Automatic composition of prototocol-based Web services

Ramy Ragab Hassen

Note d'ordre : 1943
EDSPIC : 442

Université Blaise Pascal – Clermont-Ferrand II
École Doctorale des Sciences pour l'Ingénieur de Clermont-Ferrand

Thèse de Doctorat

présenté par

Ramy RAGAB HASSEN

pour obtenir le grade de

Docteur d'Université

Spécialité : Informatique

Automatic composition of protocol-based web services

Thèse dirigée par Pr. Farouk TOUMANI et Pr. Lhouari NOURINE
Soutenue publiquement le 7 Juillet 2009 devant le jury suivant :

Prof. Claude Godart	Président
Prof. Djamal Benslimane	Rapporteur
Prof. Fabio Casati	Rapporteur
Prof. Eugene Asarin	Examinateur
Prof. Christophe Rey	Examinateur
Prof. Farouk TOUMANI	Directeur de Thèse
Prof. Lhouari NOURINE	Directeur de Thèse
Prof. Mohand-Said Hacid	Invité

This thesis is dedicated to my wonderful parents, my dear brothers Hani and Nadim and my adorable wife Dalia for their love and support. I dedicate also this work to my kind in laws.

Acknowledgment

My first and most heartfelt thanks go to my advisors, Pr. Farouk Toumani and Pr. Lhouari Nourine for invaluable discussions and advice throughout the course of this research, and for many helpful comments and concise without which this dissertation would not have been achieved. I cannot thank them enough. I am truly fortunate to have had such advisors who were able to guide me and gave me both freedom and support to pursue my research project.

I owe many thanks to Pr. Fabio Casati and Pr. Djamal Benslimane for taking the time to review my thesis report and providing useful comments and suggestions to further improve this dissertation. I would like to thank Pr. Claude Godart, Pr. Eugene Asarin and Dr. Christophe Rey for their precious participation in my thesis committee. Special thanks go to Pr. Mohand-Said Hacid for attending my viva, and being always helpful and present during the last years.

Thanks to my friends Nassim, Saci, Amine Mokhtari, Amine Defous, Hassane and Ziad for their support during the last decade. I am especially grateful to Pierre Colomb, my dear office mate and all the people of the D009 namely Julien Ponge, Olivier Coupelon and Yoan Renaud who were more than generous in offering their ideas and suggestions regarding many aspects of this work and for all the amazing moments and fun we have shared. Thanks guys! Thanks also go to Karima Toumani and Farid Chikhount for their making Clermont-Ferrand enjoyable place to live. I also want to express my thanks to Beatrice Bourdieu, Phillipe Mahey, Françoise Toledo, Daniel Breugnot and Nathalie Fonty. Finally, special thinks to all those people whose names are not listed here for their assistance and discussions on the subject matter.

I may take the opportunity to express my gratitude to all my teachers, and everyone who taught me anything. In particular, I would like to thank, and in a chronical order, Dr. Abdelkamal Tari and Pr. Moussa Kerkar.

Résumé

Les services web permettent l'intégration flexible et l'interopérabilité d'applications autonomes, hétérogènes et distribuées. Le développement de techniques et d'outils permettant la composition automatique de ces services en prenant en compte leurs comportements est une question cruciale.

Cette thèse s'intéresse au problème de la composition automatique de services web. Nous décrivons les services web par leurs protocoles métiers, formalisés sous la forme de machines d'état finies. Les principaux travaux autour de cette problématique se focalisent sur le cas particulier où le nombre d'instance de chaque service est fixé a priori. Nous abordons le cas général du problème de synthèse de protocoles où le nombre d'instances de chaque service disponible et pouvant intervenir lors de la composition n'est pas borné à priori. Plus précisément, nous considérons le problème suivant : *'étant donné un ensemble de n protocoles de services disponibles $P_1, \ldots, P_n$ et un nouveau protocole cible P_T, le comportement de P_T peut-il être synthétiser en combinant les comportements décrits par les protocoles disponibles?'*. Pour ce faire, nous proposons dans un premier temps un cadre formel de travail basé à la fois sur le test de simulation et la fermeture shuffle des machines d'états finis. Nous prouvons la décidabilité du problème en fournissant un algorithme de composition correct et complet. Ensuite, nous analysons la complexité du problème de la composition. Plus précisément, nous fournissons une borne supérieure et inférieure de complexité. Nous nous intéressons également aux cas particuliers de ce problème général. Enfin, nous implémentons un prototype de composition dans le cadre de la plateforme ServiceMosaic.

Mots clès Services web, Composition de services web, Automate shuffle, Simulation.

Abstract

Web services enable flexible integration and interoperability of autonomous, heterogeneous and distributed applications. A core challenge for the web services technology is the development of techniques and tools for automatically generating composite servicess by taking into account their behavioral properties (e.g. business protocols).

In this thesis, we focus on the problem of automatic composition of web services. We consider web services described by their business protocols which are formalized as finite states machines. Previous works on this problem dealt with the particular case where the number of instances of each component service is bounded and fixed a priori. We tackle the general case of the protocol synthesis problem where the number of instances of each component service that can be used in a composition is not bounded a priori. More precisely, we consider the following problem: *'given a set of n available web service protocols $P_1, \ldots, P_n$ and a new target protocol P_T, can the behavior described by P_T be synthesized by combining the behaviors described by the available protocols?'*. In order to cope with this problem, we first propose a formal framework for the composition synthesis based on both the simulation preorder and the shuffle closure of finite states machines. We prove its decidability through a sound and complete composition algorithm. Then, we conduct a complexity analysis of the composition problem. More precisely, we provide upper and lower bounds on the problem complexity. We also focus on several particular cases of this general problem. Finally, we implement a composition prototype within the framework of the ServiceMosaic platform.

Key words Web services, Web services composition, Shuffle automata, Simulation.

Contents

List of Figures

List of Tables

One

Introduction

1.1 Context

Web services. Application integration consists of linking wide-area, heterogeneous and distributed systems within and across enterprises [ACKM04]. It allows to simplify and to automate integration activities to the larger extent possible. However, such an integration requires adaptation of components (e.g. via middlewares) and turns out to be either resource or time consuming, or even not realizable [ACKM04].

Web services are gaining acceptance as a promising technology to deal with integration challenges such as application description, discovery and composition [HS05]. Web services provides *standardized* interfaces designed to evolve in Internet-based open environment. Indeed, the web services technology [1] relies on standardization at both the *messaging* and the *interfaces description* levels. The standardization layers enable simpler integration by going beyond heterogeneity issues among software components (e.g. data formats, transport protocols, and interface descriptions).

Web services description. The web services community proposes a multitude of standards to describe web services. These descriptions range from a

[1] http://www.w3.org/2002/ws/

just 'operations and messages' level (WSDL[WSD07]) to the business process level (BPEL[BPE07]) and may take into account additional issues (e.g. security and transactions). Indeed, a standard functional description of a web service should consider two main information : the *web service profile* and the *web service behavior*, commonly called business protocol[2]. The web service profile defines the set of operations supported by a service as well as the set of messages exchanged by these operations. In addition, pre-conditions on incoming messages and post-effects of operations executions could be considered. A web service business protocol defines the allowed sequences of messages (or operations execution) to be exchanged between partners. Thus, business protocols inform partners about which message to expect and when. For instance, a buyer might expect a confirmation message whereas the seller does not issue such a message and hence their interaction will not succeed.

Web services composition. The need for web services composition [BFHS03, DS05, BCG+05c, MW07, NM02, MS02] raises from the situation where none of the existing services can satisfy a client request, but a suitable combination of them would be able to do so. The research problems related to web services composition vary in nature and depend on several dimensions such as the composition process (manual vs automatic) and the model used to describe web services (profile or behavior). For more details, we refer the reader to a state of the art on the domain [DS05].

Automatic web services composition simplifies the development of software by reusing existing components and offers capabilities to customize complex systems built on the fly. It results in flexible applications with high reactivity to failures and dynamic adaptation to context. Moreover, automatic composition techniques have been developed to enable the verification of web services composition built either manually or automatically [Hul04]. However, composing services has been and still be a hard task to achieve. We can cite

[2]In this thesis, the terms business protocol and protocol -for short- will be used interchangeably.

many complexity sources, such as the large number of web services becoming available on daily basis over the web, the volatile nature of web services (may disappear, be modified, be temporary unavailable, etc.), and the diversity of conception models of services due to the modeling needs and to the developers vision.

In this thesis, we investigate the problem of *automatic synthesis of web services composition*. We focus on services described by business protocols and consider automatic generation of new composite protocols by combining available component protocols. The business protocols are formalized by means of finite state machines [HU69]. Recent works have shown the importance of such a formalism to describe the external behavior of web services [BCG+03, BCT04b, BCG+05c, BCT06b]. More precisely, we consider the following composition synthesis problem: *given a set of n available web service protocols $P_1, \ldots, P_n$ and a new target protocol P_T, can the behavior described by P_T be synthesized by combining (parts of) the behaviors described by the available protocols*. This composition synthesis problem has raised lots of research work [BCG+03, BDGL+04, HS05, BCG+05b, BCG+05c, BCG+05b, MW07, FGG+08].

The composition synthesis problem considered in this thesis has already been addressed in recent literature [BCG+03, BDGL+04, BCG+05b, BCG+05c, MW07] under the *restriction* that the *number of instances of an available service that can be involved in a composition is bounded by a constant k fixed a priori*. We call this restricted form of the composition synthesis problem *the (k-)bounded instances composition problem*. An instance of a web service is an occurrence effectively running. It should be noted that the restricted setting considered in the previous work is not realistic and has severe practical limitations that may hamper the usage of automatic service composition tools by organizations.

In figure 1.1 we illustrate briefly the composition synthesis problem and the practical limitations of the bounded instances case. We consider the avail-

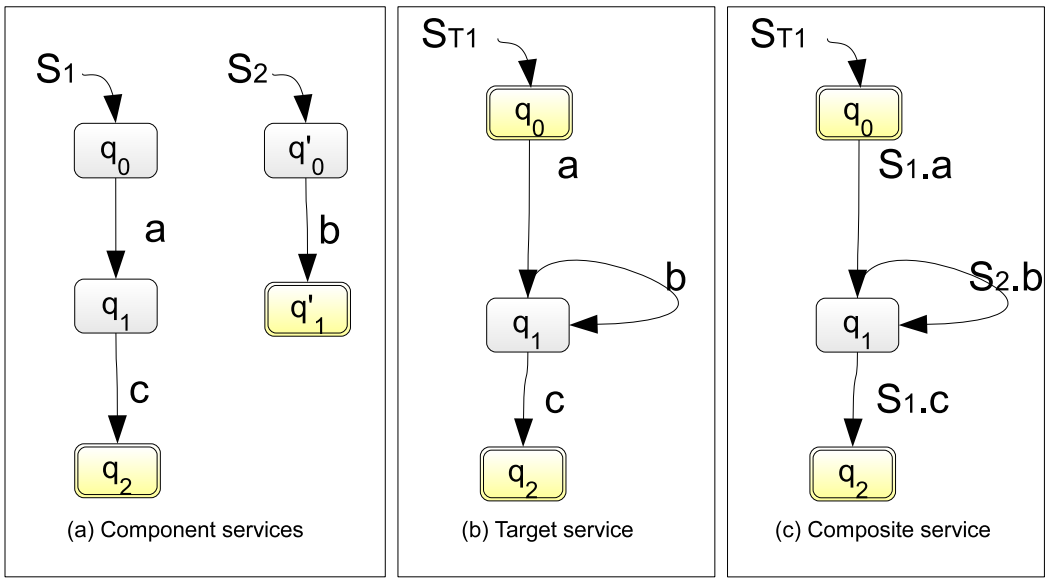

Figure 1.1: Example of composition synthesis problem

ability of two *component* services S_1 and S_2 (figure 1.1-(a)). The first allows to perform the operation a, then the operation c and ends at a final state. The second allows to perform the action b. The *target* service (figure 1.1-(b)) has the following behavior : it starts by performing a, and then allows a client to perform b as many times as he wants and to end by doing c. A rather simple *composite* service is depicted at figure 1.1-(c). This composite service relies on the invocation of a from S_1, then it invokes b from S_2 as many times as required and finally invokes c from S_1. Indeed, an execution of the composite service involves a unique instance of S_1 and an unbounded number of instances of S_2. It should be noted that the proposed composite service can not be generated within the settings of the bounded instances composition since an unbounded number of instances of S_2 is required.

The work of [BCG+03, BCG+05c] shows that the bounded instances composition problem can be reduced to that of testing the satisfiability of a Propositional Dynamic Logic [FL79] formula and proposes a *double-exptime* algorithm. Interestingly, in [MW07] the composition synthesis problem is reduced

to the problem of deciding whether there exists a simulation between the target protocol and the asynchronous product of available ones. The authors built on this reduction to prove the *exptime-completeness* of the bounded instances synthesis problem.

1.2 Contributions

This thesis investigates the unbounded case of the protocol composition synthesis problem [HNT08a, HNT08b]. We study the decidability and the complexity of this problem. We also implement a tool for automatic web services composition synthesis.

The first result was the decidability of the composition synthesis problem. Our proof relies on a terminating and a correct composition synthesis algorithm from which we derive an upper complexity bound. A lower complexity bound is provided as well. Moreover, we study the complexity of two particular cases of the composition problem. Finally, we evaluate the time complexity on a prototype over several synthetic benchmarks. In the following we will detail our contributions.

1.2.1 Formal framework for web services composition

In the same spirit as [MW07], we use a simulation-based framework to model the composition synthesis problem. We propose a new infinite state machine, called *Product Closure State Machine (PCSM)* [HNT08b]. The PCSM model is based on the notion of the shuffle closure [ORR78] which allows to run an unbounded number of asynchronous parallel instances of an FSM. Hence, the PCSM allows to describe a behavior equivalent to the one provided by all possible collaborations of the existing services. Furthermore, we model the problem of the composition synthesis as being the one of simulating a deterministic finite state machine (DFSM) by a PCSM.

The PCSM is a sub-model of both Shuffle Automata [Jed99] and Petri Nets [Pet73], two models widely used in literature but rarely connected. Indeed, we were able to describe all possible collaborations of the available component services using either shuffle automata or Petri nets. However, our choice to introduce the PCSM is justified by the flexibility that it offers to study the complexity issues. For instance, we derive easily the *NP-completeness* of a restricted case of the general composition synthesis problem. This particular form of the problem is characterized by target services without loops[3]. Moreover, we provide a survey on the existing results on the simulation preorder and various connected relations, namely language inclusion and bisimulation. The study of such relations is highly interesting with respect to the composition problem; as witnessed by [FGG+08] where language equivalence relation is used to compare services.

1.2.2 Decidability result

The protocol-based automatic composition synthesis problem is modeled as a simulation of a DFSM by a PCSM. We focus on the decidability of the unbounded case of the composition problem that was left open in recent literature [BCG+05b, BCG+05c, MW07]. The source of hardness in our proof comes from the fact that a PCSM is an infinite state machine, and hence a simulation relation may be of an infinite size. In order to solve this problem, we proposed a terminating and correct composition synthesis algorithm. The termination of this algorithm is based on the Dickson lemma [Dic13].

1.2.3 Complexity issues

A non-primitive recursive Ackermannian complexity upper bound is provided by our work, but the primitive recursiveness of our algorithm remains an open

[3]In this work, the term loop is used to characterize circuits of variant lengths (not necessery equal to 1).

question. It is worth noting that the existence of a primitive recursive algorithm is a hard question to answer. Indeed, the primitive recursiveness of many similar simulation problems [KJ06] is an open question from decades. For instance, we can consider the simulation of FSMs by Basic Parallel Processes [KM02a]. Moreover, an exp-hard lower bound on the composition synthesis problem is derived from the bounded case studied in the state of art [MW07].

Finally, we have conducted a complementary study of the complexity. We identified many particular cases of the composition synthesis problem. The first one considers target services without loops. We prove this case to be a NP-Complete problem. Another interesting case is the one where component services do not contain hybrid states. This case turned out to be exp-time.

1.2.4 Prototyping and performance evaluation

We evaluated the performances of the composition algorithm using a prototype implementation. This prototype took place inside the ServiceMosaic project[4], a model-driven environment for modeling, analyzing, and managing web services.

The experiments executed on the prototype was oriented by our aim at evaluating the effects induced by some parameters (e.g. the number of services and the height of nested loops) on the behavior of the composition algorithm. The choice of these parameters was based on theoretical observations (e.g. increasing the height of nested loops worsens the composition time). The conducted tests explain clearly how the variation of each parameter influences the execution time.

[4]http://servicemosaic.isima.fr

1.3 Outline

This thesis is organized as follows. Chapter 2 introduces the foundations of web services and their benefits in the area of applications integration. It overviews the notion of web services and their functional architecture as proposed by the W3C[5]. It then presents the web service business protocols model before looking in depth at different standards and formal models that allow to describe them. Chapter 3 provides the required material to understand the web services composition problem. In this chapter we analyze the main work achieved in the area. We concentrate on key dimensions and aspects that characterize this problem and we provide a panorama of the main proposed composition approaches in the literature. Chapter 4 defines the formal model of web services protocols and defines the composition synthesis problem that we are interested in. Chapter 5 discusses the theoretical contributions of this thesis. It starts by the definition of the Product Closure State Machine (PCSM); a state machine that allows to run an unbounded number of parallel asynchronous instances of a finite state machine. It then provides a composition algorithm that we show to be terminating and correct (i.e. the composition synthesis problem is decidable). Then we focus on the problem complexity. We end this chapter by a discussion on the relationship between our decidability results and existing ones. Chapter 6 is a state of the art on known decidability and complexity problems in the area of theory of automata and state machines. Chapter 7 presents the implemented prototype to compose web services and provides a detailed analysis of the obtained performances. Finally, chapter 8 concludes this thesis by anticipating on perspectives.

[5] http://www.w3c.org

Two

Web services

In this chapter, we will start by an overview of the web services basic notions and their interoperability architecture. We then review the web service business protocols before discussing the different languages and formal models proposed to describe them.

2.1 Basic notions

In this section, we first provide definitions of web services and then we present the standard interoperability architecture proposed by the World Wide Web Consortium(W3C)[1], a standardization organism aiming at the development of interoperability technologies (specifications, guidelines, software, and tools) to lead the Web to its full potential.

2.1.1 Definition

Web services provides standard-based application interfaces designed to evolve in Internet-based open environment. The web services technology aims at enabling the integration of autonomous, distributed and heterogeneous software

[1] http://www.w3c.org

systems, and to automate systems within and across organizations. Hence, it simplifies automatic machine-to-machine interoperability. A web service must be self described, and thus it enables loosely coupled integration of components. These components were not necessarily developed to collaborate together at the origin. In order to achieve such objectives, web services are based on a set of standards proposed by the W3C, as witnessed by the following definition:

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

This definition states also that a description of a web service must be machine-processable and expressed via WSDL specifications. The same holds for messaging and communication channels which relies on SOAP messages, over standard TCP/IP protocols such as HTTP.

Before investigating the web services architecture, we will provide a brief description of SOAP and WSDL.

2.1.2 Architecture

In this section, we will illustrate the web services interoperability architecture provided by the W3C. This architecture identifies a set of functional components and relationships between them. This architecture provides a conceptual model, and does not either specify how web services are implemented nor impose any restrictions on mechanisms to combine them.

A web service is an abstract interface which exposes a set of functionalities and must be implemented by one or many concrete agents. An agent is a piece of software or hardware that sends and receives messages.

The web services architecture relies on two entities: the provider and the consumer as shown in figure 2.1. The provider has the role of creating, advertising and hosting software applications described in a XML-based standard way, i.e. it must provide an agent that implements the web service. The service consumer retrieves somehow the web services, obtains their descriptions and interacts with them. A distinction is made between the agent which is a software module that interact with other agents and the entity, the person or the organization that provides agents and web services interfaces.

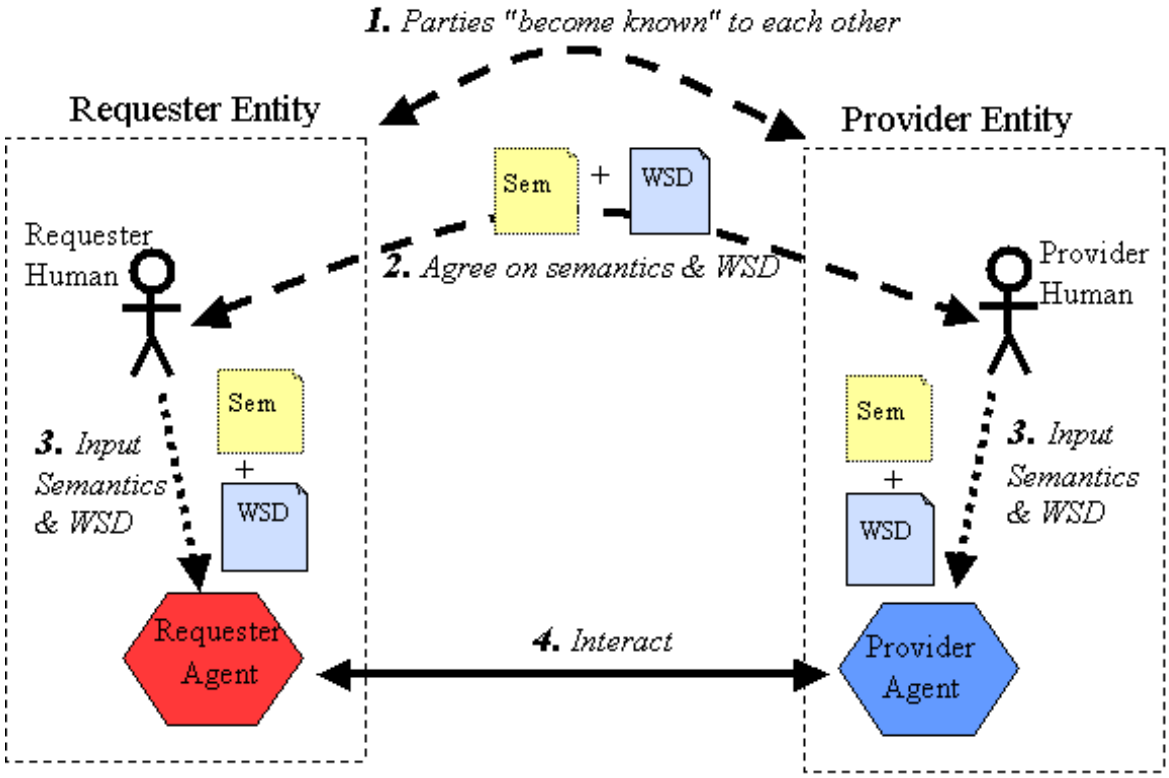

Figure 2.1: Standard web services architecture

At first, the requester and the provider become known to each other. Typ-

ically, the requester entity becomes somehow aware of the provider entity. This may happen in two ways : the requester entity knows the provider entity (e.g. human interaction) and obtains the provider agent address from him, or the requester entity makes use of some known *discovery* service to locate a suitable description of a provided service.

After being aware of each other, both requester and provider must agree on the web service *description* and *semantics*, two different description levels. The web service description represents an agreement, at syntactic level, on the mechanisms of interaction between services. It must be a machine processable specification of the web service interface. It defines low-level programming details, such as data types, messages, data formats and transport protocols. It also specifies one or many network bindings to one or many agents. The web service semantics is a step beyond the syntactic web service description. Semantics represent a contract on the purpose, the meaning and the consequences of the interactions and may be human or machine processable, implicit or explicit, legal or informal, etc..

Once agreed on web service description, and eventually semantics, both the requester and the provider entities implement and embody it into their agents as appropriate. This can be done, for instance, by hard coding. The agents are implemented and ready to interact, and thus SOAP messages exchanges can start.

