As already discussed in Chapter 7, from a practical point of view, one can use current highly optimized Description Logic systems [8, 78] to check the existence of service compositions. Since these systems are based on tableaux techniques that construct a model when checking for satisfiability, one can, with minor modifications, also return such a model, which correspond to the internal execution tree constituting the composition.

A.4 Discussion

In this chapter we have studied an alternative way to check composition existence of services resorting to a Propositional Situation Calculus setting, a well-known formalism for reasoning about actions. In such a setting we have given a characterization of the problem of finding a composition in terms of satisfiability of a certain action theory. Finally, resorting to a translation of such a Situation Calculus theory in a Description Logic we have shown that such a problem is EXPTIME, and that current tableaux based DL-reasoning procedures can be used to actually obtain the composition.

We want to observe that what our Propositional Situation Calculus setting can capture is essentially a description of services given in terms of finite state machines (compactly represented by resorting on propositional fluents). This is a particularly interesting class of descriptions since it is one of the classes most commonly used to describe services in the literature [110, 40, 30].

Note that our definition of service composition does not require execution trees to be finite branching (i.e. to have only a finite set of possible interac-

tions), but also allows for infinite branching execution trees. This opens up the possibility of having parameterized actions, whose parameters are arbitrary terms. To capture services and service composition in this case, one has to resort to the full (non-propositional) Situation Calculus. Observe that the logical theories that represent execution trees need to be complete (we have complete information on such trees) and that particular care must be taken in order to have terms denoting each action (i.e., one needs some form of infinite domain closure, expressible only in second-order logic). Naturally, in general, decidability of composition is lost in this case, and it becomes of interest to understand for which theories decidability is preserved.

In this chapter we have made use of Situation Calculus, especially because it is one of the best known formalism for reasoning about actions. However, the basic ideas of this paper may be easily exported to other reasoning about actions formalisms. Of particular interest is looking at service compositions, as defined here, in the framework proposed by [108, 107]. That is each composite service, is represented by a GOLOG/CONGOLOG program (which indeed defines an execution tree, although possibly nondeterministic). Observe that one of the main differences between our approach and that in [108, 107] is that in our approach the client's needs are themselves expressed by a service, while in [108, 107] they are expressed as customization conditions on desired composite services.

Bibliography

[1] J. Blythe and J.L. Ambite (ed.), *Proceedings of the 2nd ICAPS International Workshop on Planning and Scheduling for Web and Grid Services (P4WGS)*, 2004.

[2] M. Aiello, M.P. Papazoglou, J. Yang, M. Carman, M. Pistore, L. Serafini, and P. Traverso, *A Request Language for Web-Services Based on Planning and Constraint Satisfaction*, Proceedings of the 3rd VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2002), LNCS, vol. 2444, Springer, 2002, pp. 76–85.

[3] J.L. Ambite, C. Knoblock, S. McIlraith, M.P. Papazoglou, B. Srivastava, and P. Traverso (eds.), *Proceedings of the 1st ICAPS International Workshop on Planning for Web Services (P4WS03)*, 2003.

[4] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, *Business Process Execution Language for Web Services (BPEL4WS) -Version 1.1*, `http://www-106.ibm.com/developerworks/library/ws-bpel/`, 2004.

[5] A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara, *DAML-S: Web Service Description for the Semantic Web*, Proceedings of the 1st International Semantic Web Conference (ISWC 2002), LNCS, vol. 2342, Springer, 2002, pp. 348–363.

[6] Ariba, Microsoft, and IBM, *Web Services Description Language (WSDL) 1.1*, Available on line: `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`, 2001.

[7] F. Baader and U. Sattler, *An Overview of Tableau Algorithms for Description Logics*, Studia Logica **69** (2001), no. 1, 5–40.

[8] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P.F. Patel-Schneider (eds.), *The description logic handbook: Theory, implementation and applications*, Cambridge University Press, 2003.

[9] F. Bacchus and F. Kabanza, *Planning for Temporally Extended Goals*, Annals of Mathematics and Artificial Intelligence **22** (1998), 5–27.

[10] K. Baïna, B. Benatallah, F. Casati, and F. Toumani, *Model-Driven Web Service Development*, Proceedins of 16th International Conference on Advanced Information Systems Engineering (CAiSE 2004), LNCS, vol. 3084, Springer, 2004, pp. 290–306.

[11] C. Batini and M. Mecella, *Enabling Italian* e-*Government Through a Cooperative Architecture*, IEEE Computer **34** (2001), no. 2, 40–45.

[12] BEA, Intalio, SAP, and Sun, *Web Service Choreography Interface (WSCI) 1.0*, W3C Document. `http://www.w3.org/TR/wsci/`, 2002.

[13] M. Ben-Ari, J.Y. Halpern, and A. Pnueli, *Deterministic propositional dynamic logic: Finite models, complexity, and completeness*, Journal of Computer and System Sciences **25** (1982), 402–417.

[14] B. Benatallah, Q.Z. Sheng, and M. Dumas, *The Self-Serv Environment for Web Services Composition*, IEEE Internet Computing **7** (2003), no. 1, 40 – 48.

[15] B. Benatallah, F. Casati, H. Skogsrud, and F. Toumani, *Abstracting and Enforcing Web Service Protocols*, International Journal of Cooperative Information Systems (IJCIS) (2004), To appear.

[16] B. Benatallah, F. Casati, and F. Toumani, *Analysis and Management of Web Services Protocols*, Proceedings of 23th International Conference on Conceptual Modeling (ER 2004), 2004, To appear.

[17] B. Benatallah, F. Casati, F. Toumani, and R. Hamadi, *Conceptual Modeling of Web Service Conversations*, Proceedins of 15th International Conference on Advanced Information Systems Engineering (CAiSE 2003), LNCS, vol. 2681, Springer, 2003, pp. 449–467.

[18] B. Benatallah, M.S. Hacid, C. Rey, and F. Toumani, *Request Rewriting-Based Web Service Discovery*, Proceedings of the 2nd International Semantic Web Conference (ISWC 2003), LNCS, vol. 2870, Springer, 2003, pp. 242–257.

[19] D. Berardi, *Automatic Composition of Finite State e-Services*, Technical report, American Association for Artificial Intelligence (AAAI), 2004, To appear.

[20] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella, *Automated e-Service Composition using Simulation*, Manuscript, June 2003.

[21] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella, *e-Service Composition by Description Logic Based Reasoning*, Proceedings of the 2003 International Workshop on Description Logics (DL03), CEUR Workshop Proceedings, vol. 81, 2003, Available at: `http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-81/`.

[22] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella, *A Foundational Vision of e-Services*, Proceedings of the CAiSE 2003 Workshop on Web Services, e-Business, and the Semantic Web (WES 2003), LNCS, vol. 3095, Springer, 2004, pp. 28–40.

[23] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella, *Automatic Services Composition based on Behavioral Descriptions*, International Journal of Cooperative Information Systems (IJCIS) (2004), To appear.

[24] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella, *$\mathcal{ESC}$: A Tool for Automatic Composition of e-Services based on Logics of Programs*, Proceedings of the 5th VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2004), 2004, To appear.

[25] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella, *Synthesis of Composite e-Services based on Automated Reasoning*, Proceedings of the 2nd ICAPS International Workshop on Planning and Scheduling for Web and Grid Services, 2004.

[26] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella, *A Foundamental Framework for e-Services*, Technical Report 10-03, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Roma, Italy, 2003.

[27] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella, *Synthesis of Underspecified Composite e-Services on Automated Reasoning*, Technical Report 9, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Roma, Italy, 2004.

[28] D. Berardi, D. Calvanese, G. De Giacomo, and M. Mecella, *Composing e-Services by Reasoning about Actions*, Proceedings of the 1st ICAPS 2003 International Workshop on Planning for Web Services (P4WS 2003), 2003.

[29] D. Berardi, R. Hull, M. Gruninger, and S. McIlraith, *Towards a First-Order Ontology for Semantic Web Services*, 2004, Accepted for publication as position paper at the International W3C Workshop on Constraints and Capabilities for Web Services.

[30] D. Berardi, F. De Rosa, L. De Santis, and M. Mecella, *Finite State Automata as Conceptual Model for e-Services*, Transactions of the SDPS: Journal of Integrated Design and Process Science **8** (2004), no. 2, 105–121.