Figure 2.2 [DS05] illustrates the implementation of the previous architecture as proposed by IBM[2]. This model consists of three partners : the service provider, the service consumer and the service registry. The provider has the role of creating services. He must also describe them in a XML-based standard way (i.e. WSDL [WSD07]). Then, he publishes them in a central service registry. The latter contains additional information about services, such as the address and the contact of the providing company (i.e. UDDI [UDD01]). The service consumer retrieves the information from the registry and uses the

[2]http://www.ibm.com

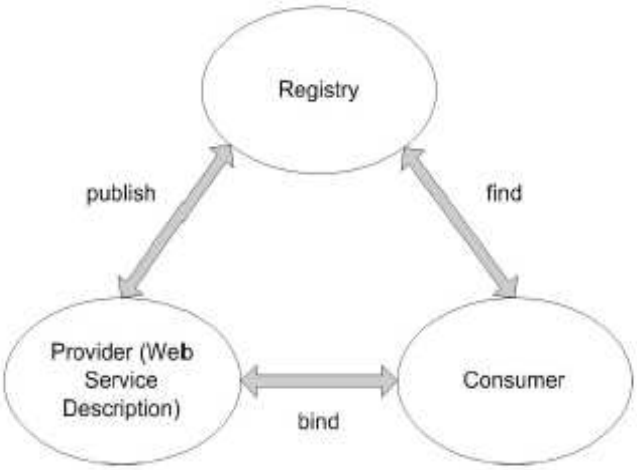

Figure 2.2: Web services architecture

service description obtained to interact with the Web service via messages exchange (i.e. SOAP [SOA07]).

SOAP

SOAP [SOA07] specifies the format of message for one-way communications whereas two-ways communications require SOAP to be combined with other protocols. It provides a set of conventions to implement Remote Procedure Calls (RPC) and a set of rules dedicated to allow entities exchanging the messages to define their own formats. It describes how to carry messages over standard Internet protocols, typically HTTP or SMTP. SOAP is completely defined by use of W3C XML schema recommendation.

SOAP arose from the difficulties encountered during application integration via open networks (e.g. Internet). We can cite for example : firewalls, the absence of a standard communication protocol between applications over the Internet, the need of loosely coupled interactions between software pieces and so on. Problems encountered when handling SOAP are mainly the standards limitations caused by the use of XML such as support to describe various

files formats (e.g. images). This fact raises a need for binary or blob formats. XML is resource consuming when transforming and parsing documents as well. Finally, we underline that SOAP is stateless and semantic-less.

Web Services Description Language : WSDL

Web services descriptions are defined via XML documents. WSDL [WSD07] provides an interface description of web services and plays the role that the Interface Description Languages (IDLs) had in conventional middlewares [ACKM04]. Indeed, WSDL allows to describe the set of operations (functionalities) provided by a web service and messages consumed and issued by these operations. Moreover, a WSDL specification must inform about binding mechanisms because a web service is not related to some existing middleware. This absence of middlewares requires information about service location. A great advantage of WSDL comes from the separation of interfaces and bindings of concrete services implementation which allows to match many implementations to an unique interface.

Universal Description, Discovery and Integration : UDDI

The goal of UDDI [UDD01] is to provide a central framework for web service description and discovery. It defines norms and rules to describe and publish services and to interrogate the registry. With regard to the discovery, two main issues are addressed. Firstly, UDDI aims at helping developers to find information about services to link, and secondly UDDI allows to discover services dynamically by allowing clients to browse the registry content. The ultimate existing reason of UDDI was the creation of an Universal Business Registry (UBR), a registry of all existing services in the world.

Pages inside UDDI are divided into three categories : white, yellow and green pages. The white pages are organization lists, with information about contacts as well as services they offer. Hence, clients can find services by

browsing organizations. The yellow pages adopt a classification of companies and services with regard to an existing taxonomy of activities which may be standard or customized. Hence, clients can find services by browsing activities. The green pages inform about how to invoke services, i.e. they point out WSDL descriptions.

2.2 Business protocols

In the first part of this section, we provide the definition of the abstraction behind business protocols. Also, we motivate the use of business protocols as web services description model. In the second part, we illustrate many existing proposals to model business protocols from formal and standardization standpoints.

2.2.1 Definition and motivation

The WSDL-based model assume low-level stateless web services interactions. This is by no mean a sufficient interaction model, as witnessed by [PG03, BCT04b, BCT04a, WSC, BCG+03]. These work raised a strong need to provide a higher-level statefull abstraction : *Business protocols*. This higher description layer specifies the set of conversations that the service can support, i.e. the valid sequences of operations execution (messages exchange).

Example 1 *Let us consider a simple example, with a private search web service that allows to perform one of the following operations : login, search, logout. Under realistic settings, it is easy to see that a valid conversation will be : (login, search... search, logout), i.e. a client must be logged to do as many 'search' as he wants, and finally must logout. Whereas, an invalid conversation is for instance : (logout, search, search), when a client starts by logging out and then performs many 'search'.*

The concern of web services protocols is the description of conversations supported by the service, i.e. its external behavior. The external behavior specifies clearly the messages or the documents to be exchanged with another web service during the interactions. Protocols do not care about the internal behavior of the services and does not inform about its implementation. By internal behavior we mean the application logic or private details of the operations execution, internal implementation and hidden mappings to partners from different enterprises the service make use of. Note that a given service may be implied in several conversations simultaneously and hence multiple *instances* of a same protocol can run concurrently.

In order to make the notion of business protocols clearer, we propose the following detailed example (figure 2.3).

Example 2 *One can imagine a select vehicle service with two operations : SelectVehicle and ModifySelection. The first operation allows a client to make a selection over a set of existing cars with some assumed options. The next permits him to change the car for some reason (e.g. because of its price). It is clear that a client can not change his selection before doing it ! Indeed, the information on the messages ordering will not appear on a WSDL description and the provider can expect many erroneous enactments when executing its proposed service. Figure 2.3(a) depicts the protocol of such a selection service. It informs the client to start by selecting a given car, and then keep modifying the selection until a suitable one is selected.*

Same as previous, figure 2.3(b) depicts the protocol of an hypothetical financing web service. The protocol specifies that the financing service is initially in the Start *state, and that clients begin using the service by executing the activity estimate payment, upon which the service moves to the* Payment Estimated *state (transition* EstimatePayment*).*

In our considered model, the initial state is indicated by an unlabeled entering arrow without source while final (accepting) states are double-circled

(figure 2.3).

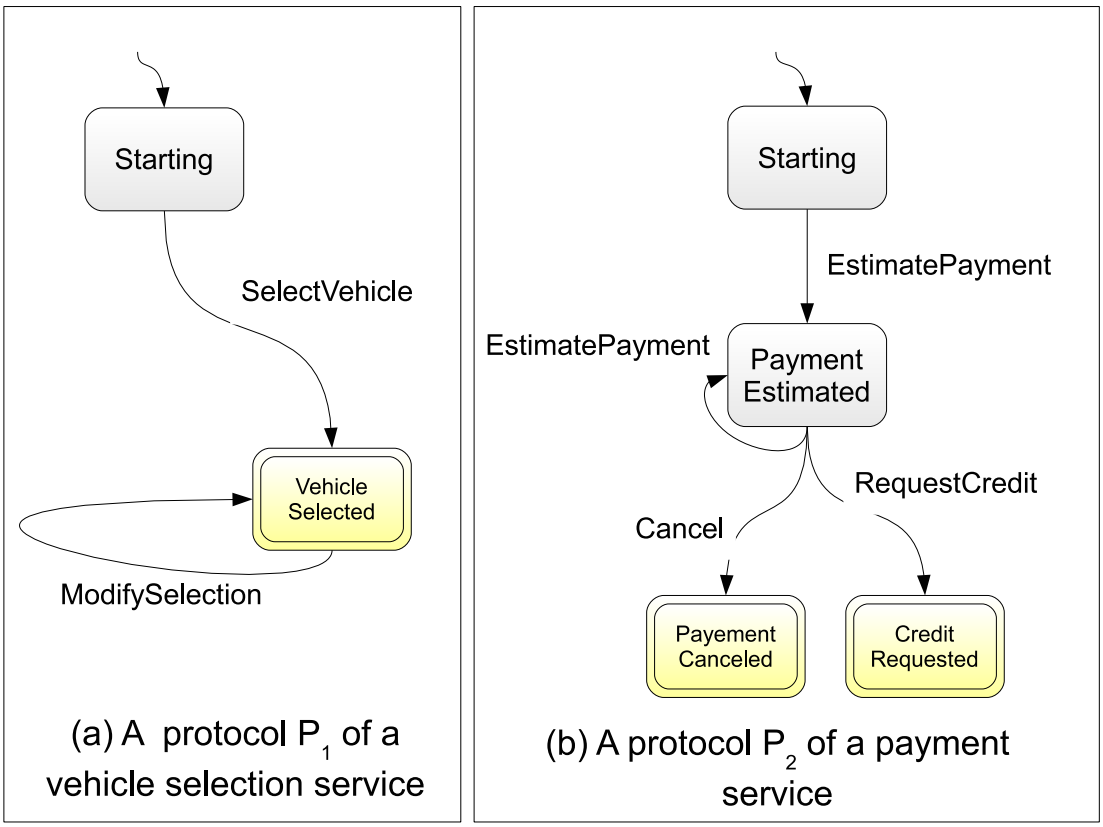

Figure 2.3: An example of web services protocols.

In the remainder of this section, we will present some existing efforts toward formal modeling and standardization of the web services business protocols.

2.2.2 Formal models

Hereafter, we present formal models of the business protocols that was proposed in the literature.

Automata-based model

The deterministic finite state-machines formalism is widely used to represent business protocols [BCT04a, BCG+05b, BCG+03, BFHS03]. States represent the different phases a service may go through during its interactions. Transitions are triggered by messages sent by the requester to the provider or vice versa. Each transition is labeled with a message name. A message corresponds to an operation invocation or its reply, and in other words, a message is the input or the output of an operation[BCT04a].

Reasons behind the use of finite state machines are that they represent a widely understood model, describe reactive behaviors suitably and own the state notion which is useful for monitoring executions[BCT04b]. These motivate us to consider FSMs as formal model for business protocols, and to build our work on the research of [BCG+03]. In the following we will briefly review some examples of use of FSMs as mean to model Business protocols.

In both [BCT04a, BFHS03], authors assume transition labeled by incoming and outgoing messages annotated by polarities "'!,?"' which indicates the direction of the message. The purpose of [BCT04a] is the analyses and the management of protocols. Among the studied analysis dimensions, we can cite the replace-ability and the compatibility of services which indicate respectively whether a service can replace an other for a set of enactments or whether two services can interact correctly by considering their protocols. The second work [BFHS03] deals with issues related to the composition of web services such as the modeling of composite services where component ones have a regular behavior. Even if both models rely on finite state automata formalism, a minor difference do exist, since [BFHS03] uses Mealy machines. Mealy machines are a particular form of finite state automata which is defined over a two alphabets, one for inputs and the other for outputs. Such a model fits naturally the concept of incoming and outgoing messages.

In [BCG+05b, BCG+03] authors label transition by abstract activities. An

abstract activity model a message exchanged or an operation execution. The underlying model is the traditional finite state automata as in [BCT04a]. The purpose of these efforts is the design of automatic algorithms for web services composition.

Petri nets

Petri nets are graphical and a formal modeling tool that allow to describe and to study information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel and nondeterministic [Mur89]. Petri nets are a much more expressive model than automata, but they lack the universal computing power of Turing machines.

In the context of web services composition, Petri nets are mainly used as a tool for execution monitoring [HB03, FLW06] and composition verification [MSZ01, YTX05, BH05]. They were also used to generate and to synthesize composite web services [MSZ01]. The drawback when using Petri nets is the high complexity and even the undecidability of most properties testing.

2.2.3 Emerging standards

As examples of existing standards to describe business protocols we will detail WSCL[WSC], OWL-S process model[OWL04] and BPEL[BPE07]. The choice of these standards is motivated by their wide impact on the literature and research approaches.

WSCL : Web Services Conversation Language

WSCL[WSC] is a proposal for a standard language submitted to the W3C in order to model business protocols for web services. It aims to be a simple standard conversation language that can be used by web service protocols and heterogeneous frameworks and platforms. It provides a specific conversation language and is located between simple state-less interface description

languages (e.g. WSDL) and more complex full-process description languages (e.g. BPEL) which describe conversations as well as application logic and internal implementations of the web service. WSCL allows the description of messages as done in WSDL and also their exchange sequences during interactions with other services.

Business Protocol Execution Language

Business Process Execution Language (BPEL) is a description and a composition language for business processes. It allows to implement a business process by defining the invocation order and detailing which operations from partners services must occur as well as data flow control. A fundamental concept inside the BPEL specification is the notion of partner links. It allows to describe relationship between partner processes, that is, web services that interact with each others. BPEL, as most programming languages, offers the possibility to store messages in variables. They can hold complex data, for instance SOAP messages from a partner, or simpler data which are only used internally. To cope with other standards, a BPEL process is exposed as a Web Service. Consequently, a BPEL process can be invoked by other BPEL processes transparently in standard way.

OWL-S process model

In order to achieve the needs in the area of semantic web services and mainly to allow services to reach a high-level expression descriptions, OWL-S (Ontology Language for Web Services)[OWL04] was proposed. OWL-S incorporate three essential types of knowledge : profile, process model and grounding. They can be presented as follows.

- The profile *presents* what the service requires and provides. It aims to describe the service as provided by the provider and needed by the client. Mainly, the profile informs about what organization provides

the service, what functions the service can compute. A function is an operation described by inputs, outputs, preconditions and effects.

- In OWL-S, a process transforms an input to an output and produces a transition an effect on the world state. Three types of processes are supported : atomic, simple and composite processes. An atomic process is directly invocable. A simple process is an abstraction of an atomic or a composite one. The later is built with atomic or composite processes by applying several operators such as : sequence, split, split+join, choice, if-then-else, etc.

- The grounding *provides* support about how to use the service. It presents information on communication protocols, message format, data serializations, transport and addressing. More precisely, grounding maps abstract specification of the service to his concrete implementation. The grounding is close to WSDL since the atomic processes are related to operations and the inputs/outputs are related to messages.

Three

Web services composition

Web services composition leads to a collaboration of services in order to achieve a task that can not be performed by using each service. Web services composition simplifies the rapid development of applications by enabling the reuse of existing components. The composition problem appears when a client's request could not be satisfied by any of the existing services on its own. Hence, it becomes necessary to combine them in order to create a new value-added composite web services. Indeed, the implementation of the business logic of a composite web service involves invocation of many component services. A web services composition is a recursive process since it applies on simple and on composite web services.

In this chapter we analyze the main work achieved in the area of web services composition. First, we concentrate on key dimensions and aspects that characterize web services composition problem. Then we provide a landscape of main composition approaches in the literature. Finally, we compare the studied approaches with respect to the previous key dimensions.

3.1 Dimensions of the composition

The problem of web services composition is quite complex and depends on several dimensions. The first dimension is the model used to describe web services. They may be defined through their behavior or their profile (or both) and may entail semantic aspects. Secondly, one can focus on the composition goals. Indeed, the composition domain covers various research fields (e.g. synthesis, validation or monitoring). A last question to ask is about how to compose services ? Thus, the composition process (manual or automatic) and the composition agility (static or dynamic) are important dimensions to take into account.

In the sequel, we discuss the aforementioned key dimensions that characterize the composition problem and the proposed approaches. We attempt to give the reader a clearer understanding of what the composition is about.

3.1.1 Services model : profile vs. behavior

This dimension allows to distinguish two models of services description. Indeed, most of existing efforts [BCG+03, BDGL+04, HS05, BCG+05b, BCG+05c, BCG+05b, MW07, FGG+08] can be categorized to deal with web services described by either their behavior or their profile (or both). The services behavior (web services protocols) represents an event driven or a control flow driven design. An event may correspond to a message exchange or to an execution of an operation (e.g. database access)[1]. The services profile deals with a data driven design where data correspond to the inputs and to the outputs of services (or more precisely, their operations) and the focus is made on data transformations via operations executions. Pre-condition on the input and (conditional) effects are often modeled.

Usually, component and composite services share the same model but it can happen that they differ. For instance, the roman model [BCG+03] consider

[1] An operation is commonly called abstract activity or atomic action

component and composite services as protocols. In [NM02], authors consider services described by their protocols where each transition is labeled by a couple (input, output). A target service is an output that the composer must produce by combining the outputs of existing services.

3.1.2 Composition semantics : semantic vs. syntactic

A web service description can consider semantics or only be limited to a syntactic description level. A syntactic description of a web service represents an agreement on the mechanisms of interaction between services. It must be a machine processable specification of the web service interface (e.g. WSDL or BPEL specifications). A semantic description represents a contract on the purpose, the meaning and the consequences of the interactions. It may be human or machine processable, implicit or explicit, legal or informal, etc.. Semantics description usually relies on ontologies [MSZ01]. Note that in many existing standards, separation between both description levels is flexible (e.g. OWL-S).

3.1.3 Composition goals

The composition process depends on the expected objectives (e.g. synthesis, verification and monitoring). The composition synthesis (e.g. [BCG+03]) consists in generating specifications of composite services by combining existing services . The composition verification (e.g. [NM02, HB03]) aims at establishing whether a web service upholds specified properties (e.g., that it ensures safety) . The monitoring (e.g. [PBB+04]) takes place at run-time and allows to analyze various execution parameters (e.g. time or cost) .

3.1.4 Composition process : manual vs. automatic

Manual composition is based on human intervention and deals with low-level programming and implementation issues. Examples of manual composition environments include BPEL [BPE07] and Microsoft BizTalk [2]. The manual composition is usually complex, does not scale and is error prone. Automatic composition simplifies the development of composition specifications by reusing existing components, and offers capabilities to customize complex systems built on the fly. This results in flexible applications with high reactivity to failures and dynamic adaptation to context changes. Moreover, automatic composition techniques have been developed to enable the verification of web services compositions computed either manually or automatically. However, automatic composition has been and still be a hard task to achieve. We can cite various sources of complexity, such as the difficulty (e.g. the decidability or the tractability?) of the composition depending on the expressiveness of the services model and the composition goal, the large number of web services over the web, and the diversity of the conception models of services due to the modeling needs and/or to the developers vision.

3.1.5 Composition agility : static vs. dynamic

The static composition takes place at design-time of an application. Thus, involved components are chosen, linked and assembled together before being deployed [DS05]. Such a composition is suitable for closed environments where components do not frequently evolve. Dynamic composition takes place at run-time and allows to autonomously create complex services by combining components on the fly based on user requests and context [FS04]. It evolves in flexible, open environments where selecting and combining components are done on demand. The dynamic composition technology is usually challenged by the large number of services becoming available on a daily basis, the volatile

[2] http://www.microsoft.com/biztalk/en/us/overview.aspx

nature of web services (e.g. they may disappear, be modified or be temporary unavailable), and the continuously growing number of services providers.

3.2 Existing composition research efforts

In this section, we will provide an overview of existing web services composition synthesis approaches. We focus on approaches related to the problem we are concerned by, i.e. automatic composition synthesis of web services described by their behavior (business protocols).

3.2.1 The Roman model

In [BCG+03], a web service is modeled by its protocol where a protocol is given by means of a deterministic finite state machine (DFSM). Transitions are labeled by the activities that a service can perform. The states indicate the phases that a service can go through and final states indicate correct haltings. The authors assume an existing finite community of services, which will be used to compose the target one. Services from a community share a common alphabet (set of activities). A target service is described as a protocol over the common alphabet. The composition problem consists in synthesizing a new composite protocol which delegates all of its activities to services from the community. The composite services act exactly as the target one from a client point of view. The concept of delegation stands for the fact that the composite service does not run any activity on its own, but makes an invocation of this activity from an existing service.

The authors reduced the composition synthesis problem to the one of the satisfiability of a deterministic propositional dynamic logics (DPDL [Eng67]) formula. Hence, a composition exists if and only if the formula is satisfied. Authors provide 2-exptime complexity bound for their problem. An exptime-

complete complexity bound on the same problem has been provided later by [MW07].

The composition synthesis approaches considered above [BCG+03, MW07] assume the strong restriction that *the number of instances of an existing service that can be involved in a composition is bounded and fixed a priori by a constant k*, i.e. the bounded instances composition problem. It should be noted that the restricted setting considered in these work is not realistic and has severe practical limitations that may limit the usage of automatic service composition tools by organizations. For more details, we illustrate in section 4.3.3, some very simple cases of web service composition cannot be solved in such a restricted setting.

3.2.2 The conversational model

In [BFHS03], the authors provided a framework for modeling and specifying the behavior of web services. They proposed a new approach for the design and the analysis of composite services. In this framework, an individual web service is called a peer. The peers communicate through message exchanges and each peer has a queue that stores incoming messages (a state of affair that enables asynchronous communications). The model assumes the existence of a global virtual watcher that keeps track of exchanged messages as they occur and a conversation is a sequence of messages observed by the watcher. A composite service is characterized by the whole set of conversations obtained by the interaction of its components.

The peers are represented by Mealy machines [Mea55], i.e. finite state transducers that generate an output based on its current state and an input. Mealy machines are an equivalent model to finite state automata. Surprisingly, the set of conversations behaves in an unexpected irregular way. For example, one can exhibit composite web services based on Mealy machines whose set of conversations is neither regular nor context-free but context-sensitive. This is

due to the ability of peers to enqueue messages.

To cope with this problem, the authors introduce two operators : prepone and projection-join. Thus they focus on peers whose conversations sets are regular languages modulo the both operators, i.e. the set of languages that after application of both operators are regular. The prepone operator consists in swapping messages in a conversation when they are independent (e.g. disjoint senders and receivers). The projection-join consists on projecting the conversations on individual peers and then on joining the obtained parts. In this context, the authors consider the following composition synthesis problem. The inputs of the problem are *(i)* a desired global service specified as a regular language L and *(ii)* a set of peers and the messages they can exchange. The output is the specification of a state machine whose conversations set is equal to L. Indeed, this state machine is a Mealy one due to the presence of the prepone and the projection-join operators.

It should be noted that the focus is done on the words (conversations) without considering the branching structure of the Mealy machine as done in the previous works [BCG+03, MW07, BDGL+04].

3.2.3 Composing web services in Colombo

Colombo [BCG+05b, BCDG+05, BCG+05a] was proposed as a composition model that merges the aforementioned transitional behavior (the Roman model) and messaging (the conversational model). More precisely, a service is characterized in terms of

1. the set of atomic processes (i.e. operations) it can perform,
2. its effects on the real world described as a relational database $\mathcal{R}$,
3. its transition-based behavior and
4. the messages it can send and receive.

The transitional based behavior of component services is described by means of deterministic finite state machines. A transition is labeled by an activity, i.e. an atomic process, a send-message or a receive-message. An atomic process performs an internal operation of the service and handles the state of the real world $\mathcal{R}$. An atomic process may also include a conditional effects, where conditions depend on both the state of $\mathcal{R}$ and the received messages and the effect itself is an update query issued on $\mathcal{R}$. A target service is modeled in the same way as component services while the composition synthesis problem IS the same as the Roman model.

The decidability of the composition synthesis problem remains an open question for most cases of the general Colombo framework. Hence, authors define a sub-model of Colombo, called $Colombo^{k,b}$ where both the complexity and decidability are provided. The assumptions can be summarized as follows :

1. in any execution of th delegator, only a finite number of values of the domain of $\mathcal{R}$ are read,

2. concurrency when accessing $\mathcal{R}$ and messages exchanging is prevented,

3. all messages exchanged during the execution of the delegator are send or received by it and finally

4. all delegators are $(p, q)-$bounded, i.e. the finite state machine associated with the delegator has at most p states and at most q variables values in its store. This store allows the mediator to queue received messages.

Note that the latter assumption implies a bound on the number of instances that can be used when synthesizing compositions. Finally, the complexity of the composition synthesis in $Colombo^{k,b}$ is 2-exptime and the proof technique is based on a reduction to the DPDL formulas satisfiability.