[31] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella, *Automatic Composition of* e-*Services that Export their Behavior*, Proceedings of the 1st International Conference on Service Oriented Computing (ICSOC 2003), LNCS, vol. 2910, Springer, 2003, pp. 43–58.

[32] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella, *Synthesis of Underspecified Composite e-Services based on Automated Reasoning*, Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC 2004), 2004, To appear.

[33] P. Bertoli and M. Pistore, *Planning with Extended Goals and Partial Observability*, Proceedings of 14th International Conference on Automated Planning and Scheduling (ICAPS 2004), AAAI, 2004, pp. 270–278.

[34] T. Bolognesi and E. Brinksma, *Introduction to the ISO Specification Language LOTOS*, The Formal Description Technique LOTOS (P. H. J. van Eijk, C. A. Vissers, and M. Diaz, eds.), Elsevier Science Publishers North-Holland, 1989, pp. 23–73.

[35] L. Bordeaux and G. Salaün, *Using Process Algebra for Web Services: Early Results and Perspectives*, Proceedings of the 5th VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2004), 2004, To appear.

[36] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella, *When are two Web Services Compatible?*, Proceedings of the 5th VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2004), 2004, To appear.

[37] A. Bracciali, A. Brogi, and C. Canal, *Dynamically Adapting the Behaviour of Software Components*, Proceedings of Conference on Coordination Models and Languages (COORDINATION), LNCS, vol. 2315, Springer, 2002, pp. 88 – 95.

[38] A. Brogi, E. Pimentel, and A. M. Roldán, *Compatibility of Linda-based Component Interfaces*, Proceedings of International Workshop on Formal Methods and Component Interaction (FMCI), vol. 66, Electronic Notes on Theoretical Computer Science, no. 4, 2002.

[39] M. Buchheit, F.M. Donini, and A. Schaerf, *Decidable reasoning in terminological knowledge representation systems*, Journal of Artificial Intelligence Research **1** (1993), 109–138.

[40] T. Bultan, X. Fu, R. Hull, and J. Su, *Conversation Specification: A New Approach to Design and Analysis of E-Service Composition*, Proceedings of the 12th International World Wide Web Conference (WWW 2003), ACM, 2003, pp. 403–410.

[41] D. Calvanese and G. De Giacomo, *Expressive description logics*, The Description Logic Handbook: Theory, Implementation and Applications (F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P.F. Patel-Schneider, eds.), Cambridge University Press, 2003, pp. 178–218.

[42] D. Calvanese, G. De Giacomo, M. Lenzerini, and D. Nardi, *Reasoning in expressive description logics*, Handbook of Automated Reasoning (A. Robinson and A. Voronkov, eds.), Elsevier Science Publishers (North-Holland), Amsterdam, 2001, pp. 1581–1634.

[43] D. Calvanese, G. De Giacomo, and M.Y. Vardi, *Reasoning about Actions and Planning in LTL Action Theories*, Proceedings of the 8th International Conferences on Principles of Knowledge Representation and Reasoning (KR 2002), 2002, pp. 593–602.

[44] M.A. Cameron, K.L. Taylor, and R. Baxter, *Web Service Composition and Record Linking*, Proceedings of VLDB Workshop on Information Integration on the Web (IIWeb-04), 2004, pp. 65–70.

[45] C. Canal, L. Fuentes, E. Pimentel, J.M. Troya, and A. Vallecillo, *Adding Roles to CORBA Objects*, IEEE Transactions on Software Engineering **29** (2003), no. 8, 242–260.

[46] F. Casati and M.C. Shan, *Dynamic and Adaptive Composition of* e-*Services*, Information Systems **6** (2001), no. 3, 143 – 163.

[47] J. Castro, M. Kolp, and J. Mylopoulos, *Towards Requirements-Driven Information Systems Engineering: The Tropos Project*, Information Systems **27** (2002), no. 6, 365–289.

[48] R. Chinnici, M. Gudgin, J.J. Moreau, J. Schlimmer, and S. Weerawarana, *Web Services Description Language (WSDL) 2.0*, Available on line: http://www.w3.org/TR/wsdl20, 2003, W3C Working Draft.

[49] V. Christophides, R. Hull, G. Karvounarakis, A. Kumar, G. Tong, and M. Xiong, *Beyond Discrete* e-*Services: Composing Session-oriented Services in Telecommunications*, Proceedings of the 2nd VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2001), LNCS, vol. 2193, Springer, 2001, pp. 58 – 73.