3.2.4 Synthesized web services

In [FGG+08], authors propose a new model, called synthesized Web services (SWSs), to describe the web services behavior. An SWS is a quite complex finite state machine. An SWS is defined by a finite set of states (Q), an initial state (q_0), a transition function (δ) and a synthesis function (σ). A transition from δ is labeled by a query from $\mathcal{L}_{Msg}$, a query language defined over a specific class of logic (e.g. first order logic). The synthesis function is defined over $\mathcal{L}_{Act}$, an action synthesis language defined over a specific class of logic. Each state q has two local stores $Msg(q)$ and $Act(q)$ in order to keep respectively a message and an action on q.

Basically, an SWS receives a sequence of messages and produces a set of activities depending on the real world instance formalized as a relational database $\mathcal{D}$. The input corresponds to a client request, and the output is the synthesized actions (a logic formula over a set of simple actions) that the client may perform to satisfy his need. Upon receiving a sequence of input messages, an SWS proceeds through two phases : downward (using δ) and upward (using σ). During the downward phase, given an input message from the sequence, a state q and $\mathcal{D}$ then the transition function δ updates the set of messages ($Msg(q')$) of each successor of q. This phase halts in a state q if it has *(i)* an empty transition rule or *(ii)* an empty message ($Msg(q) = \epsilon$) or *(iii)* the input sequence is completely consumed. Note that initially all the $Act's$ of states are empty and that the downward phase algorithm generates a tree $\mathcal{T}$ of depth equal to the size of the input sequence. The upward phase starts by generating an action on each leave q of $\mathcal{T}$ depending on $Msg(q)$ and D. Then, the upward phase algorithm climbs in a bottom-up fashion and generates for each node q of $\mathcal{T}$, the action $Act(q)$ depending on σ and on the set $\{Act(q'),\ where\ q'\ is\ a\ successor\ of\ q\}$. The output of this SWS execution is $Act(q_0)$ which corresponds to the synthesized action that fits the client needs.

Authors focus on four decision problems described below :

1. The validation problem : given a service and a conversation, one wants to know whether the conversation belongs to the set of the valid ones generated by this service. The validation is useful for, e.g., fraud detection, compatibility checking.

2. The equivalence problem : aims at determining whether two given services are equivalent, i.e. they support the same set of valid conversations. The equivalence is useful for, e.g., replace-ability checking.

3. The non-emptiness problem : one can be interested in finding out, at compilation time, whether or not a service makes sense, i.e., whether or not it can generate valid conversations.

4. The synthesis problem : aims at determining, given a target service and a set of available services, whether there exists such a mediator (delegator) that coordinates available services (by routing the output of one service to the input of another) in order to deliver the target service by invoking available services as component services.

Authors deal with several classes of SWSs characterized by the class of logic used to describe L_{Msg} and L_{Act}. A class is denoted by $SWS(L_{Msg}, L_{Act})$, where L_{Msg} and L_{Act} range over propositional logic (PL), conjunctive queries (CQ), union of conjunctive queries (UCQ) and first-order logic (FO). Indeed, the complexity of the decision problems highly depends on the SWS class within which web services are defined. As a main contribution of this paper, authors established lower and upper bounds on these decision problems for the several $SWS(L_{Msg}, L_{Act})$ classes. The results are established by, among other techniques, exploring connections between composition synthesis and (equivalent) query rewriting using views [CDGLV99, AGK99].

Interestingly, authors characterized the existing services models to belong to two categories. The first category of models specifies web services behavior

by means of finite-state machines (FSM) (e.g. the Roman model [BCG+03]). Other models are based on data-driven finite state machines (transducers) that generate an output depending on the input and the current state (e.g. Colombo [BCG+05b] and the conversational model [BFHS03]). Furthermore, they showed that finite state machines of the Roman model can be expressed in SWS(PL, PL), while data-driven finite state machines of Colombo can be expressed in SWS(FO, FO).

The considered composition synthesis problem takes in input an existing set of SWSs and a target one. The composition aims at generating a mediator that behaves as the target SWS. It coordinates SWSs by routing the output of one service to the input of another one. The mediator receives and redirects messages but does not access $\mathcal{D}$. It is worth noting that the composition synthesis problem considered here is slightly different from the one considered in the previously cited work [BCG+03, MW07, BFHS03, BCG+05b, BCDG+05, BCG+05a]. In fact, the SWS-based synthesis do not allow transitions from component services to be interleaved. For instance, if we consider a service S_1 that supports the unique conversation ab and a service S_2 that supports the unique conversation c. The obtained conversation with interleaving are $\{abc, acb, cab\}$, where the obtained ones without interleaving are only $\{abc, cab\}$. This fact prevents the results provided by this paper to be directly applied for our context, which is the same as the Roman model.

3.2.5 Graph-based composition

In [ZAAM03], authors proposed an original graph-based framework to compose services. A web service corresponds to a set of inputs, a set of outputs and a set of dependencies. A dependency (I, S, O) means that *from the input I, an operation of the service S produces the output O*. Indeed, for each input there exists at least one dependency that exploits it. Then, a services community is described by a *community graph* where the vertices represent

the inputs/outputs from the different services and the edges where each edge is labeled by a service name and represents a dependency. The target service is described in the same way expect the fact that its dependencies are without services names (i.e. dependencies of the form *(I,?,O)*). The composition consists of finding a set of paths in the community graph where each path is equivalent to a dependency of the target service. A path is said to be equivalent to a dependency if both of them start by the same input and end by the same output. The composition approach consists of two phases : component selection and composition configuration. The first phase aims at finding an equivalent dependency in the community graph for each dependency from the goal service description. This phase is solved by a first branching spread algorithm. The second phase combines the computed paths in order to give a sort of orchestration model for the composite service. This work gives an interesting intuition behind the use of graphs to compose services. However, it suffers from two major drawbacks : *(i)* composite inputs or outputs are not taken into account (e.g. $((i_1, i_2), S, o)$) and *(ii)* there is no mean to express preconditions and post-effects on operations.

3.2.6 Semantic-based composition

A slightly different way to tackle the composition synthesis arises from various semantic-based efforts [MJL07, PMBT05, PTB05, GT04]. For instance, in [NM02] authors propose a model where component services are described by a DAML-S (currently OWL-S) description. More precisely, a component service is treated as atomic, i.e. has a unique operation identified by its IOPE (Input, Output, Pre-conditions, Effects) signature. These services are then translated in a Petri nets formalism which allows to perform the formal analysis of interest in the paper. The authors deal with the simulation, the verification and the automated composition problems. In order to realize the composition, the authors provides the definition of a net that depicts the behavior of all the

services, i.e. the union of the nets modeling each service.It should be noted that under the considered settings (atomic component services), no possible interleaving among services operations can be considered by any mean. In contrast of the previous works from this section, the target service does not model a behavior to synthesize but rather an output to produce. In fact, this output corresponds to a suitable combination of the different outputs of the component services (e.g. the output of the service S_1 is the name of the person, the output of the service S_2 is the address of the person whilst the target service consists of generating the output *name and address*).

In [SPW+04], authors considered web services presented by the OWL-S process model. They proposed and proved the correspondence between the semantics of SHOP2 (a hierarchical task network planner) and the situation calculus semantics of the OWL-S process model. They provide a sound and a complete algorithm to plan over OWL-S description using SHOP2 planner. However, they made their work under two assumptions concerning the services model due to limitations of the involved planner. Firstly, an atomic process (a single step web service) can have either output or effect but not both at once. Secondly, non-atomic processes can not use the *Split*[3] and the *Split+Join*[4] control structures. It should be noted that the target and composite services are described by OWL-S processes as well however no interleaving is considered as well.

3.3 Comparison of existing approaches

Table 3.1 summarizes the results about current approaches discussed in the section 3.2. There efforts are compared on basis of the dimensions analyzed in section 3.1

[3]In OWL-S process model, the *Split* control structure leads to a concurrent execution of a bag of sub-processes.

[4]In OWL-S process model, the *Split+Join* control structure engenders a concurrent execution of a bunch of sub-processes with barrier synchronization

	CG	**CSM**	**TSM**	**Sem**
[BCG+03]	Synthesis	Behavior	Behavior	No
[BDGL+04]	Synthesis	Behavior	Behavior	No
[BFHS03]	Synthesis & verification	Behavior	Behavior	No
[BCG+05b]	Synthesis	Behavior	Behavior	Low
[FGG+08]	Synthesis, validation, equivalence and non-emptiness	Behavior	Behavior	No
[ZAAM03]	Synthesis	Profile	Profile	No
[NM02]	Synthesis, simulation and verification	Behavior	Profile	Yes
[SPW+04]	Synthesis	Behavior	Behavior	Yes

Table 3.1: Comparison of existing composition synthesis work.

In the table 3.1 CG, CSM, TSM and Sem mean respectively Composition Goals, Component Service Model, Target Service model and Semantics. The two first dimensions, namely composition Process and composition Agility are not illustrated on the table since all works are qualified to be automatic and dynamic.

Four

Protocol-based web services composition

This chapter deals with the problem of composition synthesis of web services described by their protocols. It contains the first contribution of this thesis. It provides a generic definition of protocol synthesis problem that cater for both cases where the number of instances that can be used in composition is bounded or unbounded. The former case being widely investigated in literature [BCG+03], we concentrate on the unbounded case of the composition. We formalize it using the simulation preorder and the shuffle closure operator. More precisely, we show that it can be reduced to the simulation of a deterministic finite state machine by an infinite state machine which corresponds to the shuffle closure of a finite state machine.

This chapter is organized as follows. The first section introduces basic notions and preliminaries that will be useful in the remainder. Then, we detail the formal model of web services protocols used in this work. Finally, we propose a simulation-based formalization of the composition synthesis problem that we are interesting in.

4.1 Preliminaries

In this section, we first recall the notion of state machines (SM) [HU69]. Then, we present the simulation preorder[1] between state machines. Such a preorder is usually used to compare state machines basing on their behavior [KJ06]. We also introduce two unusual operators on state machines, namely iterated product and product closure. As we will see later, these operators turned out to be useful to model an unbounded number of asynchronous parallel instances of a finite state machine. Finally, we recall the definition of the Ackermann function [Wic76].

Definition 1 ***(State Machine (SM))***
A State Machine M is a tuple $< \Sigma_M, Q_M, F_M, q_M^0, \delta_M >$, where :

- Σ_M *is a finite alphabet,*
- Q_M *is a set of states,*
- $\delta_M \subseteq Q_M \times \Sigma_M \times Q_M$ *is a set of labeled transitions (actions),*
- $F_M \subseteq Q_M$ *is the set of final states, and*
- $q_M^0 \in Q_M$ *is the initial state.*

If Q_M is finite then M is called a Finite State Machine (FSM). An FSM M is deterministic (DFSM) iff δ_M is a function rather of being a relation.

We define below the notions of intermediate and hybrid states of an FSM and the notion of the norm of a state and an FSM. Let $M =< \Sigma_M, Q_M, F_M, q_M^0, \delta_M >$ be an FSM. Then: *(i)* the set of *hybrid states* of M, noted $H_s(M)$, contains all the final states of M that have at least one outgoing transition, and *(ii)* the set of *intermediate states* of M, noted $I_s(M)$, contains

[1] A preorder is a reflexive and transitive relation.

the states of $\{Q_M - F_M\}$ that have at least one incoming and one outgoing transitions. Let $q \in Q_M$, then the norm of q, noted $Norm(q)$, is the length of the shortest path leading from q to a final state. The norm of an FSM M, noted $Norm(M)$ is the maximal norm of its states.

Example 3 *The intermediate states of the FSM M depicted at figure 4.1(a) are* s_1 *and* s_2 *(i.e.* $I_s(M) = \{s_1, s_2\}$*), while its set of hybrid states is empty (i.e.* $H_s(M) = \phi$*). The norm of* s_0 *is* 2 *(i.e.* $Norm(s_0) = 2$*) and the norm of* s_3 *is* 0 *(i.e.* $Norm(s_3) = 0$*). The norm of M is* 2 *(i.e.* $Norm(M) = 2$*).*

We provide below a definition of the simulation preorder between two SMs.

Definition 2 ***(Simulation preorder)***

Let $M =< \Sigma_M, Q_M, F_M, q^0_M, \delta_M >$ *and* $M' =< \Sigma_{M'}, Q_{M'}, F_{M'}, q^0_{M'}, \delta_{M'} >$ *be two state machines. A state* $q_1 \in Q_M$ *is simulated by a state* $q'_1 \in Q_{M'}$*, noted* $q_1 \preceq q'_1$*, iff the following two conditions hold :*

1. $\forall a \in \Sigma_M$ *and* $\forall q_2 \in Q_M$ *such that* $(q_1, a, q_2) \in \delta_M$ *there is* $(q'_1, a, q'_2) \in \delta_{M'}$ *such that* $q_2 \preceq q'_2$*, and*

2. *if* $q_1 \in F_M$*, then* $q'_1 \in F_{M'}$*.*

M is simulated by M′, noted $M \preceq M'$*, iff* $q^0_M \preceq q^0_{M'}$*. M and M′ are simulation equivalent, noted* $M =_{sm} M'$ *iff* $M \preceq M'$ *and* $M' \preceq M$*.*

Occasionally in the present thesis, we will need to use FSMs with ϵ-transitions[2]. In order to compute simulations over these FSMs, the first condition of the previous definition is adapted as follows: $\forall a \in \Sigma_M$ and $\forall q_2 \in Q_M$ s.t. $(q_1, a, q_2) \in \delta_M$ there is $((q'_1, a, q'_2) \in \delta_{M'}$ s.t. $q_2 \preceq q'_2)$, **OR** there are $((q'_1, \epsilon, q'_2) \in \delta_{M'}$ and $(q'_2, a, q'_3) \in \delta_{M'}$ s.t. $q_2 \preceq q'_3)$.

[2]The considered FSMs with ϵ-transitions can have sequences of ϵ-transitions of length less or equal to 1

Existing work on simulation between finite state machines provide several algorithms[3] as proposed in [Blo89] ($O(m^6)$), [CPS+91] ($O(mn^4)$), [CS93] ($O(m^2)$) and finally [HHK95] ($O(mn)$ with $m > n$) where n is the total states number of both machines and m is the total transitions number of both machines. The simulation between finite state machines and infinite state machines range from *exptime-complete* [KM02c] to undecidable [KM02a] depending on the considered infinite state machine (e.g. Petri nets [Pet73]). Concerning simulation between infinite state machines, the only known class where simulation preorder/equivalence remains decidable are one-counter nets [AC98].

Interestingly, simulation testing can be seen as a game [Tho93, Sti98] between an attacker and a defender. Let $M =< \Sigma_M, Q_M, F_M, q^0_M, \delta_M >$ and $M' =< \Sigma_{M'}, Q_{M'}, F_{M'}, q^0_{M'}, \delta_{M'} >$ be two state machines, $m \in Q_M$ and $m' \in Q_{M'}$. In a simulation game the attacker wants to show that ($m \not\preceq m'$), while the defender attempts to frustrate this [KM02b]. The game starts by two tokens, one on each state. The game is formed of many successive rounds where each round is performed as follows : the attacker takes his token from m and moves it over a transition from δ_M labeled by a, and the defender must move the other token along a transition from δ'_M with the same label. Indeed, if the attacher moves to a final state, the defender must do so. The attacker wins if the defender can not move (i.e. there is no simulation). The defender wins if the attacker can not move or if the game is infinite (i.e. there is simulation). We illustrate this game principle on the following example.

Example 4 *We consider the two state machines M and M' illustrated on figure 4.1 and we apply the gaming principle to show that M is simulated by M' but not the reverse. To do so, we must check the simulation between the initial states q^0_M and $q^0_{M'}$ in both directions, i.e. $q^0_M \preceq q^0_{M'}$ and $q^0_{M'} \preceq q^0_M$. Hence, we start a game row with two tokens, one on each initial state.*

[3]The algorithms are cited in their chronological order of apparition and concern SMs without ϵ-transitions.

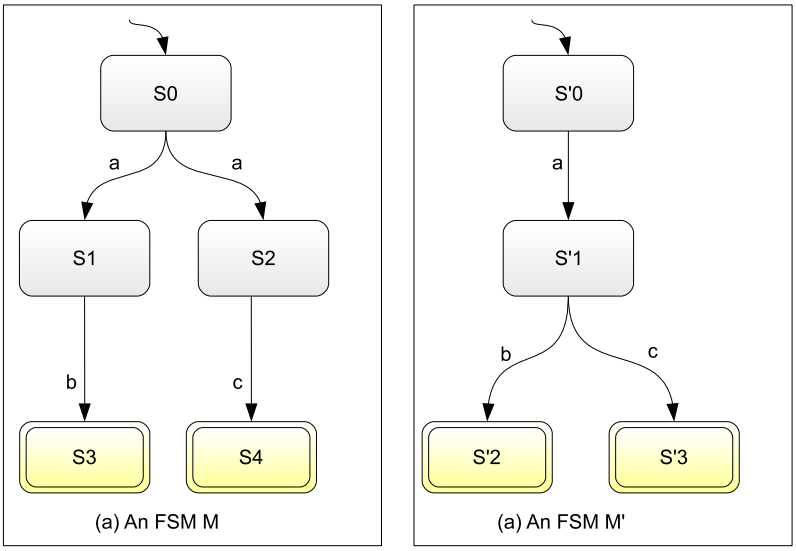

Figure 4.1: An example of the simulation of two state machines.

- ***Simulation of M by M'*** *If the attacker decides to move his token to s_1 or s_2 by doing an $'a'$ then the defender will move to s'_1. After, the attacker can either move to one of the final states s_3 or s_4 by doing respectively $'b'$ or $'c'$. In both cases, the defender will move respectively to s'_2 or s'_3 which are also final. The attacker has no more possible moves, then the defender wins, i.e. a simulation exists.*

- ***Simulation of M' by M*** *When the attacker moves his token to s'_1 along the transition labeled by $'a'$, the defender moves to either s_1 or s_2. If the defender chooses s_1, the attacker will move to s'_3 along the transition labeled by $'c'$, and the defender can not perform an equivalent move. If the defender chooses s_2, the attacker will move to s'_2 along the transition labeled by $'b'$, and the defender can not perform an equivalent move. Hence, the defender has no winning strategy and the attacker wins, i.e. there is no simulation.*

It should be noted that the simulation preorder and the language inclusion order[4] have close semantics and usually connected in literature. A simulation is a stronger relation than a language inclusion because a simulation implies a language inclusion, but the reverse does not hold. For instance, in figure 4.1, both M and M' are language equivalent ($L(M) = L(M') = \{ab,\ ac\}$) but $M' \not\preceq M$. More precisely, a simulation between two machines M_1 and M_2, ensures *(i)* an inclusion of the language of M_1 ($L(M_1)$ in the language of M_2 ($L(M_2)$) and *(ii)* that state of M_2 have at least the same branching structure of the states of M_1. It informs that M_2 can behave exactly as M_1.

We provide below a definition of the asynchronous product (shuffle product or product for short), iterated product and product closure [WH84].

Definition 3 ***(Product and iterated product)***

Let $M =< \Sigma_M, Q_M, F_M, q^0_M, \delta_M >$ *and* $M' =< \Sigma_{M'}, Q_{M'}, F_{M'}, q^0_{M'}, \delta_{M'} >$ *be two FSMs. Then :*

1. *The* ***product*** *of* M *and* M'*, denoted* $M \times M'$*, is an FSM* $< \Sigma_M \cup \Sigma_{M'}, Q_M \times Q_{M'}, (F_M \cup q^0_M) \times (F_{M'} \cup q^0_{M'}), (q^0_M, q^0_{M'}), \lambda >$ *where the transition function* λ *is defined as follows:* $\lambda = \{((q, q'), a, (q_1, {q_1}')) : ((q, a, q_1) \in \delta_M \text{ and } q' = q'_1) \text{ or } ((q', a, {q_1}') \in \delta_{M'} \text{ and } q = q_1)\}$.

2. *Given an integer* k*, the* ***k-iterated product of*** M *is defined by* $M^{\otimes_k} = M^{\otimes_{k-1}} \times M$ *with* $M^{\otimes_1} = M$.

3. *The* ***product closure*** *of* M*, noted* $M^{\otimes}$*, is defined as follows:* $M^{\otimes} = \bigcup_{i=0}^{+\infty} M^{\otimes_i}$ *(where* $\bigcup$ *stands for the union of FSMs).*

Figure 4.2 illustrates the computation of the product of two FSMs S_1 and S_2. Note that in our work, and w.l.o.g, we use FSMs without loops on initial states as it will be proved in section 5.1.

[4] An order is reflexive, anti-symmetric and transitive relation.

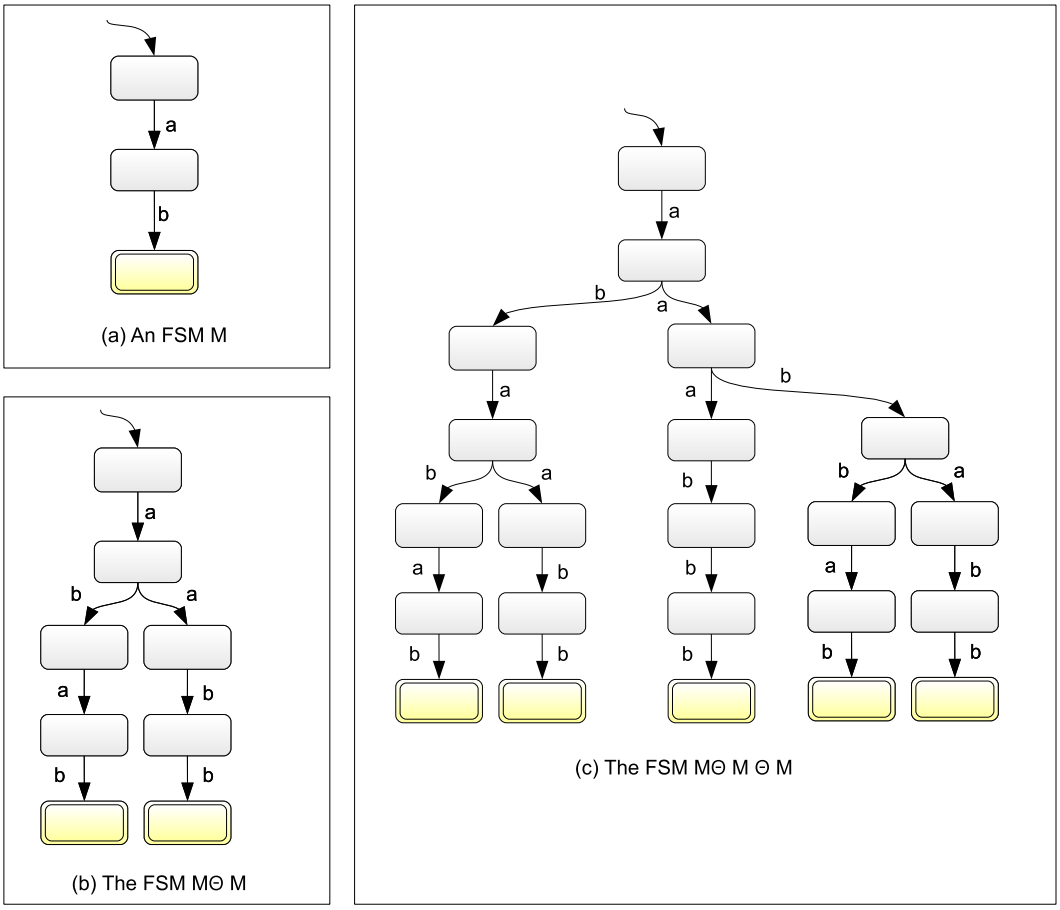

Figure 4.2: An example of a product of two FSMs

It is worth noting that for any finite positive integer k, the k-iterated product $M^{\otimes_k}$ of an FSM M is still an FSM (more precisely, $M^{\otimes_k}$ has $|Q_M|^k$ states). However, this property does not hold for $M^{\otimes}$. We introduce the following notations : let $\mathcal{R} = \{P_1, \ldots, P_n\}$ be a set of FSMs, then $\odot(\mathcal{R})$ denotes *the union of the asynchronous product of all the subsets elements* of $\mathcal{R}$, i.e., $\odot(\mathcal{R}) = \bigcup_{\{P_{i_1},\ldots,P_{i_m}\}\subseteq\mathcal{R}}(P_{i_1} \times \ldots \times P_{i_m})$ where $m \in [0, n]$. Let A and B be two state machines then we have *(i)* $A \subseteq B$ iff there exists an isomorphism $\mathcal{H}$ from A to a sub-part of B and *(ii)* $A = B$ iff $A \subseteq B$ and $B \subseteq A$. It should be noted that an isomorphism preserves a simulation.

We note $\mathcal{R}^m = \bigcup_{i=1}^{n}\{P_i^1, \ldots, P_i^m\}$, with $m \in \mathbb{N}$, the repository obtained by creating m copies of each P_i. We provide below a lemma on equivalence of two forms of products.

Lemma 1 *Let $R = \{P_1, ..., P_n\}$ be a repository of protocols then we have $\forall k \in \mathcal{N} : (\odot(\mathcal{R}^k)) = \cup_{i=1}^{k}(\odot(\mathcal{R}))^{\otimes_i}$.*

Proof 1 *(sketch) We have $\odot(\mathcal{R}^k) = \mathcal{R}_1 \cup ... \cup \mathcal{R}_k$, where $\mathcal{R}_{i,i\in[1,k]}$ is the union of FSMs which are the products that contain at most i instances of a protocol and that are formed of at least i instances. By definition, the FSM $\mathcal{R}_i$ corresponds to $\odot(\mathcal{R})^{\otimes_i}$. Hence, $\odot(\mathcal{R}^k) = \odot(\mathcal{R})^{\otimes_1} \cup ... \cup \odot(\mathcal{R})^{\otimes_k} = \cup_{i=1}^{k}(\odot(\mathcal{R}))^{\otimes_i}$.* ■

We end this section by the definition of the Ackermann function. The Ackermann function is a simple example of computable functions that are not primitive recursive.

Definition 4 ***(The Ackermann function)***

The Ackermann function is defined recursively for non-negative integers m and n as follows

- *If $n = 0$, then $A(n, m) = m + 1$.*
- *If $n > 0$ and $m = 0$, then $A(n, m) = A(n - 1, 1)$.*
- *If $n > 0$ and $m > 0$, then $A(n, m) = A(n - 1, A(n, m - 1))$.*

Using the Knuth notation, we have $A(n, m) = 2 \uparrow^{n-2} (m + 3) - 3$. In fact, $x \uparrow y$ equals $\underbrace{x^{(x^{(\ldots^x)})}}_{y}$ a very large integer totally different from $\underbrace{(((x)^x)^x)^{\ldots x}}_{y}$ that is $(x)^{(x^{y-1})}$.

4.2 Web services protocol model

The main goal of a web service protocol is to describe the ordering constraints that govern message exchanges between a service and its clients. The message exchange is strongly connected to operations execution, since basically an operation is a couple of (possibly empty) messages as described in the WSDL specification [WSD07]. In our work, we use the deterministic finite state machines formalism to represent protocols, following the model proposed in [BCG+03, BCT04a]. States represent the different phases that a service may go through during its interaction with a requester. Transitions are triggered by messages sent by the requester to the provider or vice versa. Each transition is labeled by a message name. A message corresponds to an operation invocation or its reply, i.e. a message is the input or the output of an operation. A complete sequence of messages exchange between two services is called *a conversation*. A web service may be implied in several conversations simultaneously and hence multiple *instances* of a same protocol can run concurrently. Formally, a protocol is a deterministic finite state machine (DFSM).

A protocol P of a web service S describes the set of the valid conversations supported by S. More precisely, a valid conversion of P corresponds to an execution path starting from the initial state of P and ending at a final state, i.e. a word from $L(P)$. In other words, the set of valid conversations is exactly the language of P. Usually the messages names are followed by message polarity [BCT06b] to denote whether the message is incoming (e.g. the plus sign) or outgoing (e.g. the minus sign). For simplicity reasons, and without loss of generality, we do not consider message polarities in this work. That is to say, incoming and outgoing messages are considered to be distinguished activities. Therefore, our protocol model corresponds to the *Roman model* [BCG+05c], i.e. an FSM where transitions are labeled by *abstract activities*.

4.3 The protocol-based composition synthesis problem

Let us now turn our attention to the problem of synthesizing composite services that are described by their business protocols. We illustrate this problem using the protocols from figures 4.3 and 4.4. We assume a repository of two available services S_1 and S_2 described respectively by their protocols P_1 and P_2 depicted at Figure 4.3(a) and (b). We consider the development of a new composite web service S_T whose target protocol P_T is depicted at Figure 4.4(a). The protocol P_T specifies the following behavior. First, a client starts by selecting a car. Then he has several possibilities to continue his interaction with the service S_T. He can either make a request for a credit or cancel all the operations performed or iterate many times on the loop 'ModifySelection-EstimatePayement'.

The interesting question is to see whether or not it is possible to implement the service S_T by combining the functionalities provided by (parts of) the available services S_1 and S_2. Dealing with this composition problem at the business protocol abstraction level, leads to the following question: *is it possible to generate (synthesize) the protocol P_T by combining (parts of) the available protocols P_1 and P_2?*

In our illustrative case the answer is *yes* and the composite service protocol P_T corresponding to the target protocol P_T using the protocols P_1 and P_2 is depicted at Figure 4.4(b). In this case, P_T is called the target protocol, P_C is called the delegator (composite protocol) while P_1 and P_2 are called the component protocols.

From formal standpoint, the services composition synthesis problem was defined in [BCG+03] as the problem of generating a *delegator* of a target service using available services. A delegator is an FSM whose activities are annotated with suitable *delegations* (protocols names) in order to specify which available service will run each activity of the target service. To make things

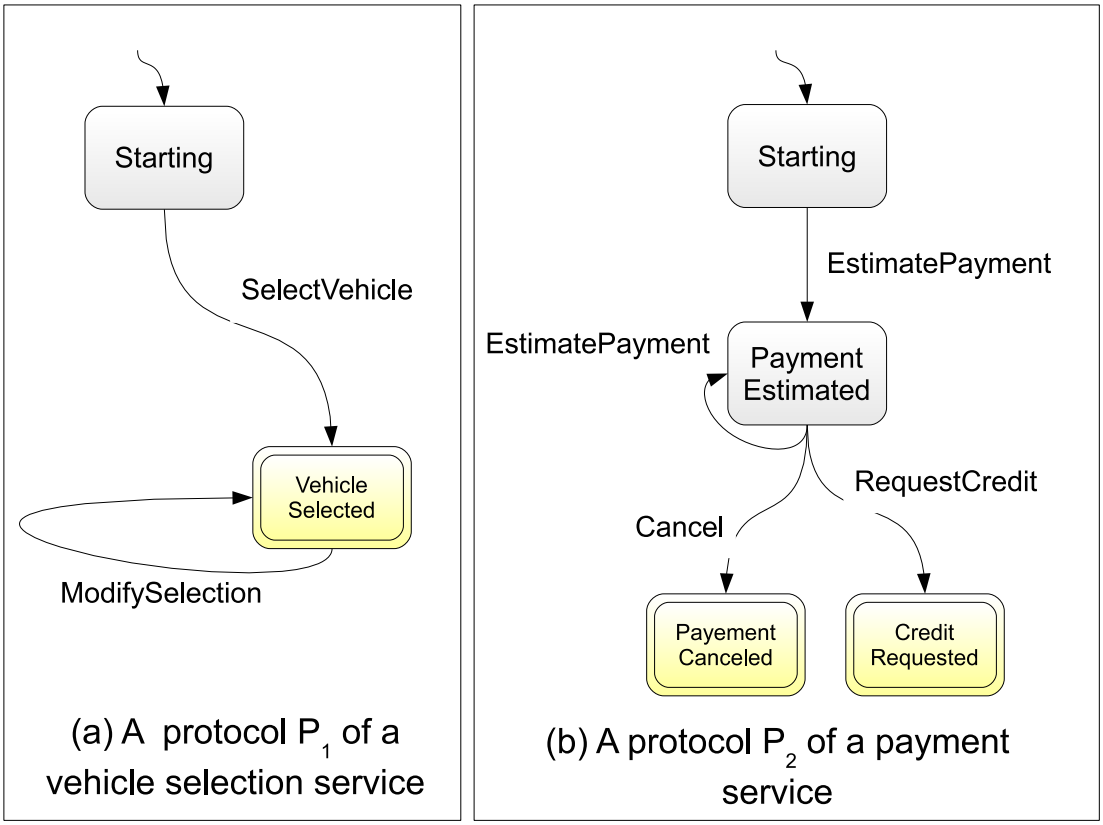

Figure 4.3: An example of web services protocols.

simpler, one can see a delegator as an FSM whose each transition label is prefixed by a component service name. The prefixes allow the execution engine to know which service will perform each invoked activity. In our example, the delegator is P_C, and the prefixes are either P_1 and P_2. For instance, this delagator specifies that the activity selectVehicle of the target protocol is delegated to the protocol P_1 while the activity estimatePayment is delegated to the protocol P_2. The notion of a delegator as well as its correctness are defined formally in[BCG+05c].

Indeed, in order to generate composite services, the combination of com-

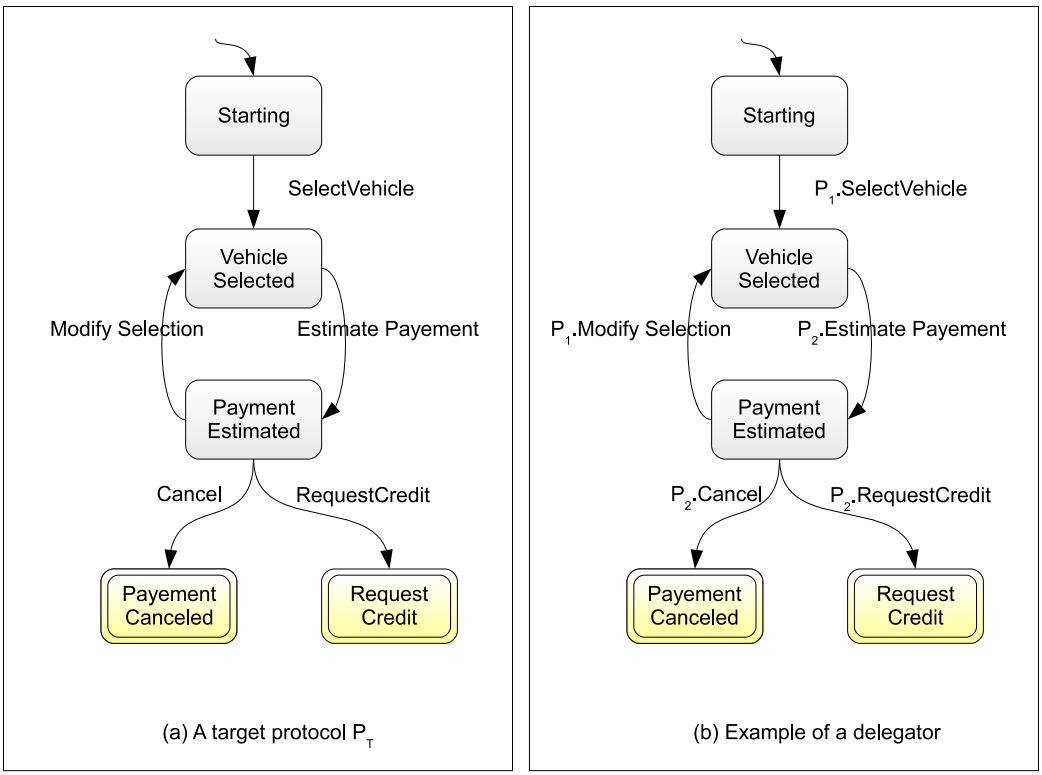

Figure 4.4: Example of protocol-based composition

ponent services must obey a set of constraints and can not be done anyway. Firstly, when a composition uses a component service S, all the engaged conversations of S must be valid. For instance, when using the vehicle selection service, a delegator can not invoke the *ModifySelection* operation before invoking the *SelectVehicle* operation. Also, it invokes the *SelectVehicle* once and then it can invoke the *ModifySelection* a (infinite) many times. Secondly, final states of the delegator must correspond to final states of all used component service. That is, one should not leave a running instance of a component service unterminated . For instance, the *PaymentCanceled* final state in the target protocol can not correspond to a set of states including the not final *PaymentEstimated* state. Thirdly, a delegator must be correct, i.e. if and only

if the delegator whose the projection without prefixes simulates (after prefixes deletion) the target service protocol.

In this thesis, we will not provide here a formal definition of the delegation, since we use instead an approach based on the simulation preorder. But it should be noted that for many composition synthesis cases, one instance per service is not sufficient to compute composite services (i.e. the bounded instances assumption [BCG+03]), and the composition may involve a number of instances that is not bounded a priori. Hence, we introduce below a definition of a generic composition synthesis problem that makes explicit the number of instances of protocols allowed in a composition.

4.3.1 Generic Composition Synthesis Problem (GCSP)

Let $\mathcal{R} = \{P_i,\ i \in [1,n]\}$ be a repository of services protocols, where each $P_{i,i\in[1,n]} =< \Sigma, S_i, F_i, s_i^0, \delta_i >$ is a protocol. Indeed, we consider that we are able to use several copies (duplicates) of each protocol. Hence, for each $P_i \in \mathcal{R}$, we denote by P_i^j the j^{th} copy of the protocol P_i. Given a protocol repository $\mathcal{R}$, we note by $\mathcal{R}^m = \bigcup_{i=1}^{n}\{P_i^1, \ldots, P_i^m\}$, with $m \in \mathbb{N}$, the repository obtained by creating m copies of each P_i.

Definition 5 ***(generic protocol composition problem)***

Let $\mathcal{R}$ be a set of available service protocols and S_T be a target protocol and let $k \in \mathbb{N}$. A generic protocol synthesis problem, noted $Compose(\mathcal{R}, S_T, k)$ is the problem of deciding whether there exists a composition of S_T using $\mathcal{R}^k$.

Note that, instances of this generic composition problem are characterized by the maximal number of instances of component protocols that are allowed to be used in a given composition. We distinguish in the following between two main cases, namely the bounded instances composition and the unbounded instances one.

4.3.2 Protocol synthesis problem: the bounded case.

Existing work [BCG+05c, BCG+05b, MW07] that investigated the protocol synthesis problem made the simplifying assumption that k, the number of instances of a service that can be involved in composition of a target service is bounded by a constant k fixed *a priori*, i.e., they address the problem $Compose(\mathcal{R}, P_T, k)$.

The solution proposed in [BCG+03] consists in reducing the problem $Compose(\mathcal{R}, S_T, 1)$ into the satisfiability of a suitable formula of Deterministic Propositional Dynamic Logic (DPDL) [FL79]. In [BCG+05b], this DPDL-based framework proposed was extended to deal with a more expressive protocol model [BCG+05c].

Interestingly, in [MW07] the protocol synthesis problem is reduced to the problem of testing a simulation relation between the target protocol and the product of component protocols. Using such a reduction, [MW07] shows the Exptime completeness of the bounded instances protocol synthesis problem[5]. More precisely, assuming a repository $\mathcal{R}$ and a target protocol P_T then there exists a composition of P_T using $\mathcal{R}$ if and only $P_T \preceq \odot(\mathcal{R})$. Example 5 illustrates the use of the simulation relation to compute a composition.

Example 5 *On figure 4.5 we aim at composing P_T (figure 4.5-(a)) using P_1 and P_2 (figure 4.5-(b)). In order to achieve this task, we compute $P_1 \odot P_2$ (figure 4.5-(c)) and then we check the simulation ($P_T \preceq P_1 \odot P_2$). The dashed arrows illustrate the computation of this simulation by connecting each pair of states which will appear on it.*

Note that this k-bounded instance protocol synthesis problem can be reduced w.l.o.g to the simplest case where $k = 1$. Indeed, if $k > 1$ the problem $Compose(\mathcal{R}, S_T, k)$ can be straightforwardly reduced to the problem $Compose(\mathcal{R}^k, S_T, 1)$. This reduction is trivial because a composition with k in-

[5]The Exptime upper bound is known from [BCG+05c].

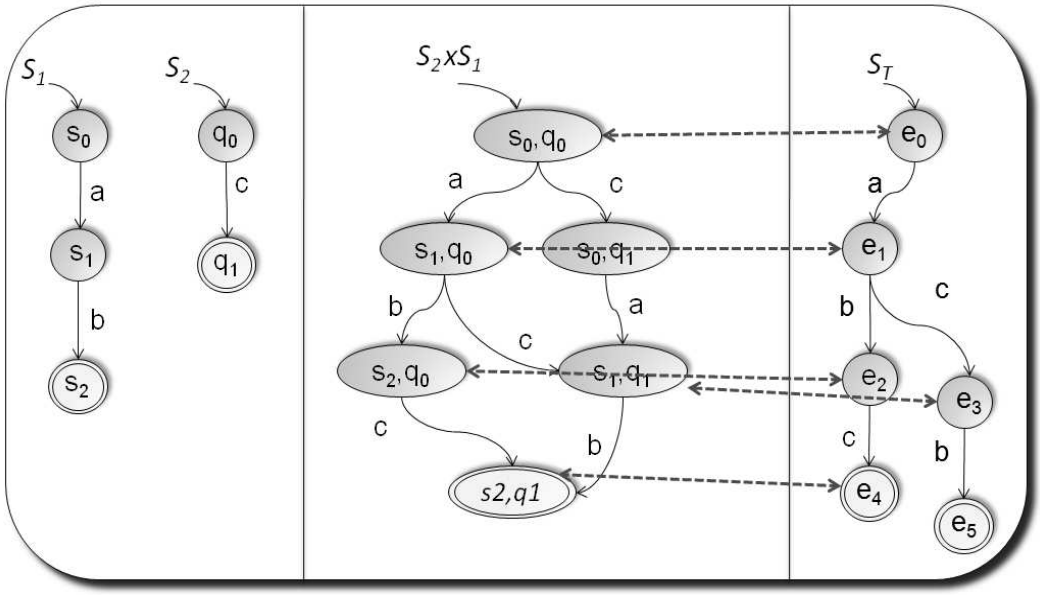

Figure 4.5: The use of simulation and product to compositions.

stances is performed by duplicating each component service k times [BCG$^+$03] and allowing each copy to run only one instance. The following proposition summarizes the formalization of the bounded protocol synthesis problem using the k-iterated product operator and using the simulation preorder as proposed in [MW07].

Proposition 1 *Let $Compose(\mathcal{R}, S_T, k)$ be a protocol synthesis problem with k a finite positive integer. The problem $Compose(\mathcal{R}, S_T, k)$ has a solution iff $S_T \preceq \odot(\mathcal{R}^k)$ or equivalently $S_T \preceq \cup_{i=1}^{k}(\odot(\mathcal{R}))^{\otimes_i}$.*

Proof 2 *We know that the problem $Compose(\mathcal{R}, S_T, k)$ has a solution iff $S_T \preceq \odot(\mathcal{R}^k))$ since since (i) $(\mathcal{R}^k)$ is the set containing k copies of each component protocols and (ii) the problem of composing with k instances is defined as being the one of composing with k copies of each service whose the number of allowed instances is equal to 1.*

Compose$(\mathcal{R}, S_T, k)$ *has a solution iff* $S_T \preceq \cup_{i=1}^{k}(\odot(\mathcal{R}))^{\otimes_i}$ *holds since* $\odot(\mathcal{R}^k) = \cup_{i=1}^{k}(\odot(\mathcal{R}))^{\otimes_i}$. ■

Example 6 illustrates a composition synthesis that requires many instances of a same service and how one instance may be insufficient to compute compositions, but this becomes possible when using two instances.

Example 6 *On figure 4.6-(a) we consider a target service* P_{T1} *and a component service* P_1 *for which only one instance is allowed. It is straightforward to see that no delegator can be generated since* $P_{T1} \not\preceq P_1$. *Hence, there is no possible composition for* P_T. *However, this becomes possible when considering two instances or copies(*P_1^1 *and* P_1^2*) of* P_1 *(figure 4.6-(b)) since* $P_{T1} \preceq P_1^1 \odot P_1^2$ *. This delegator, that involves the two instances of* P_1*, is depicted on figure 4.6-(c).*

The settings of the bounded instances composition is very restrictive in the sense that some simple protocol synthesis problems, in which the solution may use an unbounded number of instances of component protocols, cannot be solved. We illustrate this limitation on example 7.