[50] E. Colombo, C. Francalanci, B. Pernici, P. Plebani, M. Mecella, V. De Antonellis, and M. Melchiori, *Cooperative Information Systems in Virtual Districts: the VISPO Approach*, IEEE Data Engineering Bulletin **25** (2002), no. 4, 36 – 40.

[51] Jupitermedia Corporation, *Webopedia: Online Computer Dictionary for Computer and Internet Terms*, http://www.webopedia.com/.

[52] G. De Giacomo and F. Massacci, *Combining Deduction and Model Checking into Tableaux and Algorithms for Converse-PDL*, Information and Computation **160** (2000), no. 1–2, 117–137.

[53] G. De Giacomo and M. Lenzerini, *PDL-based framework for reasoning about actions*, Proceedings of the 4th Conference of the Italian Association for Artificial Intelligence (AI*IA'95), LNAI, vol. 992, Springer, 1995, pp. 103–114.

[54] A. Deutsch, L. Sui, and V. Vianu, *Specification and Verification of Data-driven Web Services*, Proceedings of the 23nd ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS 2004), ACM, 2004, pp. 71–82.

[55] A. Dix, J. Finlay, G. Abowd, and R. Beale, *Human-Computer Interaction*, Prentice Hall, 1998.

[56] A. Dogac, I. Cingil, G. Laleci, and Y. Kabak, *Improving the Functionality of UDDI Registries through Web Service Semantics*, Proceedings of the 3rd VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2002), LNCS, vol. 2444, Springer, 2002, pp. 9–18.

[57] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang, *Similarity Search for Web Services*, Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004), Morgan Kaufmann, 2004, pp. 132–143.

[58] F.M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf, *Reasoning in Description Logics*, Principles of Knowledge Representation (G. Brewka, ed.), Studies in Logic, Language and Information, CSLI Publications, 1996, pp. 193–238.

[59] V. Draluk, *Discovering Web Services: An Overview*, Proceedings of the 27th International Conference on Very Large Databases Conference (VLDB 2001), 2001.

[60] E.A. Emerson and E.M. Clarke, *Using Branching-Time Logic to Synthesize Program Skeleton*, Science of Computer Programming (1982), no. 2, 241–266.

[61] M. Fattorosi-Barnaba and F. De Caro, *Graded modalities I*, Studia Logica **44** (1985), 197–221.

[62] M.C. Fauvet, M. Dumas, B. Benatallah, and H.Y. Paik, *Peer-to-Peer Traced Execution of Composite Services*, Proceedings of the 2nd VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2001), LNCS, vol. 2193, Springer, 2001, pp. 103 – 117.

[63] D. Fensel, K.P. Sycara, and J. Mylopoulos (eds.), *Proceedings of the 2nd international semantic web conference (iswc 2003)*, LNCS, vol. 2870, 2003.

[64] K. Fine, *In so many possible worlds*, Notre Dame Journal of Formal Logic **13** (1972), no. 4, 516–520.

[65] J. Firby, *An Investigation into Reactive Planning in Complex Domains*, Proceedings of the 6th National Conference on Artificial Intelligence (AAAI 1987), AAAI Press, 1987, pp. 202–206.

[66] M.J. Fischer and R.E. Ladner, *Propositional dynamic logic of regular programs*, Journal of Computer and System Sciences **18** (1979), 194–211.

[67] M. Fowler and K. Scott, *UML Distilled – Applying the Standard Object Modeling Laguage*, Addison-Wesley, 1997.

[68] X. Fu, T. Bultan, and J. Su, *Analysis of interacting BPEL web services*, Proceedings of the 13th International World Wide Web Conference (WWW 2004), ACM, 2004, pp. 621–630.

[69] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services. Concepts, Architectures and Applications*, Springer, 2004.

[70] G. Gans, M. Jarke, G. Lakemeyer, and D. Schmitz, *Deliberation in a Modeling and Simulation Environment for Inter-organizational Networks*, Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE'03), LNCS, vol. 2681, Springer, 2003, pp. 242–257.

[71] C.E. Gerede, R. Hull, O.H. Ibarra, and J. Su, *Automated Composition of E-services: Lookaheads*, Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC 2004), 2004.