Example 7 *Figure 4.7 illustrates examples of the composition where the k-bounded settings prevent the generation of rather simple delegators. On figure 4.7-(a) we provide two component protocols* P_1 *and* P_2. *As a first target protocol, we consider* P_{T1} *(figure 4.7-(b)). It is easy to see that* *Compose*$(P_T, \{P_1, P_2\}, k)$ *has no solution for any finite* $k \in \mathbb{N}$. *Now, one can add epsilon transitions from each final state to the the initial one in both component services in order to enable the composition and hence he creates two new protocols* P_1' *and* P_2'. *Indeed, the intuition is to allow an instance of each of* P_1' *and* P_2' *to generate respectively an infinite number of sequential instances of* P_1 *and* P_2 *and we have* $P_T \preceq P_1' \odot P_2'$ *(i.e. a composition exists!). However, note that adding these epsilon transitions allow to only generate an*

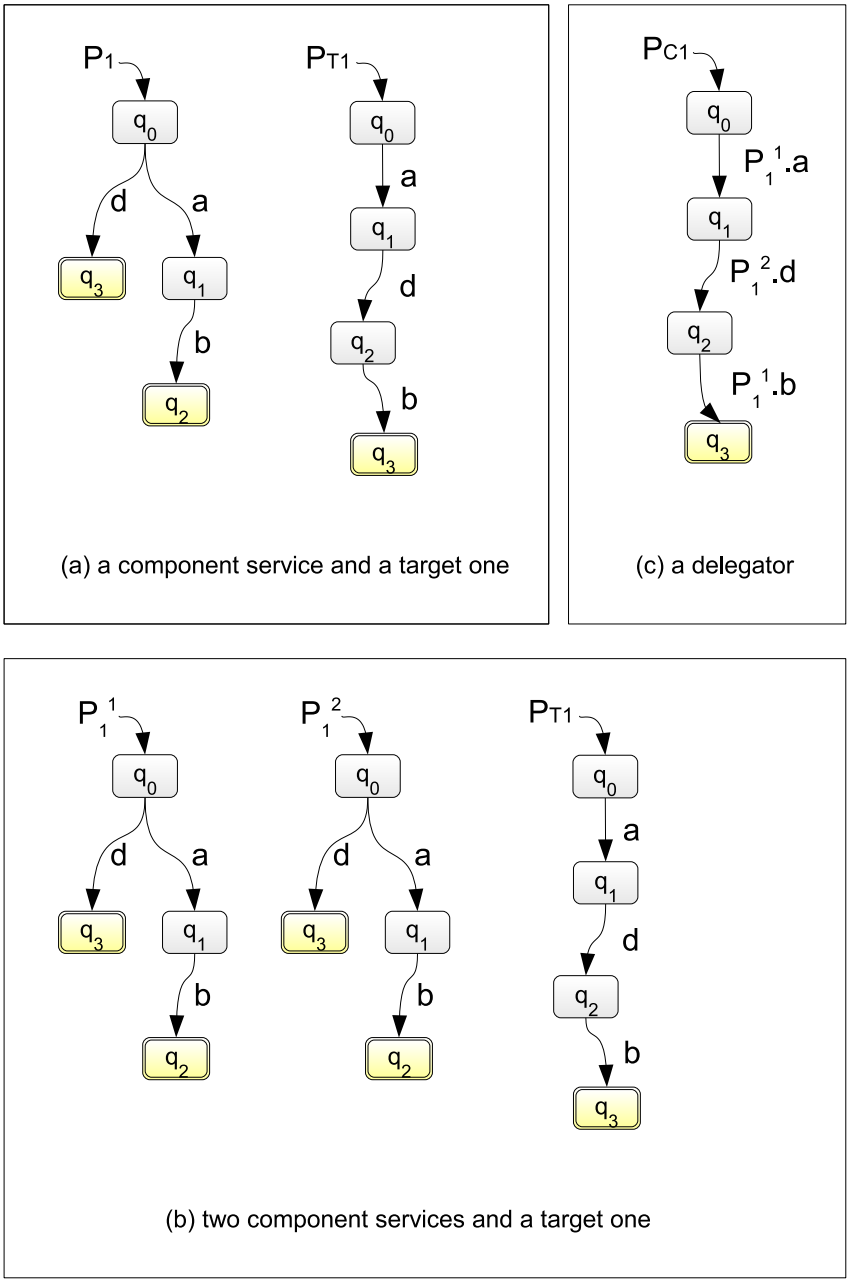

Figure 4.6: Example of the use of many instances for the composition

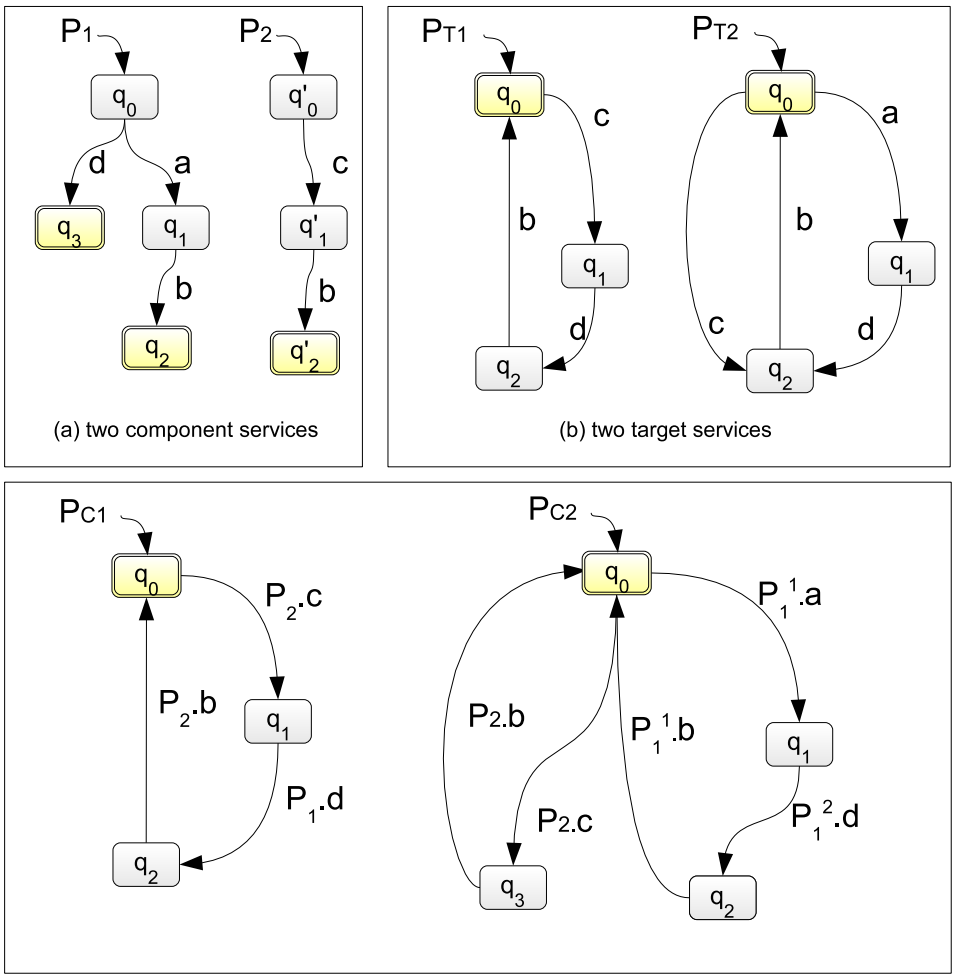

Figure 4.7: Instances Bounded Limitations

infinite number of sequential instances and not parallel one. To explicit this fact, we propose to compute a delegator for the second target service P_{T2} (figure 4.7-(b)). The interesting ascertainment on the latter's behavior is that an execution of P_{T2} can involve two parallel instances of P_1, that is P_1^1 and P_1^2. Indeed, the previous solution (adding ϵ-transitions) does not hold any more and the real hardness during a composition synthesis comes from the need of (a priori) unknown number of parallel instances.

These strong limitations motivated our work on the unbounded instance case of the protocol synthesis problem.

4.3.3 Protocol synthesis problem: the unbounded case.

In the remainder of this work we study the protocol synthesis problem in the case where the number of protocol instances that can be used in a composition are not bounded *a priori* (i.e., the problem $Compose(\mathcal{R}, S_T, +\infty)$).

In other words, given a repository $\mathcal{R} = \{P_1, \ldots, P_n\}$ of service protocols, we consider the generation of new composite protocols that can be obtained by an asynchronous product of any subset of protocols in $\mathcal{R}^{+\infty}$.

More precisely, we consider in this paper the decision problem underlying the general protocol synthesis problem, i.e., the problem $Compose(\mathcal{R}, S_T, +\infty)$.

Problem 1 ***Unbounded composition synthesis problem (UCSP)***

Let $\mathcal{R}$ and S_T defined as previously. Is the problem $Compose(\mathcal{R}, S_T, +\infty)$ decidable?

One way to answer this open question is to consider the related 'simulation relation' decision problem. Indeed, $Compose(\mathcal{R}, S_T, +\infty)$ has a solution if S_T is simulated by a product of any subset elements of $\mathcal{R}^{+\infty}$ (i.e., $S_T \preceq \odot(\mathcal{R}^{+\infty})$). Such a characterization of solutions can also be expressed using the product closure operator as stated below.

Theorem 1 *The problem $Compose(\mathcal{R}, S_T, +\infty)$ has a solution iff $S_T \preceq \odot(\mathcal{R}^{+\infty})$ (or equivalently, $S_T \preceq (\odot(\mathcal{R}))^{\otimes}$).*

Proof 3 *Since we are assuming an infinite number of instances, one can replace the k from proposition 1 by $(+\infty)$. Hence, we obtain $Compose(\mathcal{R}, S_T, k = +\infty)$ has a solution iff $S_T \preceq \cup_{i=1}^{k} (\odot(\mathcal{R}))^{\otimes_i}$ and thus*

Compose$(\mathcal{R}, S_T, +\infty)$ *has a solution iff* $S_T \preceq \odot(\mathcal{R}^{+\infty})$. *From lemma 1, we have* $\odot(\mathcal{R}))^{\otimes} = \odot(\mathcal{R}^{+\infty})$ *thus Compose*$(\mathcal{R}, S_T, +\infty)$ *has a solution iff* $S_T \preceq \odot(\mathcal{R}))^{\otimes}$.■

The main difficulty for the unbounded case comes from the fact that a product closure of an FSM is not an FSM. Since $\odot(\mathcal{R})$ is an FSM, we shall prove in the sequel that checking simulation between an FSM M and a product closure of an FSM (i.e., $M^{\otimes}$) is decidable. We formalize this problem as follows.

Problem 2 ***Simulation Decidability Problem (SDP)***

Let A and M be two FSMs. Is it decidable whether $A \preceq M^{\otimes}$?

To investigate the previous problem , we need first to define a suitable state machine model that enables to describe a product closure of an FSM.

Discussion

In this chapter we introduced the problem of the generic composition synthesis (GCSP). An important contribution was to show that the GCSP can be formalized as the problem of testing the simulation of a DFSM by the product closure of an FSM. Since each service may be instantiated one or many times (possibly infinite times), we provided a definition of the synthesis problem making the number of involved instances explicit. The next chapter is devoted to the investigation of the Simulation Decidability Problem (SDP) and the tractability of the GCSP.

Five

Decidability and complexity

In the previous chapter, we showed that the unbounded composition synthesis problem (UCSP) can be reduced to a simulation test between a DFSM and a product closure of an FSM, i.e. the SDP. To cope with this problem, we first propose a infinite state machine, called Product Closure of State Machines (PCSM), that enables to describe product closures of FSMs. Then we investigate the problem of testing a simulation of an DFSM and a PCSM, or more generally testing a simulation of an FSM and a PCSM. The latter being an infinite state machine, we develop a technique to prove that while testing the existence of such a simulation relation, it is sufficient to explore only a finite part of the PCSM.

This chapter is organized as follows. We start by the definition of a PCSM. We then provide a composition algorithm and we show it to terminate and to be correct. We end this chapter by a complexity analysis of our problem and two of its restrictives forms.

5.1 Product Closure State Machine (PCSM)

In this section, we introduce the Product Closure State Machine (PCSM) as a new state machine that will be used to describe the product closure of an FSM. The PCSM describes the execution of a possibly infinite number of asynchronous parallel instances of an FSM. We first present a PCSM informally through examples and then we provide its formal definition.

Given an FSM M, its associated PCSM $M^{\otimes}$ can be seen as the FSM M with unbounded stacks of tokens on each state. Therefore, unlike finite state machines where the *instantaneous description (ID)* of a given state machine is given by its current state, an ID of a PCSM involves the set of states of its underlaying FSM as well as the number of tokens (i.e. number of instances) at each state. We illustrate the notion of an ID of a PCSM on the following example.

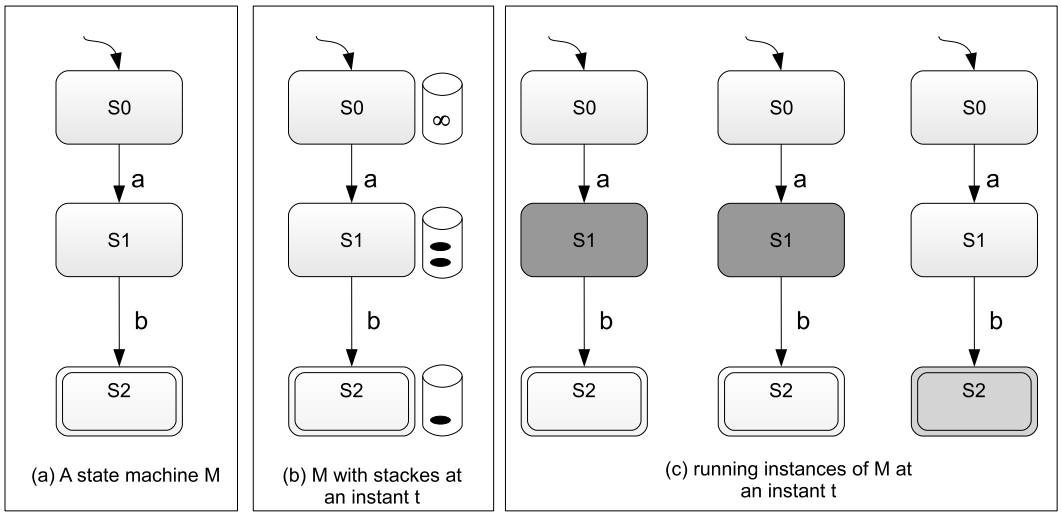

Figure 5.1: An FSM with associated stacks.

Example 8 *Figure 5.1-(a) illustrates an FSM M that allows to perform $'a'$ and then goes to a final state along a transition labeled by $'b'$. Assume that, we are able to run several instances of M. Consider now the global state of*

the PCSM $M^{\otimes}$ at an instant t after the execution of a sequence of activities aaba. Such a global state is represented by Figure 5.1-(b) which associates with each state of M a stack of tokens that keep track of the number of running instances of M. Figure 5.1-(b) shows that at the instante t, two instances of M are at state s_1 while an other token is at state s_2. Figure 5.1-(c) depicts all the running instances of M by shadowing the current state that each instance has reached. Note that, having a token on a final state is not useful since it can not move any more.

Informally, PCSM can execute a transition labeled by $'a'$ in two ways :

1. **creation of a new instance of M:** if there is an outgoing transition labeled by a from the initial state of M to a state q. Upon such a transition, a token is added to q, or

2. **moving an existing instance of M:** if there exists two states q and q' such that $(q, a, q') \in \delta_M$ and q has one or more tokens, then upon this transition, a token is moved from q to q'.

We illustrate the execution of a sequence of achtivities by the following example.

Example 9 *Figure 5.2 illustrates the execution of the sequence $'abbc'$ by making use of a new PCSM $M^{\otimes}$ depicted at figure 5.2-(a). At the beginning (the instant t), the initial ID of $M^{\otimes}$ is described by empty stakes on all states of M except the initial one that has an infinite number of tokens (figure 5.2-(a)). In order to execute the transition labeled by $'a'$, we move a token from s_0 to s_1. The new ID of $M^{\otimes}$ is provided on figure 5.2-(b) where we have one token at state s_1. This corresponds to the creation of a new instance of M. Then, we execute the transition labeled by $'b'$ by moving a token from s_0 to s_3. The new ID of $M^{\otimes}$ is depicted at figure 5.2-(c) where we have one token at each of s_1 and s_3. Note that, at this instant (t+2) we have two concurrent running*

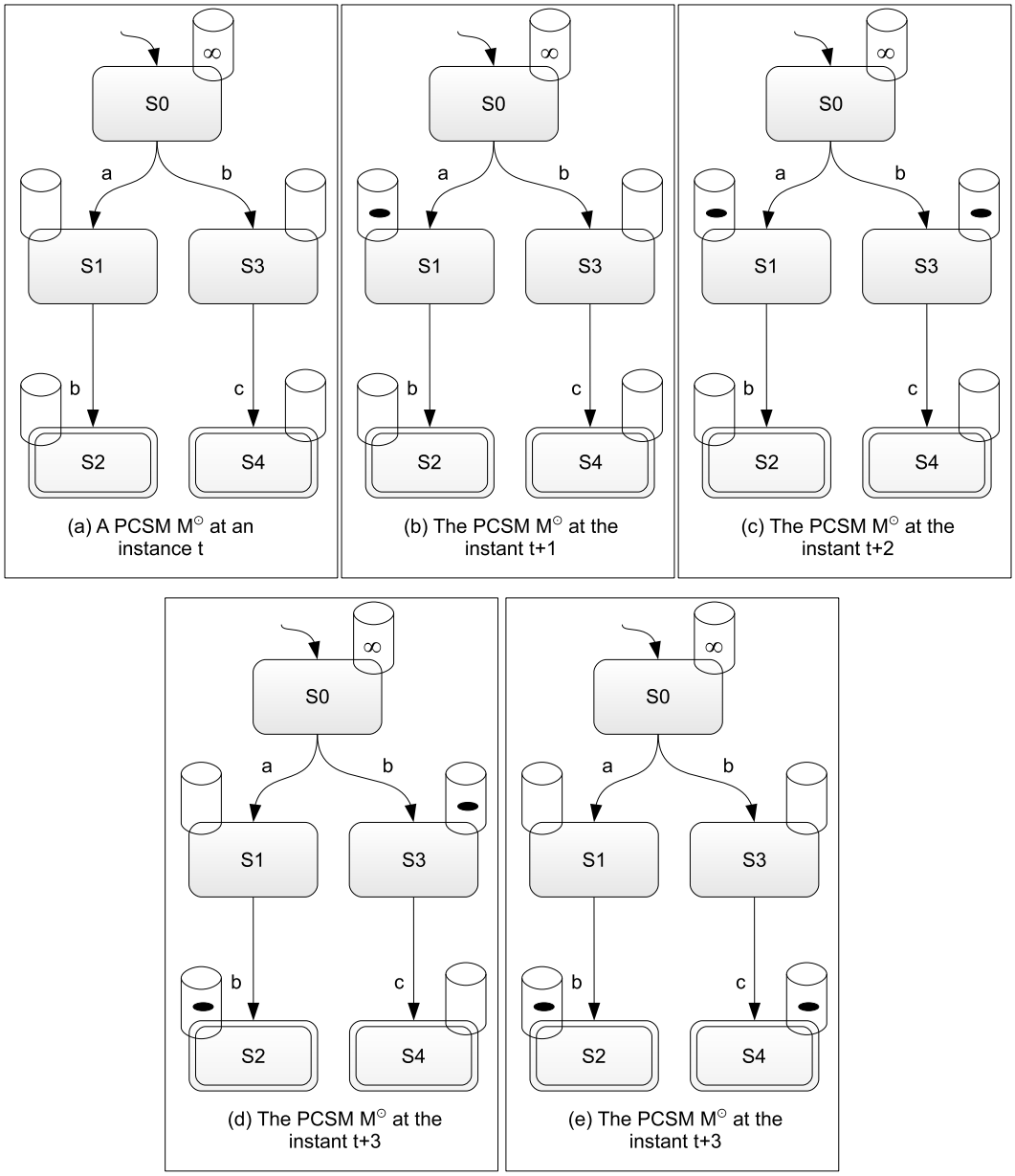

Figure 5.2: An example of execution of a sequence using a PCSM.

instances of the FSM M. To execute the second $'b'$ of the sequence, we move the token from s_1 to s_3 (figure 5.2-(d)). This corresponds to moving an existing instance of M. Finally, we execute the transition labeled by $'c'$ by moving the token from s_2 to s_4 (figure 5.2-(e)). At the instant (t+4) depicted on the figure 5.2-(e), the PCSM $M^{\otimes}$ is in a final ID since all the running instances of M are in final states.

Note that, as it can be seen in the previous example, a PCSM $M^{\otimes}$ may be indeterministic even if its corresponding FSM M is deterministic. For instance, at instant $(t+1)$ and in order to execute the transition labeled by b, $M^{\otimes}$ can either by creating a new instance of M (which leads to Figure 5.1-(c)) or by moving a token from s_1 to s_3.

It is necessary to devote a specific treatment to FSMs that have initial states with incoming transitions. In such a case, we need to be able to distinguish between : *(i)* an infinite number of instances that are at the initial state and which have not yet started their execution and *(ii)* the other running instances that have reached the initial state through one of its incoming transitions. This problem is illustrated by figure 5.3 where we can see that, after the execution of the activity a (figure 5.3-(b)), we have to distinguish the instance that was used. To cope with this problem, we propose a pre-processing of an FSM M in order to transform it into a simulation-equivalent FSM $\widetilde{M}$ such that $\widetilde{M}$ does not contain incoming transitions. The main idea of such a pre-preocessing is to duplicate the initial state q_0 into a new state $\widetilde{q_0}$ that has no incoming transitions. We provide below a formal definition of this transformation.

Definition 6 ***Transformation of FSMs***

Let $M =< \Sigma_M, S_M, F_M, q_M^0, \delta_M >$ FSM, we build a new FSM $\widetilde{M} =< \Sigma_M, \widetilde{S_M}, \widetilde{F_M}, \widetilde{q_M^0}, \widetilde{\delta_M} >$ such that :

- $\widetilde{S_M} = S_M \cup \widetilde{q_M^0}$.

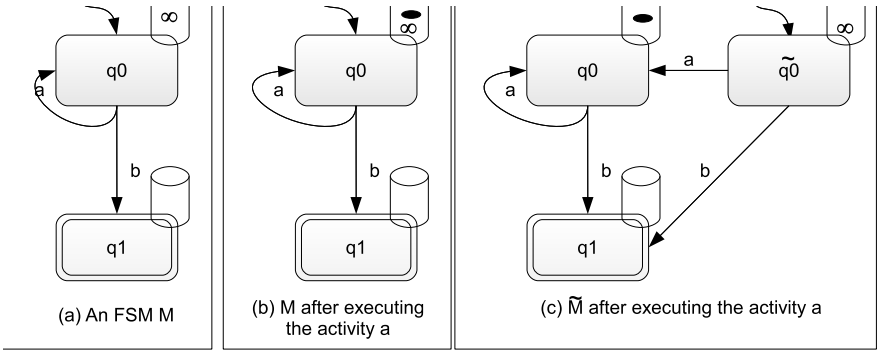

Figure 5.3: Transformation of FSMs

- $\widetilde{\delta_M} = \delta_M \cup \{(\widetilde{q_M^0}, a, q) for\ each\ (q_0, a, q) \in \delta_M\}$.
- $\widetilde{q_M^0} \in \widetilde{S_M}$ *is the initial state of* $\widetilde{M}$.
- $\widetilde{F_M} = F_M$.

We call $\widetilde{M}$ *the associated FSM of* M.

In order to model the infinite number of tokens, it is enough to put them on the new initial state $\widetilde{q_M^0}$ as illustrated by figure 5.3-(c) and we are now able to distinguish the running instance that has reached q_M^0. The following lemma states that an FSM and its associated one are simulation-equivalent.

Lemma 2 *Let* $M =< \Sigma_M, S_M, F_M, q_M^0, \delta_M >$ *FSM and* $\widetilde{M} =< \Sigma_M, \widetilde{S_M}, \widetilde{F_M}, \widetilde{q_M^0}, \widetilde{\delta_M} >$ *be its associated FSM. Then* $M =_{sm} \widetilde{M}$ *(i.e.* $M \preceq \widetilde{M}$ *and* $\widetilde{M} \preceq M$*).*

Proof 4 *(sketch)*

1. *($M \preceq \widetilde{M}$). We have $(\widetilde{q_M^0})$ is a copy of q_M^0 with the same outgoing transitions but not the incoming ones. Each incoming transition in q_M^0 is translated into a transition from $\widetilde{q_M^0}$ to the state associated with q_M^0 of $\widetilde{M}$. Since by construction q_M^0 of $\widetilde{M}$ simulates q_M^0 of M, then q_M^0 of M is simulated by $\widetilde{q_M^0}$.*

2. *($\widetilde{M} \preceq M$). The new state $\widetilde{q_M^0}$ which is trivially simulated by q_M^0 since the former has exactly the same set of outgoing transitions (i.e. same labels and same target states). Hence, the simulation holds.*

In the rest of this thesis, and w.l.o.g, we consider FSMs that do not have initial states with incoming transitions. We see the transformation as a pre-processing step that can be executed when we need to put in an adequate form. We now turn our attention to a formal definition of a PCSM. Thus, we start by defining a configuration, a formal characterization of an ID of a PCSM.

5.1.1 Configuration

Let $M =< \Sigma_M, S_M, F_M, q_M^0, \delta_M >$ be an FSM and let $|I_s(M)| = l$ and $|H_s(M)| = n$ be respectively the set of intermediate and hybrid states of M. We assume states of $I_s(M)$ (respectively, $H_s(M)$) ordered according to the lexicographical order and relabeled accordingly with integers from 1 to l (respectively, from $l+1$ to $l+n$). The configurations of $M^{\otimes}$ are formally defined below.

Definition 7 ***(Configuration)*** *A configuration C of a product closure $M^{\otimes}$ is a tuple of size $l+n$ of positive integers. The i^{th} element of C, written $C[i]$, denotes the number of tokens (i.e., instance of M) that are at state i. We*

say that $C[i]$ is the witness of the state i in a configuration C. Note that, if $i \leq l$ (respectively, $i > l$) then $C[i]$ is a witness of an intermediate state (respectively, an hybrid state).

A configuration C is an initial (respectively, final) configuration of $M^{\otimes}$ iff $C[i] = 0$, $\forall i \in [1, l+n]$ (respectively, iff $C[i] = 0$, $\forall i \in [1, l]$.

Note that, a configuration informs about the number of tokens (instances of M) that are at intermediate or hybrid states. Indeed, it is useless to keep track of tokens that are at final, not hybrid, states since such instances are terminated and can no longer be used in the future. In the rest of this thesis, we consider PCSMs built on FSMs M that have at least an hybrid or an intermediate state. As shown in section 4.3.2, the real hardness when composing is related to parallel instances. Whenever M has neither hybrid nor intermediate states, then all sequences of transitions in M have lengths equal to 1. Hence, M can not have parallel instances and a simple way to solve the GCSP is to create a new FSM M' by adding ϵ-transitions to M from each final state to the initial one, and then to test a simulation against M' instead of $M^{\otimes}$.

Example 10 *In the example of Figure 5.4, the FSM M contains only one intermediate state (state q_1) and one hybrid state (state q_2). Hence, a configuration associated with $M^{\otimes}$ is a pair of integers where the first (respectively, the second) integer is the witness of the state q_1 (respectively, q_2). For instance, the configuration $C_2 = (2, 0)$ indicates an instantaneous description of $M^{\otimes}$ in which there are two instances of M at q_1 and zero instances of M at q_2. While the configuration $C_3 = (0, 1)$ that there are zero instances of M at q_1 and one instance at q_2.*
We define below a new preorder on configurations.

Definition 8 ***Configuration cover*** *Let C and C' be two configurations of $M^{\otimes}$. C covers C', denoted by $C' \lhd C$, iff $\forall i \in [1, l] : C[i] = C'[i]$ and $\forall i \in [l+1, l+n] : C[i] \leq C'[i]$.*

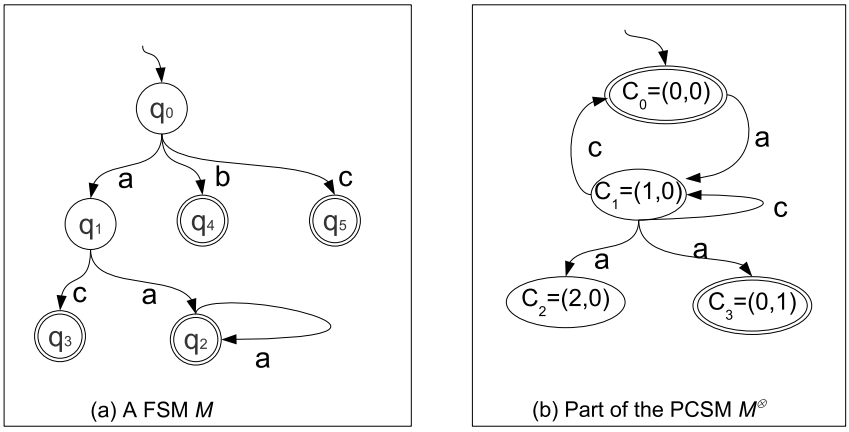

Figure 5.4: An FSM M and a part of the PCSM $M^{\otimes}$.

5.1.2 Definition of the PCSM

Using the notion of configuration, we formally define below PCSMs.

Definition 9 ***(PCSM)*** *Let* $M =< \Sigma_M, S_M, F_M, q_M^0, \delta_M >$ *be a FSM with* $|I_s(M)| = l$ *and* $|H_s(M)| = n$. *The associated PCSM of* M *noted* $M^{\otimes} =< \Sigma_M, \mathcal{C}, F_C, C_0, \phi >$, *with:*

- $\mathcal{C}$ *is an (infinite) set of states consisting of all the configurations of* $M^{\otimes}$,
- F_C *is the set of final configurations of* $M^{\otimes}$, *i.e.,* $\{C \in \mathcal{C} \mid C[i] = 0, \forall i \in [1, l]\}$,
- C_0 *is the initial state of* $M^{\otimes}$ *and corresponds to the initial configuration, i.e.,* $C_0[i] = 0$, $\forall i \in [1, l + n]$,
- $\phi \subseteq \mathcal{C} \times \Sigma_M \times \mathcal{C}$ *is an infinite set of transitions. The set* ϕ *is built as follows. Let* C_1 *and* C_2 *be two configurations in* $\mathcal{C}$. *We have* $(C_1, a, C_2) \in \phi$ *iff* $(q, a, q') \in \delta_M$ *and one of the following conditions holds:*
  1. $q = q_M^0$ *and* $q' \in (F_M \backslash H_s(M))$ *with* $C_1[i] = C_2[i]$, $\forall i \in [1, l + n]$, *or*
  2. $q = q_M^0$ *and* $q' \in (I_s(M) \cup H_s(M))$ *with* $C_2[q'] = C_1[q'] + 1$, $C_1[i] = C_2[i]$, $\forall i \in [1, l + n]$ *and* $i \neq q'$, *or*

3. $\{q, q'\} \subseteq (I_s(M) \cup H_s(M))$ *with* $C_1[q] > 0$, $C_2[q] = C_1[q] - 1$, $C_2[q'] = C_1[q'] + 1$, $C_1[i] = C_2[i]$, $\forall i \in [1, l+n]$ *and* $i \notin \{q, q'\}$, *or*
4. $q \in (I_s(M) \cup H_s(M))$ *and* $q' \in (F_M \backslash H_s(M))$ *with* $C_2[q] = C_1[q] - 1$, $C_1[i] = C_2[i]$, $\forall i \in [1, l+n]$ *and* $i \neq q$.

We illustrate a part of a PCSM by the following example.

Example 11 *Figure 5.4(b) describes a part of $M^{\otimes}$, the PCSM of the FSM M depicted at Figure 5.4(a). As mentioned before, configurations of $M^{\otimes}$ are pairs (i, j) where i (respectively, j) is the witness of the state q_1 (respectively, q_2). The infinite state machine $M^{\otimes}$ is initially in the configuration $C_0 = (0, 0)$ then it can, for example, execute the activity a, upon which it moves to the configuration $C_1 = (1, 0)$. At this stage, $M^{\otimes}$ has two possibilities to execute the activity c : (i) by moving the current instance of M that is at state q_1 into the final state q_3, or (ii) by creating a new instance of M and moving it from state q_0 into the final state q_5. Note that, as the final states q_3 and q_5 are not described in configurations, case (i) make the $M^{\otimes}$ moving back to the configuration C_0 while case (ii) makes it looping on configuration C_1.*

5.2 Simulation Decidability Problem

Let us first recall the Simulation Testing Problem (SDP).

Problem 3 ***(SDP)***

Let A and M be two FSMs. Is it decidable whether $A \preceq M^{\otimes}$?

This section answers positively to this problem by providing an algorithm that checks the existence of a simulation relation between an FSM and a PCSM. We prove that the proposed algorithm terminates and is correct (i.e. sound and complete). The main difficulty to devise our algorithm comes from the fact that we have to check the existence of a simulation relation between

an FSM and a PCSM, this latter being an infinite state machine. The corner stone of our proof is to show that to check the existence of such a simulation relation we need only to explore a finite part of the corresponding PCSM.

5.2.1 Composition synthesis algorithm

Our algorithm is made of three boolean procedures : Check-Sim, Check-Candidate and Check-Cover.

- The inputs of Check-Sim are an FSM A, a state q from A, a PCSM $M^{\otimes}$ and a configuration C from $M^{\otimes}$. Check-Sim allows to check whether the state q from A is simulated by the configuration C from $M^{\otimes}$.

- The inputs of Check-Candidate, are an FSM A, a state q' from A, a PCSM $M^{\otimes}$, a configuration C from $M^{\otimes}$ and a letter a from Σ_M. Check-Candidate allows to compute the successor configuration of C from $M^{\otimes}$ by transiting with the letter a. These configurations represent the candidates to simulate the state q from A.

- The inputs of Check-Cover are an FSM A, a state q from A, a PCSM $M^{\otimes}$ and a configuration C from $M^{\otimes}$. Check-Cover allows to check whether the state the configuration C from $M^{\otimes}$ covers one of the configurations which was tested to simulate q previously.

These procedures are detailed respectively on algorithms 1, 2, 3 and run as follows. When checking the simulation between a given state q and a configuration C, the Check-Sim procedure will recursively generate new simulation tests by making calls to the Check-Candidate procedure for each transition (q, a, q') in A. This latter procedure enables to check if the state q' is simulated by at least one configuration C' such that (C, a, C') is in $M^{\otimes}$. The execution of the algorithm generates a tree where the nodes are labeled with pairs (q, C) that correspond to the calls of the Check-Sim algorithm and the

edges are labeled by the letters from the alphabet that are used when calling Check-Candidate.

Example 12 *Figure 5.5-(c) shows an execution of a Check-Sim between the initial state q_1 of the FSM of figure 5.5-(a) and the initial configuration $C_0 = (0,0)$ of the product closure of the FSM M of figure 5.5-(b). Recall that , configurations of $M^{\otimes}$ are pairs (i,j) where i (respectively, j) is the witness of the state q_1 (respectively, q_2).*

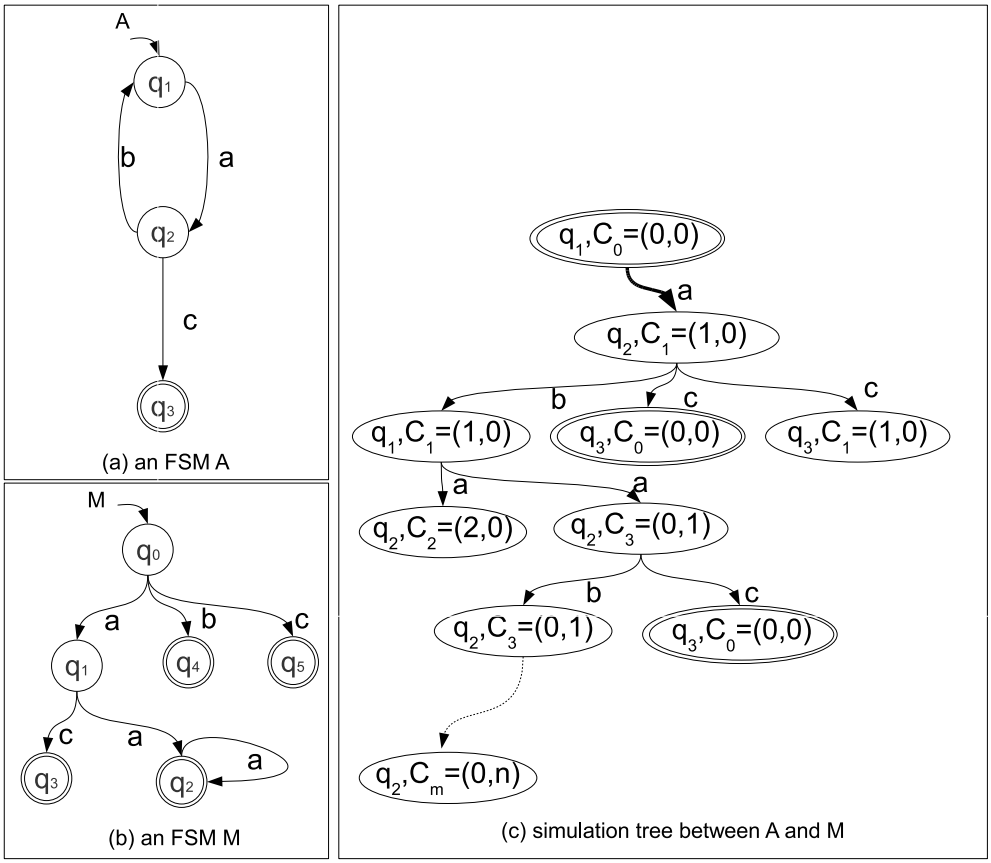

Figure 5.5: Example of a simulation tree.

A crucial question is then to ensure that the algorithm terminates. Observe that for each state q', the number of candidates C' generated by the Check-

Candidate procedure is linear in the size of M since for any configuration C of a PCSM $M^{\otimes}$, the number of outgoing transitions is finite and bounded by the total number of transitions in M. Therefore, to ensure termination of the algorithm it remains to show that there are no infinite branches in the execution tree of the algorithm. Hereafter, we distinguish two cases :

- In the simple case where A is an FSM without loops, it is easy to see that the corresponding execution tree of the algorithm is finite since the length of the branches are bounded by the size of the maximal path in A.
- For the general case, a state q belonging to a loop in A may appear an infinitely many times in a branch of the execution tree of the algorithm. Such a case is illustrated on the Figure 5.6-(b) where the branch depicted in bold font involves many times the state q_1 which belongs to the loop $(ab)^*$ of the FSM A.

An important technical contribution of this work is to provide necessary and sufficient conditions that enable to cut such infinite branches. This is achieved by the second terminating condition of the Check-Sim (i.e., the call to the Check-Cover procedure) which is based on the following property.

Property 1 *Let A and M be two FSMs and we consider the simulation of A by $M^{\otimes}$. If a state q of A appears infinitely many times in a given branch of the simulation tree then there is necessarily a sub-path in this branch from a node (q, C) to a node (q, C') such that C' is a cover of C. Interestingly, this condition characterizes the cases where a loop in A is simulated by $M^{\otimes}$.*

The proof of this property is a part of the proof of the correctness of the simulation algorithm (proof 6).

Example 13 *Continuing with the example of Figure 5.6(b), the potentially infinite branch (depicted by bold arrows) which is cut at node $(q_1, (0, 1))$ since*

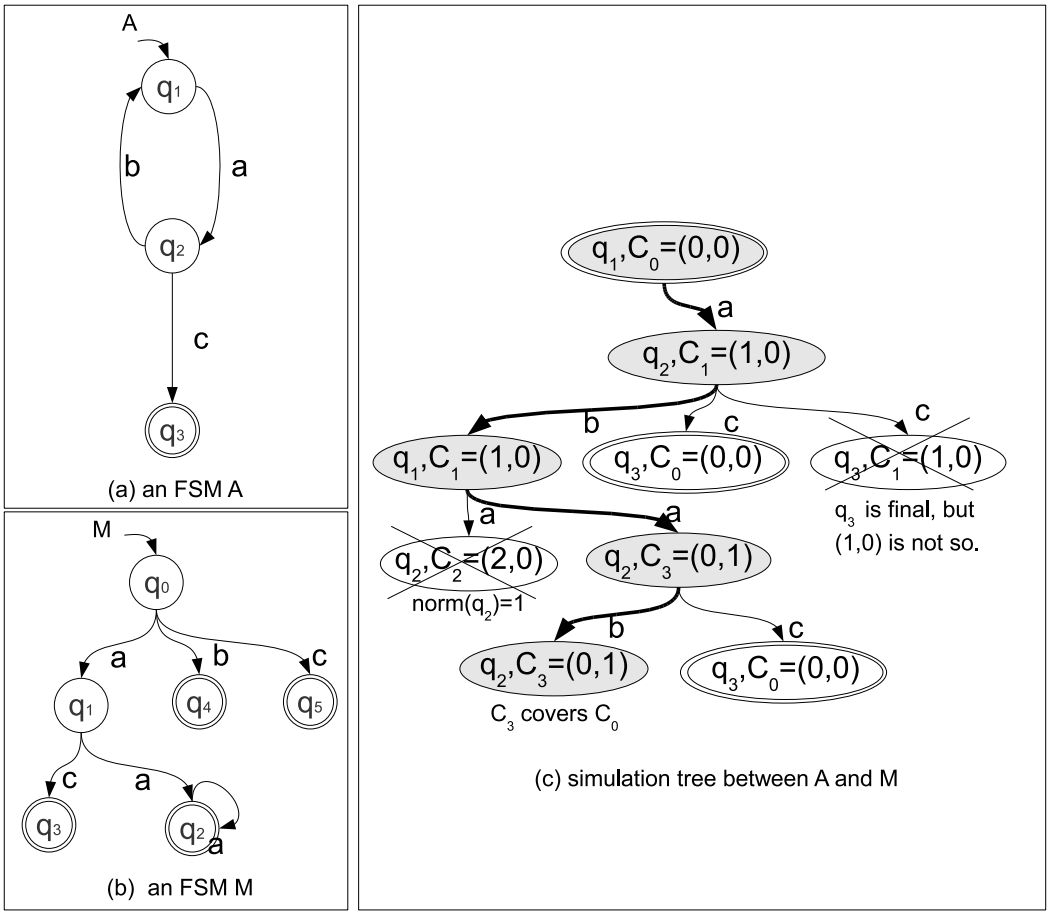

Figure 5.6: Example of a simulation tree.

the configuration $(0,1)$ *is a cover of the configuration* $(0,0)$ *which appear previously in a node* $(q_1,(0,0))$ *in the same branch. Moreover, there exist two ways to cut a branch. The first is when a final, not hybrid, state appears. If the corresponding configuration is final, then we have a positive simulation test. If the corresponding configuration is not final, then we have a negative simulation test (e.g. the node* (q_3,C_1) *is a negative simulation test since* q_3 *is final but not* C_1*). The second way relies on norms of states as illustrated on node* (q_2,C_2)*. We have* $norm(q_2)=1$*, and hence the smallest path that must start from its corresponding configuration (*C_2*) must be able to lead to a final*

configuration by executing only one activity (since $norm(q_2) = 1$*).*

Note that, to verify such a condition, the Check-Cover procedure maintains for each state q in a given branch a list, noted $L(q)$, of all the configurations C' corresponding to the nodes (q, C') of this branch. In our example, we have at node $(q_1, (0, 1))$ of the bold branch the sequence $L(q_1) = [(0, 0), (1, 0)]$.

Algorithm 1: Check-Sim

```
Input: Two FSM A and M, a state q of A, a configuration C of M^⊗
Output: boolean
begin
    if q ∈ F_A \ H_s(A) then
        return(∑_{i=1}^{|I_s(M)|} C[i] = 0);
    if Check-Cover(q,C) then
        Return(true);
    for each transition (q, a, q') in δ_A do
        if not(Check-Candidate(q', C, a)) then
            return(false);
    return(true);
end
```

In the following we will show the termination, the soundness and the completeness of our algorithm.

5.2.2 Termination of the composition algorithm.

The following theorem proves the termination of our algorithm. It mainly relies on the Dickson lemma [Dic13].

Theorem 2 *The algorithm* Check-Sim *halts.*

Proof 5 *Let us suppose that the procedure* Check-Sim *does not halt, i.e. there exists an infinite branch in its execution tree. This means that a given state* $q \in S_M$ *may appear infinitely many times in this branch, i.e.* $|L(q)|$ *is infinite.*

Algorithm 2: Check-Candidate

Input: a state q' of A, a configuration C of $M^{\otimes}$, $a \in \Sigma_M$
Output: boolean
begin
  Candidates=$\emptyset$;
  for *each transition* (C, a, C') *in* ϕ **do**
    if $\sum_{i=1}^{|I_s(M)|} C'[i] \leq norm(q')$ **then**
      Candidates= Candidates $\cup\{C'\}$;
  flag=0;
  while *Candidates*$\neq \emptyset$ *and (not flag)* **do**
    C' =first element in Candidates;
    flag= Check-Sim(q',C');
  return(flag);
end

Algorithm 3: Check-Cover

Input: a state q of A, a configuration C of $M^{\otimes}$
Output: boolean
begin
  for $C' \in L(q)$ **do**
    if $C' \triangleleft C$ **then**
      return(true);
  return(false);
end

Thus $L(q)$ corresponds to a cover-free sequence $(C_i)_{i\in\mathbb{N}}$ of configurations,i.e. $\forall j, k \in \mathbb{N}\ j < k \Rightarrow C_j \not\triangleleft C_k$.

Since the sum of tokens in intermediate states are bounded by $norm(q)$ then $(C_i)_{i\in\mathbb{N}}$ may be split into a finite number of sub-sequences $(C_{i_k})_{i_k\in\mathbb{N}}$, such that for all $C, C' \in (C_{i_k})_{k\in\mathbb{N}}$, $C[j] = C'[j]\ \forall j \in [1, l]$. In other words, each $(C_{i_k})_{i_k\in\mathbb{N}}$ represents a sequence of configurations where the witnesses for intermediate states are the same. Hence, $(C_i)_{i\in\mathbb{N}}$ is an infinite cover-free sequence of configurations iff $(C_{i_k})_{i_k\in\mathbb{N}}$ is an infinite sequence of configurations without inclusion, i.e. $\forall C_1, C_2 \in (C_{ik})_{i_k\in\mathbb{N}}\ \ C_1 \triangleleft C_2 \Leftrightarrow C_1 \subseteq C_2$.

Hereafter, we will prove that such a sequence $(C_{ik})_{i_k \in \mathbb{N}}$ of configurations without inclusion cannot exist. This proof is based on the following lemma established by [Dic13] and reported in [Gal91].

Lemma 3 *[Gal91] Let n be any integer such that $n > 1$. Given any infinite sequence $(C_i)_{i \geq 1}$ of n-tuples of natural numbers, there exists positive integers i, j such that $i < j$ and $C_i \preceq_n C_j$, where $\preceq_n$ is the partial order on n-tuples of natural numbers induced by the natural ordering $\leq$ on $\mathbb{N}$.*

This lemma states that there does not exist an infinite sequence of configurations without inclusion. Thus $(C_{ik})_{i_k \in \mathbb{N}}$ is not infinite without inclusion. Thus we conclude that $(C_i)_{i \in \mathbb{N}}$ can not be an infinite without cover and therefore the procedure Check-Sim halts.■

5.2.3 Correctness of the composition algorithm

Theorem 2 states that our algorithm is correct, i.e. sound and complete.

Theorem 3 *The Algorithm Check-Sim is correct.*

Proof 6

- *Soundness. Suppose that Algorithm Check-Sim returns true. We show that there exists simulation between q_0 and C_0, and thus, there exists simulation between A and $M^{\otimes}$. Let us consider a call to the Algorithm Check-Sim with q a state of A and C a configuration of $M^{\otimes}$. We can distinguish three acceptance cases :*
  - *$q \in F_A \setminus H_s(A)$ and $(\sum_{i=1}^{|I_s(M)|} C[i] = 0$; i.e. C is final). Then $q \preceq C$.*
  - *For each transition $(q, a, q') \in \delta_A$: q' is simulated by a given C'. Then $q \preceq C$.*
  - *$Cover(q, C) = 1$. This case represents the difference between our algorithm and classic simulation algorithms. It corresponds to an execution of a loop in A which goes through q. That is to say,*

there exists a sub-path in the execution tree from (q, C) to (q, C') such that $C \triangleleft C'$. This cover condition allows us to avoid the test of simulation between q and C', because C' possesses the same number of tokens as C on intermediate states of M and more tokens than C for hybrid states of M. Since C' and C need to simulate the same state q, we can restrict C' to be equal to C by deleting the extra tokens in hybrid states of C'.

- *Completeness.* *Now suppose that Algorithm* Check-Sim *returns false. First we show that Algorithm* Check-Sim *looks for all the possibilities to simulate the state q by a configuration C. In order to simulate q by C, the Algorithm* Check-Sim *checks for each transition (q, a, q') in δ_A if q' can be simulated by a configuration C' such that $(C, a, C') \in \phi$. The Algorithm* Check-Candidate *computes all configurations that may be candidate to simulate q'. Candidates that do not satisfy the condition $\sum_{i=1}^{|I_s(M)|} C'[i] \leq norm(q')$ are rejected. Indeed, these configurations cannot simulate q', since there exists a path from q' to a final state in F_A such that the tokens on the intermediate states of C' cannot be all consumed. From the list of candidates, the Algorithm Check-Candidate try to find a candidate configuration that simulates q'. The algorithm returns false if no such a configuration exist.*

  Now suppose that the Algorithm Check-Sim *returns false. We distinguish two cases:*

  - *$q \in F_A \setminus H_s(A)$ and $\sum_{i=1}^{|I_s(M)|} C[i] \neq 0$. This means that q is a final state and C is not a final state. Thus q cannot be simulated by C.*
  - *There is a transition (q, a, q') in δ_A such that the state q' cannot be simulated. Since all candidate configurations C' such that $(C, a, C') \in \phi$ are checked, we conclude that q cannot be simulated by C.*

We conclude that Algorithm Check-Sim *is correct.*■

It is worth noting that the proposed proof is constructive in the sense that if the answer is true, the algorithm may be easily modified to exhibit a simulation relation between its inputs. This is an interesting point in the context of the protocol synthesis problem since such a simulation relation can be effectively used to build a delegator.

Theorem 4 *Let A and M be two FSMs. It is decidable whether $A \preceq M^{\otimes}$.*

Finally, in the following corollary, we derive the main result of this work regarding the addressed web service composition problem.

Corollary 1 *Let S_T be a target protocol and $\mathcal{R}$ be a repository of protocols. The problem $Compose(\mathcal{R}, S_T, +\infty)$ is decidable.*

5.3 Complexity analysis

In this section, we investigate the complexity of the GCSP. A lower bound can be derived immediately from existing work [MW07]. We provide an Ackermannian [Wic76] upper bound that makes our complexity non-elementary. We then turn our attention to a study of two particular cases of the GCSP, namely the case where target services have no loops, and the case where the component services are without hybrid states.