[72] M. Ghallab, D. Nau, and P. Traverso, *Automated Task Planning: Theory & Practice*, Morgan Kaufmann, 2004.

[73] G. De Giacomo, Y. Lesperance, and H. Levesque, *ConGolog, a Concurrent Programming Language Based on the Situation Calculus*, Artificial Intelligence **1–2** (2000), no. 121, 109–169.

[74] B. Grosof, M. Gruninger, M. Kifer, D. Martin, D. McGuinness, B. Parsia, T. Payne, and A. Tate, *Semantic Web Services Language Requirements*, `http://www.daml.org/services/swsl/requirements/swsl-requirements.shtml`, 2004.

[75] M. Gruninger, *A Guide to the Ontology of the Process Specification Language*, Handbook of Ontologies (R. Studer and S. Staab, eds.), Springer-Verlag, 2003, pp. 575–592.

[76] M. Gruninger and C. Menzel, *Process Specification Language: Theory and Applications*, AI Magazine **24** (2003), 63–74.

[77] A. Karp H. Kuno, M. Lemon and D. Beringer, *Conversations + Interfaces = Business Logic*, Proceedings of the 2nd VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2001), LNCS, vol. 2193, Springer, 2001, pp. 30 – 43.

[78] V. Haarslev and R. Möller, *RACER system description*, Proceedings of IJCAR 2001, LNAI, vol. 2083, Springer, 2001, pp. 701–705.

[79] V. Haarslev and R. Möller, *Expressive ABox Reasoning with Number Restrictions, Role Hierarchies, and Transitively Closed Roles*, Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR 2000), 2000, pp. 273–284.

[80] A.Y. Halevy, *Answering queries using views: A survey*, Very Large Data Base Journal **10** (2001), no. 4, 270–294.

[81] D. Harel, D. Kozen, and J. Tiuryn, *Dynamic logic*, The MIT Press, 2000.

[82] D. Harel, O. Kupferman, and M.Y. Vardi, *On the Complexity of Verifying Concurrent Transition Systems*, Information and Computation **173** (2002), no. 2, 143–161.

[83] D. Harkey and R. Orfali, *Client/Server Programming with Java and CORBA (2nd Edition)*, Wiley & Sons, 1998.

[84] I. Horrocks, *The FaCT system*, Proceedings of TABLEAUX'98, LNAI, vol. 1397, Springer, 1998, pp. 307–312.

[85] I. Horrocks and J.A. Hendler (eds.), *Proceedings of the 1st International Semantic Web Conference (iswc 2002)*, LNCS, vol. 2342, 2002.

[86] I. Horrocks and P.F. Patel-Schneider, *Optimizing Description Logic Subsumption*, Journal of Logic and Computation **9** (1999), no. 3, 267–293.

[87] HP, *Web Services Concepts: a Technical Overview*, HP Document. `http://www.bluestone.com/downloads/pdf/web_services_tech_overview.pdf`, 2001.

[88] R. Hull, *Web Services Composition: A Story of Models, Automata and Logics*, Presented at the EDBT 2004 Summer School. Available online at: `http://edbtss04.dia.uniroma3.it/program.html/`.

[89] R. Hull, M. Benedikt, V. Christophides, and J. Su, *E-Services: A Look Behind the Curtain*, Proceedings of the 22nd ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS 2003), ACM, 2003, pp. 1–14.

[90] N. Kavantzas, D. Burdett, and G. Ritzinger, *Web Services Choreography Description Language (WS-CDL) Version 1.0*, Available on line at: `http://www.w3.org/TR/ws-cdl-10/`, W3C Working Draft.

[91] T. Kirk, A.Y. Levy, Y. Sagiv, and D. Srivastava, *The Information Manifold*, Working Notes of the AAAI 1995 Spring Symposium on Information Gathering from Heterogeneous, Distributed Enviroments, AAAI Press, 1995, pp. 85–91.

[92] D. Kozen and J. Tiuryn, *Logics of programs*, Handbook of Theoretical Computer Science — Formal Models and Semantics (Jan van Leeuwen, ed.), Elsevier Science Publishers (North-Holland), Amsterdam, 1990, pp. 789–840.