5.3.1 Complexity bounds

We first provide a rather trivial result on the lower bound derived from [MW07] and then we focus on the upper bound. Indeed, the GCSP is **exptime-hard** since the bounded instances composition synthesis studied in [BCG+03, MW07] is exptime-complete. The latter problem is nothing else than a particular case of the GCSP.

Henceforth, we consider an FSM $A =< \Sigma, Q_A, F_A, q_A^0, \delta_A >$ and a PCSM $M^{\otimes} =< \Sigma, \mathcal{C}, F_C, C_0, \phi >$ where $M =< \Sigma, Q_M, F_M, q_M^0, \delta_M >$. We assume that M corresponds to $\odot\mathcal{R}$ where $\mathcal{R} = \{P_1, ..., P_n\}$ is a repository of web services protocols. We focus on the problem of testing a simulation of A by $M^{\otimes}$.

The composition algorithm (algorithm 1) generates a simulation tree ($\mathcal{T}$) where nodes are made of pairs (state, configuration) : (q, C) with $q \in Q_A$ and $C \in \mathcal{C}$. In order to provide an upper complexity bound for our problem, we need to evaluate the maximum size of $\mathcal{T}$. The branching factor at each node (q, C) (the number of successor nodes) does not exceed the number of the transitions from M. Hence, the difficulty to evaluate the tree size is the one of evaluating the depth of its longest path. In the following, we will focus on this computation.

We start by encoding $\mathcal{T}$ to uphold an existing result (lemma 4) proposed by [Soc91, GKOS08]. This lemma provides an upper bound on the length of dicksonian sequences. A sequence of non-negative integer n-tuples t_1, t_2,...,t_k is called dicksonian, if for all $1 \leq i < j \leq k, (t_j - t_i)$ has at least one negative coordinate. In order to make benifit of this lemma, we will express a branch of $\mathcal{T}$ by mean of a Dicksonian sequence. Note that, the finiteness of a Dicksonian sequence relies on the partial order ($\leq_n$) on n-tuples of natural numbers induced by the natural ordering ($\leq$) on $\mathbb{N}$. Indeed, this does not apply directly to our notion of *cover*. That motivates us to encode our simulation tree ($\mathcal{T}$) to a new one ($\mathcal{T}'$) such that the cover checking between nodes from ($\mathcal{T}$) is equivalent to checking $\leq_{\mathbb{N}}$ over the corresponding nodes from $\mathcal{T}'$. More precisely, the branches of the encoded simulation tree $\mathcal{T}'$ must uphold two constraints : *(i)* when testing the simulation, the branches will be cut using the preorder ($\leq_n$) and not using the cover any more, and *(ii)* The difference of the maximal coordinates (we denote DMC for short) of two successive tuples does not exceed 1. We call these constraints the $\leq_n$ *constraint* and the *DMC constraint* respectively. The lemma on which we build our computation is

provided below.

Let $L_{f,n}$ denote the maximal length of a dicksonian sequence of n-tuples, whose maximal coordinates are bounded by a function f. For a function $f : \mathcal{N} \to \mathcal{N}$, let $f^{-1}(x)$ be the least number k such that $f(k) \geq x$.

Lemma 4 *[GKOS08] Let $f : \mathcal{N} \to \mathcal{N}$ be an increasing function, and $d \in \mathcal{N}$ be a number such that $\forall i > 0 : f(i+1) - f(i) \leq A(d, f(i) - 1)$. Then $L_{f,n} < f^{-1}(A(n+d, f(1) - 1)$ and the maximal entry of the last n-tuple does not exceed $A(n+d, f(1) - 1)$.*

We derive below a particular case of the previous lemma that fits well our settings. More precisely, we consider $f(i) = i$ and $d = 0$. Hence, we allow the maximal coordinate on a vector to increase at most by $f(i+1) - f(i) = 1$ at each step (i.e. invocation of Check-Sim).

Proposition 2 *Let $f : \mathcal{N} \to \mathcal{N}$ be an increasing function, such that $\forall i > 0 : f(i+1) - f(i) \leq 1$. Then $L_{f,n} < A(n, 0)$ and the maximal entry of the last n-tuple does not exceed $A(n, 0)$.*

Hereafter, we will provide the tree encoding that will allow us to apply the proposition 2. We assume that the states from Q_A are numbered from 1 to $|Q_A|$, and hence a state can be represented by uniquely its index. A node (q_k, C) from $\mathcal{T}$ is a tuple $(k, m_{i_1}, ..., m_{i_I}, h_{j_1}, ..., h_{j_H})$ where $I = I_s(M)$ and $H = H_s(M)$. We propose below an encoding with respect to the previous tuple that will have the form (V_1, V_2, V_3) where $V_{i, i \in [1,3]}$ is a vector of integers.

- V_1 is a vector of $|Q_A|$-integers and informs which state $q_k \in Q_A$ is present in the node. In fact, we encode a state $q_k \in Q_A$, $i \in [1, |Q_A|]$, as follows : *(i)* $V_1[k] = 1$ and *(ii)* $V_2[j]_{j \neq k, j \in [1, |Q_A|]} = 0$. The intuition behind this encoding is to prevent the DMC to exceed 1, i.e. verify the *DMC constraint*. Secondly, instead of testing the equality between states of two nodes, it is sufficient to test the 'less or equal' relation

between the vectors $\{V_1\}$, i.e. verify the $\leq_n$ constraint. For instance, assume that $|Q_A| = 5$ and there exists $q_1, q_5 \in Q_A$ and $C, C' \in \mathcal{C}$ such that $N = (q_1, C)$ is a node from $\mathcal{T}$ and $N' = (q_5, C')$ is an immediate successor of N. Hence, the DMC between N and N' is equal to 4. Once the encoding applied, we obtain two new nodes $E = (1, 0, 0, 0, 0, C)$ and $E' = (0, 0, 0, 0, 1, C')$ corresponding to N and N' respectively. Note that, both constraintes hold.

- V_2 is a vector of integers formed by sub-vectors $(V_{2,1}, ..., V_{2,I})$, where each $V_{2,i}$, $i \in [1, I]$ is formed of $Norm(Q_A)$ integers and informs about the value of m_i. In fact, we use the same encoding approach as in the vector $\{V_1\}$. Indeed, if $m_i = k$, $k \in [1, Norm(Q_A)]$, then *(i)* $V_{2,i}[k] = 1$ and *(ii)* $V_{2,i}[j]_{j \neq k, j \in [1, Norm(Q_A)]} = 0$.

- V_3 is a vector of H-integers and equals to $(h_{j_1}, ..., h_{j_H})$. Indeed, no encoding is required with respect to the hybrid states because their corresponding components uphold both constraints.

Finally, from the previous encoding, we can evaluate the size of tuples to $\alpha = (|Q_A| + Norm(Q_A) * I + H)$ integers. All of them range over 0 or 1 except the last H integers, i.e. the ones that model the hybrid states.

Example 14 *As an example of the encoding procedure, we propose to encode the simulation tree depicted at figure 5.6. On figure 5.7-(a) we recall the concerned simulation tree, and figure 5.7-(b) depicts the associated encoded tree. The encoding of states of the target A states occupies the three first positions of each vector since A has three states. The following two integers inform about the number of tokens of the unique intermediate state from the FSM associated with the PCSM. The last integer is the representation of the token number on the hybrid state of the latter. Note that for this example, we have* $\alpha = 3$.

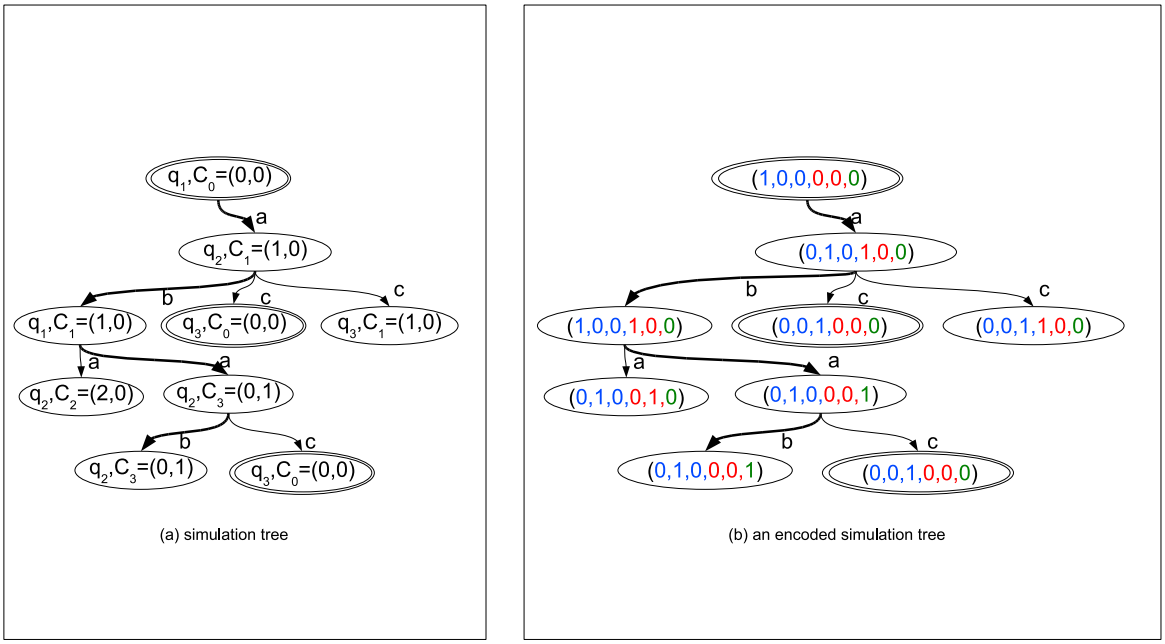

Figure 5.7: Example of the encoding of a simulation tree.

Before turning our attention to the theorem on the maximal sequence length, we comment the correctness and the usefulness of our encoding. Firstly, it is easy to see that the maximal DMC does not exceed 1 between successive nodes. Secondly, let (q, C) and (q', C') be two nodes of $\mathcal{T}$, and E and E' be their corresponding nodes. We recall that we cut a branch of $\mathcal{T}$ due to a cover if and only if $(q = q')$ and $(C \lhd C')$. By construction, the latter conjunction of conditions is equivalent to $(E \leq_\alpha E')$. This means that, in the encoded tree, it is enough to compare nodes basing on a simple $(\leq_\alpha)$. Hence, we conclude that an encoding procedure is correct and an encoded tree fits well the settings of propositoin 2.

Now that we have a simulation tree $\mathcal{T}'$ for which the increasing of the maximal coordinate does not exceed 1 and the branches are cut by doing a $\leq_\alpha$ test, we are able to provide below the theorem 5 which gives the upper complexity bound on the length of the longest path in such a tree.

Theorem 5 *Let* $A =< \Sigma, Q_A, F_A, q_A^0, \delta_A >$ *be an FSM and* $M^{\otimes} =< \Sigma, \mathcal{C}, F_C, C_0, \phi >$ *be a PCSM, where* $M =< \Sigma, Q_M, F_M, q_M^0, \delta_M >$*. The length of the longest path in the simulation tree of* A *by* $M^{\otimes}$ *does not exceed* $A(\alpha, 0)$*.*

Proof 7 *The proposition 2 and our encoding leads us to the following : the maximal coordinate in the last tuple is equal to* $A(\alpha, 0) = A(H_s(M) + I_s M * Norm(Q_A) + |Q_A|, 0)$*.* ■

Corollary 2 *Let* $A =< \Sigma, Q_A, F_A, q_A^0, \delta_A >$ *be an FSM and* $M^{\otimes} =< \Sigma, \mathcal{C}, F_C, C_0, \phi >$ *be a PCSM, where* $M =< \Sigma, Q_M, F_M, q_M^0, \delta_M >$*. The size of the underlaying simulation tree of* A *by* $M^{\otimes}$ *does not exceed* $|\delta_M|^{A(\alpha,0)}$*.*

Proof 8 *We have (i) the branching factor at each node is less or equal to* $|\delta_M|$ *and (ii) the length of the longest path in the simulation tree does not exceed* $A(\alpha, 0)$*. This implies that the size of the simulation tree does not exceed* $|\delta_M|^{A(\alpha,0)}$*.*

5.3.2 Complexity study of particular cases

Hereafter, we focus on two particular cases of the generic composition synthesis problem, namely *(i)* the case where target services are without loops and *(ii)* the case where component services are without hybrid states.

Case 1 : target services without loops

Let us now consider the case where target services are without loops. We will show the NP-Completeness of this problem. The NP-hardness is proved by reducing the problem 4 of the inclusion of a finite word in the language generated by the shuffle closure of another finite word. This problem is known to be NP-Complete from [JS01].

Problem 4 • ***Input:*** *u and v two finite words over an alphabet* Σ*.*

- ***Output:*** *a boolean result that informs whether* $L(u) \subseteq L(v^{\otimes})$*, where* $L(u)$ *and* $L(v^{\otimes})$ *are the languages built on* u *and* $v^{\otimes}$ *respectively.*

Our problem belongs to NP because we can validate a certificate on the computed simulation relation in a linear time of the size of the target service. This fact is true since the target service has no loops and hence the computed simulation relation has exactly the same number of transitions as the target service, and at most the same number of states. We will start by providing the problem considered in [JS01].

Theorem 6 *Let* $A =< \Sigma, Q_A, F_A, q_A^0, \delta_A >$ *be an FSM and* $M^{\otimes} =< \Sigma, \mathcal{C}, F_C, C_0, \phi >$ *be a PCSM, where* $M =< \Sigma, Q_M, F_M, q_M^0, \delta_M >$*. If* A *has no loops, then the problem* $A \preceq M^{\otimes}$ *is NP-Complete.*

Proof 9 ***NP.*** *Since* A *has no loops, the simulation algorithm will pass only once through each transition. Hence the computed simulation relation will have exactly the same number of transitions as the target service. In other words, the size of the computed solution is equla to the size of the input.*

NP-hardness. *An instance of the problem 4 is a particular case of our problem where target services are without loops. Indeed, we consider the word* u *as being a target service* A *with a unique path from the initial state to the final one, that corresponds to the word* u*. the word* v *will correspond to* M*, which contains a unique path from the initial state to the final one, that corresponds to the word* v*. Hence, we have* $L(u) = L(A)$ *and* $L(v) = L(M)$*. The fact that all services are linear implies that both the simulation existence and the language inclusion problems are equivalent since no branching structure does exist.*■

Case 2 : component services without hybrid states

We provide the complexity of the restrictive case of the synthesis composition problem where the component services are without hybrid states. The primitive recursiveness of this case is trivial since the simulation algorithm termination does not rely on the Dickson lemma.

Theorem 7 *Let* $A =< \Sigma, Q_A, F_A, q_A^0, \delta_A >$ *be an FSM and* $M^{\otimes} =< \Sigma, \mathcal{C}, F_C, C_0, \phi >$ *be a PCSM, where* $M =< \Sigma, Q_M, F_M, q_M^0, \delta_M >$. *If* M *has no hybrid states, then there exists an algorithm to solve the problem* $A \preceq M^{\otimes}$ *in* $O((|Q_M|^{Norm(Q_A)} + |Q_A|) \times (|Q_M| + |\delta_M|^{Norm(Q_A)} + |\delta_A|))$.

Proof 10 *We base our proof on the evaluation the maximal part of* $M^{\otimes}$ *that is involved in a simulation. Indeed, having no hybrid states in* Q_M *allows us to estimate a priori the maximal number of parallel instances that can participate in a simulation. At any instant* t, *the number of running instances of* M *can not exceed* $Norm(Q_A)$ *since all states of* M *are intermediate. Hence, it will be enough to duplicate each component service* $Norm(Q_A)$ *times to obtain the maximal set of parallel services instances (copies) which will be used in a simulation. We allow each of these copies to run an unbounded number of sequential instances by adding* ϵ*-transitions on* M *as explained previously. In other words, we will check the simulation between* A *and* $\cup_{j=1}^{Norm(Q_A)} M'^{\otimes_j}$, *where* M' *is* M *with* ϵ*-transitions. The states number of* $\cup_{j=1}^{Norm(Q_A)} M'^{\otimes_j}$ *is* $|Q'_M|^{Norm(Q_A)} = |Q_M|^{Norm(Q_A)}$ *and it has* $(|\delta_{M'}|^{Norm(Q_A)} \leq |Q_M| + |\delta_M|^{Norm(Q_A)})$ *transitions.*

The best known simulation algorithm is provided by [HHK95] and has a complexity of $O(number\ of\ states \times number\ of\ transitions)$. *Hence, we can derive the following complexity* $O((|Q_M|^{Norm(Q_A)} + |Q_A|) \times (|Q_M| + |\delta_M|^{Norm(Q_A)} + |\delta_A|))$.■

5.4 Discussion

In this chapter we presented the main contributions of this thesis concerning the decidability and the complexity of the composition synthesis problem. The first result is that the GCSP is decidable and more precisely exptime-hard. We provide an ackermannian upper bound for the GCSP complexity. We study two praticular cases of the problem, namely where *(i)* target services have no loops, or *(ii)* component services have no hybrid states.

In the following, we discuss the connection between our decidability result and a similar one on the simulation of FSMs by Petri nets (PNs) proposed by [Jan95]. We recall that a PCSM is a particular form of Vector Addition Systems [KM69] or equivalently Petri nets [Pet73]. It is worth noting that the existing result on the decidability of testing a simulation between FSMs and PNs is known for the class of Petri nets without final markings (final states). The presence or the absence of final markings in Petri nets can alter the decidability of a given problem. For instance, testing the trace inclusion between FSMs and PNs is decidable whilst testing the language inclusion between FSMs and PNs is undecidable [JM95]. Indeed, the difference between both problems is that the latter consider traces with final markings. Within our context, the definition of final configurations, basing on intermediate and hybrid state, has suitable properties that allow us to propose the *cover* as preorder to compare configurations. The cover has the advantages of allowing to cut branches by ensuring a simulation and of being finitely computable. The existence of such kind of preorders w.r.t any (infinite) set of final markings, that can be defined in several ways, is still an open decidability problem. It represents an interesting theoretical perspective for this work beyond the composition context.

Six

Formal background

This chapter will start by providing a panorama of various labeled transition systems we encountered and dealt with during this thesis. Then, we try to summarize existing results regarding important problems such as emptiness checking, universe and closure under complement. Also, we mention existing efforts on the problems testing the following preorders: language inclusion, simulation and bisimulation. The study of such proprieties and preorders is highly interesting with respect to the composition problem; as can witness [FGG+08] where both emptiness checking and language equivalence are used to compare web services behaviors.

6.1 Panorama of Models

In this chapter, we will study the following state machines formalisms : Finite State Machines (FSM) [HU69], Product Closure State Machine (PCSM) [RNT], Simple Shuffle Automata (SSA) [Jed87], Shuffle Automata (SA) [Jed99], Push Down Automata (PDA) [HU69], Linear Bounded Automata (LBA) [HU69], Basic Parallel Processes (BPP) [KM02a], Vector Addition Systems (VAD) [KM69] and Petri Nets (PN) [Pet73]. Of course, all of these state

machines are sub-models of the Turing machine. We also denote the languages of each class by $L(Name\ of\ the\ state\ machine)$ (e.g. $L(PN)$ is the class of languages of Petri nets).

The figure 6.1 details relationships between all the aforementioned models. Arrows directions indicate the 'sub-model' relation. The absence of arrows means that the concerned models are incomparable. The dashed arrows highlight the well known Chomsky classification of the most used machines classes. Various references led to us to derive the figure 6.1 (e.g. [Gis81, KM69, HU69, AKT81]).

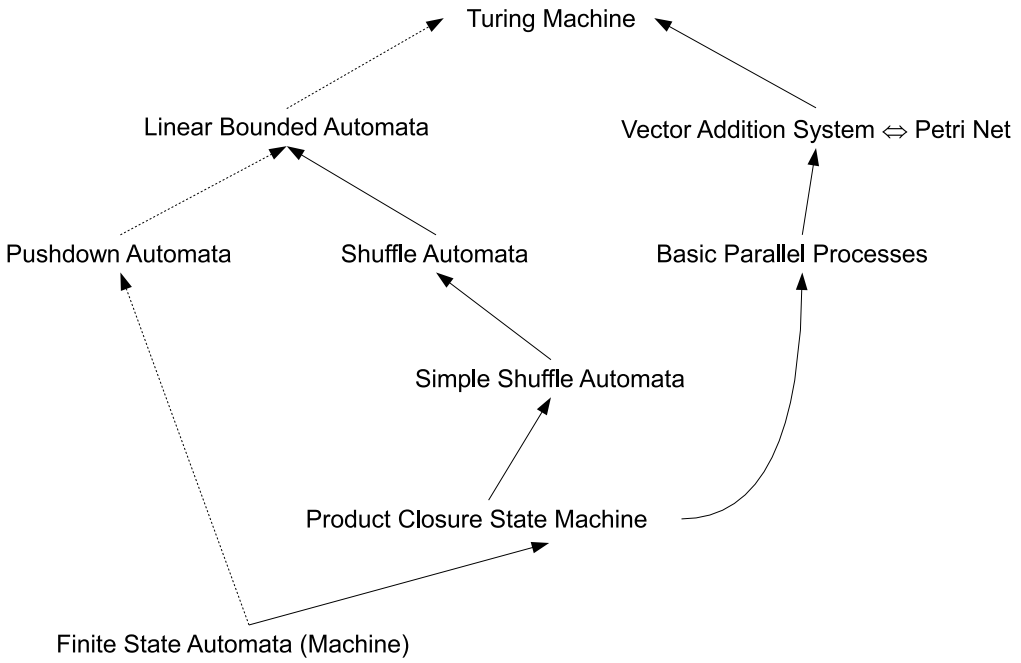

Figure 6.1: Hierarchy of models

FSMs are a suitable model to represent the individual behavior of each component service and the target one as well. However, they lack the expressive power to model the behavior generated by the possible interactions

between an unbounded number of instances of each service (i.e. the unbounded parallel behavior, or the parallel behavior for short). PDAs are a more expressive model than FSMs because they have stacks that offer a kind of counting mechanism and PDAs allow to recognize context-free languages. However, PDAs are easily discarded since they are not able to recognize rather simple languages (e.g. $(abc)^{\otimes}$) that are recognized by a PCSM and not context-free ones. Going up within the Chomsky hierarchy leads to the LBA, a highly expressive state machine even more expressive than our needs. Unfortunately, most proprieties on the LBAs are undecidable. At this level, the aim is at characterizing the exact sub-model of LBAs that is required to model parallel behavior. SAs are again a suitable state machine formalism to model the required parallel behavior but the SAs research area are rarely concerned by the simulation preorder. A deep look inside the SAs theory inspired us the proposition of the PCSM. Interestingly, PCSMs are a sub-model of PNs (that are incomparable with SAs) and hence we can exploit the existing results in the field. Finally, two super-models of PCSMs deserve to be cited : BPPs and SSAs. These machines play a symetric role : the BPPs are the smallest sub-class of PNs that are not SAs, and SSAs are the smallest sub-class of SAs that are not PNs.

6.2 Languages problems

This section provides a set of properties of each class of language/model from the ones cited before. The studied properties are :

- Inclusion : tests whether the inclusion between two languages from a given class is decidable. Let L_1 and L_2 be two languages, L_1 is said to be included in L_2, noted $L_1 \subseteq L_2$, iff $\forall \omega \in L_1 : \omega \in L_2$.
- Universe problem : tests whether it's decidable or not to check whether the language from a given class equals the universe language (Σ^*). Let L_1

be a language, L_1 is said to be the universe language iff $\forall \omega \in \Sigma^* : \omega \in L_1$.