[93] O. Kupferman, P. Madhusudan, P.S. Thiagarajan, and M.Y. Vardi, *Open Systems in Reactive Environments: Control and Synthesis*, Proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000), LNCS, vol. 1877, Springer, 2000, pp. 92–107.

[94] O. Kupferman and M.Y. Vardi, *Church's Problem Revisited*, The Bulletin of Symbolic Logic **5** (1999), no. 2, 245–263.

[95] O. Kupferman, M.Y. Vardi, and P. Wolper, *An Automata-Theoretic Approach to Branching-Time Model Checking*, Journal of the ACM **47** (2000), no. 2, 312 – 360.

[96] O. Kupferman and M.Y. Vardi, *Freedom, Weakness, and Determinism: from Linear-Time to Branching-Time*, Proceedings of the 13th IEEE Symposium on Logic in Computer Science, 1998, pp. 81–92.

[97] O. Kupferman and M.Y. Vardi, *Weak Alternating Automata Tree and Tree Automata Emptiness*, Proceedings of the 30th ACM SIGACT Symposium on Theory of Computing (STOC'98), 1998, pp. 224–233.

[98] U. Kuter, E. Sirin, D. Nau, B. Parsia, and J. Hendler, *Information Gathering during Planning for Web Service Composition*, Proceedings of the 2nd ICAPS International Workshop on Planning and Scheduling for Web and Grid Services, 2004.

[99] U. Dal Lago, M. Pistore, and P.Traverso, *Planning with a Language for Extended Goals*, Proceedings of the 18th National Conference on Artificial Intelligence (AAAI 2002), AAAI Press, 2002, pp. 447–454.

[100] A. Lazovik, M. Aiello, and M.P. Papazoglou, *Planning and Monitoring the Execution of Web Service Requests*, Proceedings of the 1st International Conference on Service Oriented Computing (ICSOC 2003), LNCS, vol. 2910, Springer, 2003, pp. 335–350.

[101] M. Lenzerini, *Data Integration: A Theoretical Perspective*, Proceedings of the 21nd ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS 2002), ACM, 2002, pp. 233–246.

[102] F. Leymann, *Web Service Flow Language (WSFL 1.0)*, IBM Document. `http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`, 2001.

[103] C. Lutz and U. Sattler, *A Proposal for Describing Services with DLs*, Proceedings of the 2002 Description Logic Workshop (DL 2002), CEUR Electronic Workshop Proceedings, `http://ceur-ws.org/Vol-53/`, 2002, pp. 128–139.

[104] A. Martens, *On compatibility of web services*, Petri Net Newsletter, no. 65, 2003, pp. 12–20.

[105] E. Martinez and Y. Lesperance, *Web Service Composition as a Planning Task: Experiments using Knowledge-Based Planning*, Proceedings of the 2nd ICAPS International Workshop on Planning and Scheduling for Web and Grid Services, 2004.

[106] D. McGuinness and J.R. Wright, *An Industrial Strength Description Logic-based Configuration Platform*, IEEE Intelligent Systems (1998), 69–77.

[107] S. McIlraith and T.C. Son, *Adapting Golog for Composition of Semantic Web Services*, Proceedings of the 8th International Conferences on Principles of Knowledge Representation and Reasoning (KR 2002), Morgan Kaufmann, 2002, pp. 482 – 493.

[108] S. McIlraith, T.C. Son, and H. Zeng, *Semantic Web Services*, IEEE Intelligent Systems **16** (2001), no. 2, 46 – 53.

[109] M. Mecella, F. Parisi Presicce, and B. Pernici, *Modeling e-Service Orchestration Through Petri Nets*, Proceedings of the 3rd VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2002), LNCS, vol. 2444, Springer, 2002, pp. 38–47.

[110] M. Mecella and B. Pernici, *Building flexible and cooperative applications based on* e-*services*, Tech. Report 21-2002, Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”, 2002, Available on line at `http://www.dis.uniroma1.it/~mecella/publications/mp_techreport_212002.pdf`.

[111] M. Mecella, B. Pernici, and P. Craca, *Compatibility of* e-*Services in a Cooperative Multi-Platform Environment*, Proceedings of the 2nd VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2001), LNCS, vol. 2193, Springer, 2001, pp. 44– 57.

[112] B. Medjahed, A. Bouguettaya, and A.K. Elmagarmid, *Composing Web services on the Semantic Web*, Very Large Data Base Journal **12** (2003), no. 4, 333–351.

[113] B. Medjahed, A. Bouguettaya, A.H.H. Ngu, and A.K. Elmagarmid, *Business-to-business interactions: issues and enabling technologies*, Very Large Data Base Journal **12** (2003), no. 1, 59–85.

[114] G. Meredith, *Contract and Types*, Invited Talk at the 3rd VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2002), 2002.

[115] Microsoft, *Distributed Component Object Model (DCOM)*, `http://www.microsoft.com`, 2003.

[116] R. Milner, *Communication and Concurrency*, International Series in Computer Science, Prentice Hall, 1989.

[117] R. Monson-Haefel, *Enterprise JavaBeans (3rd Edition)*, O'Reilly, 2001.

[118] S. Narayanan and S. McIlraith, *Simulation, Verification and Automated Composition of Web Services*, Proceedings of the 11th International World Wide Web Conference (WWW 2002), ACM Press, 2002, pp. 77 – 88.

[119] B. Orriëns, J. Yang, and M.P. Papazoglou, *ServiceCom: A Tool for Service Composition Reuse and Specialization*, Proceedings of the 4th International Conference on Web Information Systems Engineering (WISE 2003), IEEE Computer Society, 2003, pp. 355–358.

[120] B. Orriëns, J. Yang, and M.P. Papazoglou, *Service Component: a mechanism for web service composition reuse and specialization*, Transactions of the SDPS: Journal of Integrated Design and Process Science **8** (2004), no. 2, 13–28.

[121] M.P. Papazoglou and D. Georgakopoulos, *Service Oriented Computing (special issue)*, Communication of the ACM **46** (2003), no. 10, 24–28.

[122] C. Pautasso and G. Alonso, *From Web Service Composition to Megaprogramming*, Proceedings of the 5th VLDB International Workshop on Technologies for *e*-Services (VLDB-TES 2004), 2004, To appear.

[123] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso, *Planning and Monitoring Web Service Composition*, Proceedings of the 2nd ICAPS International Workshop on Planning and Scheduling for Web and Grid Services, 2004.

[124] R. Reiter, *Knowledge in action: Logical foundations for specifying and implementing dynamical systems*, The MIT Press, 2001.

[125] R.H. Katz, *Contemporany logic design*, Benjamin Commings/Addison Wesley Publishing Company, 1993.

[126] J. Rintanen, *Complexity of Planning with Partial Observability*, Proceedings of 14th International Conference on Automated Planning and Scheduling (ICAPS 2004), 2004.

[127] S. Ross-Talbot, *Web Services Choreography and Process Algebra*, Available at: `http://www.daml.org/services/swsl/materials/WS-CDL.pdf`, April 29th 2004, Presentation given to SWSL working group.

[128] G. Salaün, L. Bordeaux, and M. Schaerf, *Describing and Reasoning on Web Services using Process Algebra*, Proceedings of 2004 IEEE International Conference on Web Services (ICWS 2004), IEEE Computer Society Press, 2004, pp. 43–51.

[129] G. Salaün, A. Ferrara, and A. Chirichiello, *Negotiation among Web Services using LOTOS/CADP*, Proceedings of ECOWS'04, LNCS, vol. 3250, Springer, 2004, pp. 198–212.

[130] T. Satish, *XLANG. Web Services for Business Process Design*, Microsoft Document. `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.hm`, 2001.

[131] U. Sattler, *Terminological Knowledge Representation Systems in a Process Engineering Application*, Ph.D. thesis, LuFG Theoretical Computer Science, RWTH-Aachen, Germany, 1998.

[132] K. Schild, *A correspondence theory for terminological logics: Preliminary report*, Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI'91), Morgan Kaufmann, 1991, pp. 466–471.

[133] C. Schmidt and M. Parashar, *A Peer-to-Peer Approach to Web Service Discovery*, World Wide Web Journal **7** (2004), no. 2, 211–229.

[134] S. Schneider, J. Davies, D.M. Jackson, G.M. Reed, J.N. Reed, and A.W. Roscoe, *Timed CSP: Theory and Practice*, Proc. of REX Workshop on Real-Time: Theory in Practice (Germany) (J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, eds.), LNCS, vol. 600, Springer, 1992, pp. 640–675.

[135] G. Shegalov, M. Gillmann, and G. Weikum, *XML-enabled Workflow Management for* e-*Services across Heterogeneous Platforms*, Very Large Data Base Journal **10** (2001), no. 1, 91–103.

[136] M. Spielmann, *Abstract State Machines: Verification problems and complexity*, Ph.D. thesis, RWTH Aachen, 2000.

[137] M. Spielmann, *Verification of Relational Transducers for Electronic Commerce*, Journal of Computer and System Sciences **66** (2003), no. 1, 40–65.

[138] C. Szyperski, *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley, 2002.

[139] S. Thakkar, J.L. Ambite, and C.A. Knoblock, *A View Integration Approach to Dynamic Composition of Web Services*, Proceedings of the 1st ICAPS International Workshop on Planning for Web Services (P4WS 2003), 2003.

[140] S. Thakkar, J.L. Ambite, and C.A. Knoblock, *A Data Integration Approach to Automatically Composing and Optimizing Web Services*, Proceedings of the 2nd ICAPS International Workshop on Planning and Scheduling for Web and Grid Services, 2004.

[141] S. Thakkar, J.L. Ambite, C.A. Knoblock, and C. Shahabi, *Dynamically Composing Web Services from On-line Sources*, Proceeding of 2002 AAAI Workshop on Intelligent Service Integration, 2002, pp. 1–7.

[142] UDDI.org, *UDDI Technical White Paper*, (Available on line at: `http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf`), 2000.

[143] W.J. van den Heuvel, J. Yang, and M.P. Papazoglou, *Service Representation, Discovery and Composition for* e-*Marketplaces*, Proceedings of the 9th International Conference on Cooperative Information Systems (CoopIS 2001), LNCS, vol. 2172, Springer, 2001, pp. 270 – 284.

[144] W. Van der Hoek, *On the semantics of graded modalities*, Journal of Applied Non-Classical Logics **2** (1992), no. 1, 81–123.

[145] W. Van der Hoek and M. de Rijke, *Counting Objects*, Journal of Logic and Computation **5** (1995), no. 3, 325–345.

[146] W3C, *XML Protocol*, XML Protocol Web Page: `http://www.w3.org/2000/xp/`.

[147] W3C, *Simple Object Access Protocol (SOAP) - Version 1.2*, (Available on line at: `http://www.w3.org/TR/SOAP12/`), 2000.

[148] W3C, *Web Service Architecture Requirements*, Available on line: `http://www.w3.org/TR/wsa-reqs/`, 2002.

[149] Y. Wang and E. Stroulia, *Flexible Interface Matching for Web-Service Discovery*, Proceedings of the 4th International Conference on Web Information System Engineering (WISE 2003), IEEE Computer Society, 2003, pp. 147–156.

[150] G. Weikum, *Towards Guaranteed Quality and Dependability of Information Services*, Proceedings of 8th German Database Conference (Datenbanksysteme in Bro, Technik und Wissenschaft), Informatik Aktuell, Springer, 1999, Invited Talk, pp. 379–409.

[151] D. Wodtke and G. Weikum, *A Formal Foundation for Distributed Workflow Execution Based on State Charts*, Proceedings of the 6th International Conference on Database Theory (ICDT '97), 1997.

[152] J. Yang and M.P. Papazoglou, *Interoperation Support for Electronic Business*, Communication of the ACM **43** (2000), no. 6, 39–47.

[153] J. Yang and M.P. Papazoglou, *Web Components: A Substrate for Web Service Reuse and Composition*, Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE 2002), LNCS, vol. 2348, Springer, 2002, pp. 21–36.

[154] J. Yang and M.P. Papazoglou, *Service Components for Managing the Life-cycle of Service Compositions*, Information Systems **29** (2004), no. 2, 97 – 125.

[155] D. Yellin and R. Strom, *Protocol Specifications and Component Adaptors*, ACM Transactions on Programming Languages and Systems **19** (1997), no. 2, 292 – 333.

[156] E.S.K. Yu, *Modelling Strategic Relationships for Process Reengineering*, Ph.D. thesis, Dept. of Computer Science, University of Toronto, Toronto, ON, 1995.