- Emptiness : tests whether the emptiness of a language from a given class is decidable. Let L_1 be a language, L_1 is said to be empty iff $\forall \omega \in \Sigma^*, \omega \notin L_1$ or equivalently $L_1 = \phi$.

- Intersection emptiness : tests whether the emptiness of the intersection of two languages from a given class is decidable. Let L_1 and L_2 be two languages, the intersection of L_1 and L_2 is said to be empty iff $\forall \omega \in \Sigma^* : \omega \notin L_1 \cap L_2$.

- Complement : informs whether the class is closed under the complement operator. Let L_1 be a language, its complement $\overline{L_1}$ is defined such that $\forall \omega \in \Sigma^* : \omega \notin L_1 \Rightarrow \omega \in \overline{L_1}$ or equivalently $L_1 \cap \overline{L_1} = \phi$.

- Intersection : informs whether the class is closed under the intersection operator. Let $\mathbb{C}$ be a class of languages (e.g. regular languages), $\mathbb{C}$ is said to be closed under intersection iff $\forall L_1, L_2 \in \mathbb{C} : L_1 \cap L_2 \in \mathbb{C}$.

In order to avoid possible confusions, we recall that the class of Petri nets considered in this thesis is the labeled, marked Petri net with final markings [Pet81]. The presence of the final markings is required to allow the study of the languages properties.

Table 6.1 summarizes known results with a selected reference for each. For the non referred cells, the exponent points out an intuition of the proof on the concerning result. These proofs sketches can be found just after the table. Inside the table and on columns, we refer all language classes of interest : regular languages ($L(FSM)$), context-free languages ($L(PDA)$), shuffle languages ($L(SA)$), Petri net languages ($L(PN)$) and context-sensitive languages ($L(LBA)$). On lines, one can find the studied properties, and the cells indicate whether the property (on the line) is *decidable or not* for the language (on the column). Note that for the last two lines of the table, the answer is

closed or not rather than *Decidable or not*. Finally, D stands for decidable, U for undecidable, C for closed and N for not closed.

	FSM	**PDA**	**SA**	**PN**	**LBA**
Inclusion	D[UH79]	U[GR63]	U[4]	U[JM95]	U[HU69]
Universe	D[1]	U[GR63]	U[Iwa82]	U[7]	U[HU69]
Emptiness	D[2]	D[GR63]	D[Iwa82]	D[8]	U [HU69]
Intersection emptiness	D[3]	U[HAR67]	U[Jke96]	D[9]	U [HU69]
Complement	C[UH79]	N[HAR67]	N[5]	N[10]	C[HU69]
Intersection	C[UH79]	N[HAR67]	N[6]	C[RV82]	C[HU69]

Table 6.1: Properties of the languages classes.

In the following we provide details about the numbers annotating the results in the previous table. Our goal is to recall in intuitive way the known proofs on these results.

1. Let R be a regular language. We have $R \subseteq \Sigma^*$ and hence to test whether $R = \Sigma^*$ it is enough to test whether $\Sigma^* \subseteq R$. Recall that the universe language is itself a regular one and then testing whether $\Sigma^* \subseteq R$ is equal to test the inclusion between two regular languages which is a decidable problem. Finally, we conclude that the universe problem is decidable for the class of regular languages.

2. The class of regular languages is closed under complement. In order to test the emptiness of a regular language, it is enough to test the universality of its complement. Since the universe problem is decidable for regular languages and the complement is computable, so the emptiness problem is decidable.

3. The class of regular languages is closed under intersection, and hence testing the emptiness of the intersection of two regular languages is equal

to test the emptiness of a regular language. The latter is known to be a decidable problem.

4. Again, the universe language is a regular one. Hence, the universe language is a shuffle language. The undecidability of the universe problem induces the fact that testing the inclusion between a regular language and a given shuffle language is undecidable. Hence the inclusion between two shuffle languages is undecidable.

5. The class of shuffle languages is not closed under intersection. Hereafter, we assume this class to be closed under complement. We know that the shuffle languages are closed under union, and hence the union of the complement of two shuffle languages ($L_1 \cup L_2$) will be a shuffle language (L_3). One can negate the previous form, and obtain some thing like the intersection of any two shuffle languages is a complement of a shuffle language ($\overline{L_1} \cap \overline{L_2} = \overline{L_3}$) and hence a shuffle language (by assumption). This fact leads us to a contradiction with the non-closure of the class under intersection. Thus, shuffle languages can not be closed under complement.

6. We consider the following shuffle languages : a^*b^*c and $(abc)^{\otimes}$. It's easy to see that their intersection is $a^nb^nc^n$. A language that is a shuffle language, but a context-sensitive one. The latter language allows also to prove that the class of shuffle languages is a *proper* subset of the class of context-sensitive ones ($\mathbb{L}(SA) \subset \mathbb{L}(LBA)$).

7. The inclusion between regular languages and Petri net languages is undecidable[JM95]. As illustrated previously (in 1), it is easy to derive the undecidability of the universe problem.

8. The emptiness checking for a Petri net is equivalent to the reachability problem where the target marking is the final one. Recall that the set of

all final markings can be reduced to only one equal to the zero marking. From [Lip76], we know that the reachability problem is decidable, and thus the emptiness one is decidable as well.

9. The class of Petri nets languages is closed under intersection. It results that hence testing the emptiness of the intersection of two Petri nets languages is equal to test the emptiness of a Petri net language. The latter is known to be a decidable problem.

10. Let PN be a Petri net. Note that the emptiness of PN is decidable but the universe problem is not so. Let us suppose that $\overline{PN}$ is a Petri net. We have $PN \stackrel{?}{=} \phi$ is decidable and this implies that $\overline{PN} \stackrel{?}{=} \Sigma^*$ is also decidable. This engenders a contradiction with the assumption. Hence, the class of Petri nets is not closed under complement.

6.3 Properties of the PCSM languages

In this section, several properties of $\mathbb{L}(PCSM)$ are depicted, mainly the problems of emptiness checking and universe checking, and finally closure of this class under intersection operator. The rest of the considered properties (e.g. the closure under complement) are still open.

Lemma 5 *Emptiness checking for the class $\mathbb{L}(PCSM)$ is polynomial.*

Proof 11 *Let R be an FSM. Then $L(R^{\otimes}) = \emptyset$ iff $L(R) = \emptyset$. Emptiness checking for regular languages is a simple instance of graph-reachability problem and known to be polynomial [HMU01].*

Lemma 6 *Universe checking for $\mathbb{L}(PCSM)$ is linear.*

Proof 12 *Let R be an FSM, $L(R^{\otimes}) = \Sigma^*$ iff $\Sigma \subseteq L(R)$.*

- ***(if)*** *Since* $\Sigma \subseteq L(R)$*, so the language of* R *can be written as* $\Sigma \cup L(R')$*, with* R' *an FSM. Thus* $L(R^{\otimes}) = L(\Sigma \cup R'^{\otimes})$*. We have* $(\Sigma)^{\otimes} = \Sigma^*$*, thus* $L(\Sigma \cup R')^{\otimes} = \Sigma^* = L(R^{\otimes})$
- ***(only if)*** $L(R^{\otimes}) = \Sigma^*$ *so all words from* Σ^* *are in* $L(R^{\otimes})$*. We focus on words from* Σ^* *that have the length is* 1 *(i.e.* Σ*). Since words of length* 1 *from* $L(R^{\otimes})$ *can only appear in* $L(R)$ *thus,* $\Sigma \subseteq L(R)$*.*

Lemma 7 *The class* $\mathbb{L}(PCSM)$ *is not closed under intersection.*

Proof 13 *Let us consider an alphabet* $\Sigma = \{a, b, c\}$*, and two shuffle closure expressions* $S_1 = a^*b^*c^*$ *and* $S_2 = (abc)^{\otimes}$*. We have* $S_1 \cap S_2 = a^n b^n c^n$*, with* n *an integer. The last language is context-sensitive language and not a language from the class* $\mathbb{L}(PCSM)$*.*

6.4 Language inclusion, Simulation and Bisimulation

The section deals with the language inclusion problem regarding the classes of languages introduced in the previous section and studies the simulation/bisimulation problem for some models of interest as well.

6.4.1 Language inclusion decidability problem

Table 6.2 summarizes the known results about language inclusion among many of the presented languages classes. In order to not hamper the table, we deleted columns/lines corresponding to context-sensitive languages since all the inclusions, in both directions, are undecidable [HU69]. As previous, inside the cells D stands for decidable problems and U stands for undecidable problems.

	FSM	**PDA**	**SA**	**PN**
FSM	D[UH79]	U[HU69]	U[1]	U [JM95]
PDA	D [HU69]	U[2]	U [1]	U[3]
SA	- [4]	U[2]	U[1]	U[3]
PN	D[JM95]	U[2]	U[1]	U[3]

Table 6.2: Results on the language inclusion preorder.

1. Since the universe problem is undecidable for shuffle languages[Iwa82], one can conclude easily the undecidability of the language inclusion problem between regular and shuffle languages. Since all the rest of present languages classes are super-classes of the regular languages, the undecidability result follows for all of them with regard to shuffle languages.

2. The inclusion problem is undecidable between regular and context free languages. Hence, this is true for the classes of languages that contain the class of regular languages.

3. Same as in 2.

4. Up to our knowledge, the inclusion of shuffle languages in regular languages is still an open problem.

6.4.2 Simulation and bisimulation decidability problems

It should be noted that the considered PNs here are the marked labeled Petri nets *without final markings*. This consideration is motivated because most of existing results on the field of simulation/bisimulation concern this category of PNs. An important aspect to point out is that the presence of final markings or states can alter decidability results. For instance, we mention the decidability of the trace inclusion and the undecidability of language inclusion of regular languages in Petri nets languages [JM95]. Indeed, the main

	FSM	**PDA**	**BPP**	**PN**
FSM	Polynomial [HHK95]	exptime-comp [KM02c]	pspace-hard [KM02a]	D [JM95]
PDA	exptime-comp [KM02c]	U [KJ06]	-	-
BPP	coNP-Hard [KM02a]	-	U[Hfi94]	U[1]
PN	D [JM95]	-	U[1]	U[1]

Table 6.3: Results on the simulation preorder.

difference between the trace and the language inclusion is the consideration of final markings in the latter. Table 6.3 summarizes known results about the simulation relation for state machines models of interest. Table 6.4 summarizes most of known results about the bisimulation relation for machines models of interest. The cells marked by $'-'$ stand for problems that are, up to our knowledge, still open. Note that the LBAs are omitted since, up to our knowledge, all the existing simulation/bisimuation problems are undecidable. The SAs are omitted since, up to our knowledge, no simulation/bisimulation results do exist.

1. All are immediate results of the undecidability of the simulation of a BPP by a BPP. Recall that the BPP class is a sub class of Petri nets.

6.5 Conclusion

In this chapter, we covered main existing results that have connections to our work. We attempt also to illustrate many interesting aspects of the theory

	FSM	**PDA**	**BPP**	**PN**
FSM	Polynomial [HHK95]			
PDA	pscpace-comp [KM02c]	exptime-Hard [KM02c]		
BPP	Polynomial [KS05]	-	PSPACE-Complete [Jan03]	
PN	D[JM95]	-	-	U [KJ06]

Table 6.4: Results on the bisimulation preorder.

of automata and transition systems. We started by giving an overview of the main known state machines. Next, we have illustrated many decidability results about properties of machines languages such as the universe and the emptiness problems. We also provided many decidability results with regard to the three most studied pre-orders among these classes, namely: language inclusion, simulation and bisimulation.

Seven

Prototyping

In this chapter we detail the implementation of the composition algorithm that we proposed in this thesis. As mentioned before, our composition algorithm is not primitive recursive. This motivates us to run an empirical evaluation of the composition synthesis algorithm performances in order to evaluate its real-use possible scales. We also studied the time complexity sources that allows us identify parameters (e.g. state number per component service) that can make instances easy or hard to solve.

We implemented our algorithm as part of ServiceMosaic[1], a model-driven prototype case tool for modeling, analyzing, and managing web services. We developed two main components: (i) WS-protocol-generator that enables to generate synthetic web service protocols according to several input parameters, such as the number of transitions per services, number of services, etc., and (ii) WS-protocol-composer that is an implementation of our composition algorithm. These components have been implemented using the JavaTM platform version 6 and the Eclipse framework.

This chapter is divided into two sections. The first section describes the prototype and its environment of development. The second motivates and

[1] http://servicemosaic.isima.fr

explains the test sets we have run. It provides an analysis on the obtained performances curves and their interpretation as well. We end this chapter by a discussion on the learned lessons from this evaluation.

7.1 Prototype

The implementation was done using the Java platform 5 and 6 and run under the Eclipse environment[2]. ServiceMosaic is the umbrella project under which our prototype has been implemented. The ServiceMosaic project is an international academic platform that provides facilities for modeling, analyzing, discovery and adaption of web services models[BCT+06a] (e.g. business protocols, timed business protocols). The involved research groups are from the University of New South Wales (Sydney, Australia), the University Blaise Pascal (Clermont-Ferrand, France), the University Claude Bernard Lyon (France) and the University of Trento (Italy). This platform comprises the following components that can be all accessed through a set of programmatic SOAP web service interfaces [MNSPB+07]:

- **Models and manipulation components** support representing, storing and manipulating service descriptions and protocols.
- **Analysis and management components** include various web services analysis and management tools. For instance, operators for protocol compatibility and replaceability analysis [BCT+06a],and protocol discovery from service execution logs [MNSPB+07].
- **The development components** provide a visual environment and a graphical editor for modifying, analyzing and managing model elements. We used this editor to visualize and to create and to handle target, component and composite services. This editor is built on GEF[3] (Graphical

[2]http://www.eclipse.org/
[3]http://www.eclipse.org/gef/

Editing Framework) that allows developers to take an existing application model and quickly create a rich graphical editor.

- **Models representation and storage components** provide Input/Output primitives to store and to physically represent (e.g. XML, relational DB) models on hard drives.

Our contribution within the settings of ServiceMosaic is the advent of the sub-project *WSC : Web Services Composer*. The main goal of WSC, as its name indicates, is to allow the automatic composition of web services protocols. These protocols can be designed by making use of the ServiceMosaic *development components* (editor). We also rely on the *models representation and storage components* for the physical representation and storage of our services.

The WSC project is based on two main functional components, namely :

- **Web services protocols generator** that enables to generate synthetic web service protocols according to several input parameters. Indeed, a user can generate a protocol by parameterizing its states number, transitions number, final states number, messages number (the alphabet size) and optionally the hybrid states number. If the last parameter is not specified, it will range randomly over 0 and the number of final states. It should be noted that this component allows to generate protocols with states that are all accessible and useful, i.e. there is at least a path leading from the initial state to each state and from each intermediate state there exists at least a path leading to a final state.

- **Web services protocols composer** that provides an implementation of our composition algorithm.

7.2 Performance evaluation

7.2.1 Evaluation goals.

We can observe that the time complexity of our composition algorithm depends on the size of the execution tree of the algorithm Check-sim. The sizes of such a tree vary depending on two main parameters: the degrees of the nodes (i.e., the number of childrens of a given node) and the depth of the tree (i.e. the sizes of the paths between the root and the leaves).

To better understand this issue, we focused our first experiments on the analysis of the impact of the following parameters on the execution time of the algorithm:

- Number of services in the service repository, noted #S,
- Total number of distinct message labels that appear in the services of the repository, noted #M,
- Number of hybrid states in each service in the repository, noted #H,
- Level of nested loop in each service in the repository, noted #L.

Indeed, the degree of a node depends on the number of candidates computed by the procedure Check-Candidate. It corresponds to the number of transitions labeled by the same message in the services of the repository. Note that the degree does not depend on the number of active instances of each service, since using any of them leads to the same configuration. To increase the node degree one can either increase the number of services (i.e., the value of #S) or decrease the number of message labels (i.e., the value of #M).

Secondly, The depth of the tree depends on the presence of loops. Indeed, our proof was based on the Dickson lemma [Dic13] which ensures the finiteness of the Check-sim procedure when hybrid states are present. This motivates

the use of the parameters #H and #L in our tests. The case where only intermediate states are considered, the depth of the execution tree is exponential. As will be seen below, this theoretical deduction was confirmed by the results of our experimental tests.

Test ID	**#S**	**#M**	**#H**	**#L**	Number of variants	Total number of generated tests
Test 1	200	2, 4, 8 .. 4096	c	c	12	12000
Test 2	10, 25, 50, 100, 200	2, 4, 8 .. 4096	c	c	60	60000
Test 3	100	10	0,1 .. 4	c	5	5000
Test 4	100	10	c	0, 1, 2, 3	4	4000

Table 7.1: Description of the test sets.

7.2.2 Building the test sets.

To achieve the aforementioned goals, we constructed 4 test sets each of which focusing on the study of some specific parameters among the ones mentioned above.

Each test set describes the main features of the studied composition problem. The description of the test sets, summarized at table 1, as well as the results of the experimental evaluation are presented in the remainder of this section. The experiments have been achieved on Xeon double process HT 3GHz and 2GO of RAM. In the presented results, the execution times are given in milliseconds.

7.2.3 Test 1.

This test set enables to assess the impact of the number of the distinct message labels that appear in the available services. For this test set (first line of the table 1, we defined a first variant with a target service and a repository of

200 available services taking their message from an alphabet of 4096 distinct labels.

Then starting with this first variant, we generate other variants by relabeling at each step the messages in order to reduce the total number of distinc labels by magnetitude of 2. The total number of variants is then equal to 12 (i.e., $\#M = 4096$, $\#M = 2048$, $\#M = 1024, \ldots, \#M = 2048$). Note that, the occurrence of symbol c in the table 1 indicates a constant value, generated randomly, and used for the different variants of the test set.

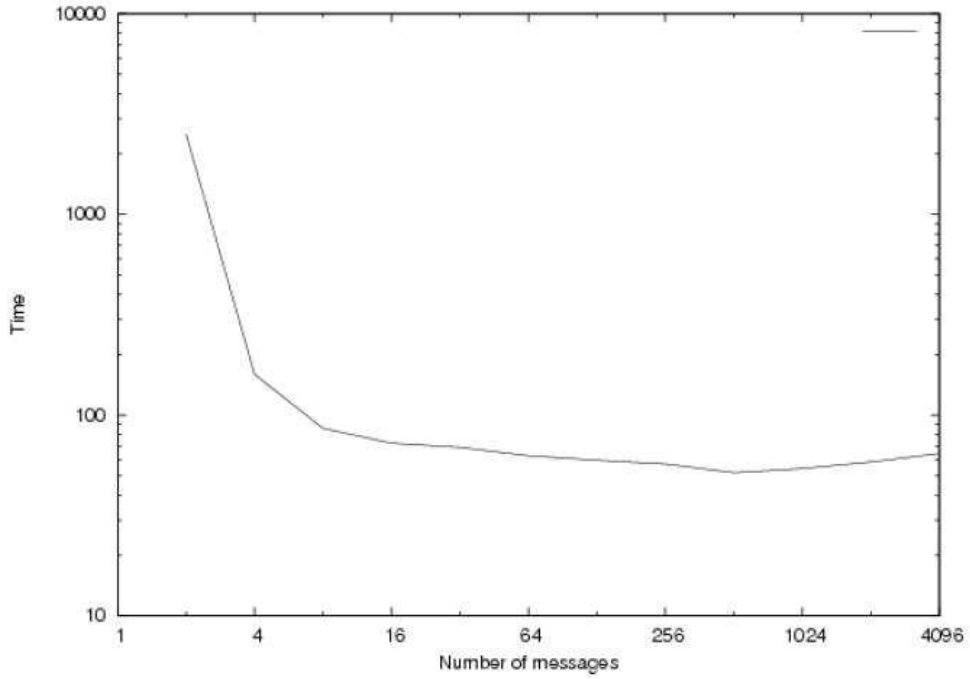

Figure 7.1: Experimental results of Test 1

For each of the variant of Test 1, we generated and runned 1000 instances. The result is reported on figure 7.1. Each point of the given curve denotes the average execution time of the 1000 instances of the corresponding variant. Observe that when $\#M$ decreases below a given threshold, namely 8 in the figure, this leads to an exponential blow up in the execution time while above

this threshold, the values of the parameter $\#M$ seem to have less impact on the performance of the algorithm.

7.2.4 Test 2.

In addition to the number of messages labels ($\#M$), this test set enables to assess the impact of the number of services available in the service repository ($\#S$). We considered five variants of this test set obtained by varying the value of the parameter $\#S$ (respectively, 10, 25, 50, 100 and 200). As previously, for each value of $\#S$, we define several variants for different values of $\#M$ (ranging from 4096 to 2). We generated and executed 1000 instances of each variant (i.e., a total number of 60000 tests). The average execution time of each variant is reported on figure 7.2.

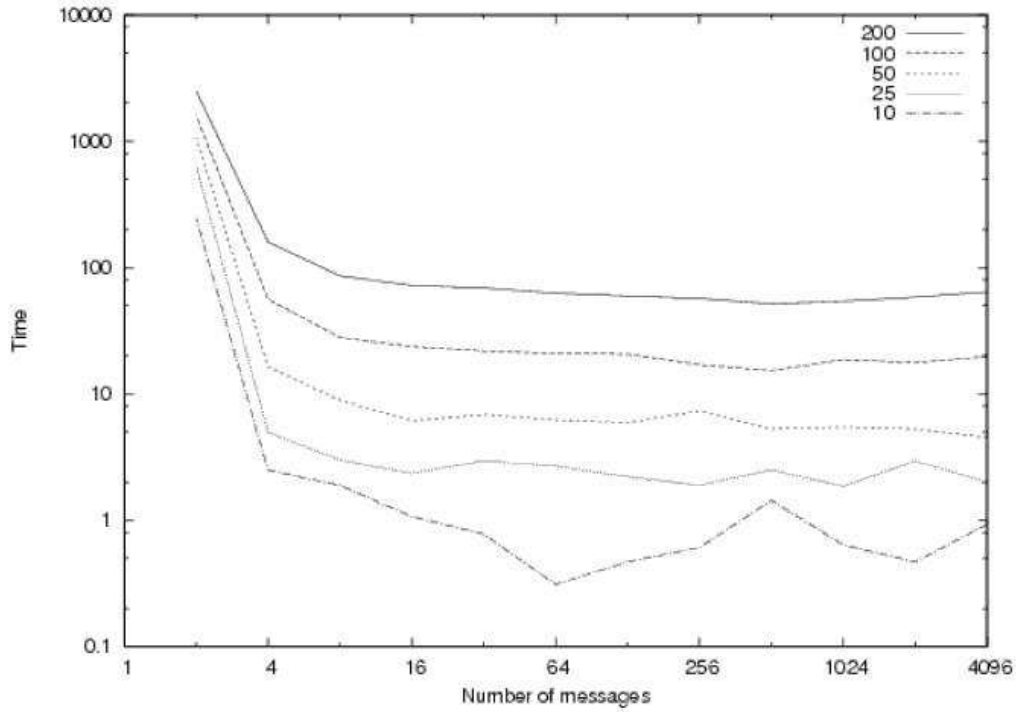

Figure 7.2: Experimental results of Test 2.

Unsurprisingly, the results show that number of available services to explore during the composition process impacts the global performance of the algorithm. Moreover, this test set confirms the trend observed previously regarding the impact of the number of the distinct message labels on the performance.

7.2.5 Test 3.

Test 3 studies the impact of the number $\#H$ of hybrid states (respectively, the level $\#L$ of nesting) in the available protocols. As previously, we generated a first variant of Test 3 with $\#S = 100$ and $\#M = 10$ and $\#H = 0$ (i.e., no hybrid state). Then, we generate other variants by modifying the first one by increasing the number of hybrid states (from 0 to 4). Therefore, we obtain a total number of 5 variants. We generate and executed 1000 instances of each variant. The results are depicted at figure 7.3.

Interestingly, we can observe two main phases in the results depicted on this figure. In the first phase (from $\#H = 0$ to $\#H = 1$) , the augmentation of the number of hybrid states leads to a proportional increase of the execution time while we observe the converse behaviour in the second phase (i.e., when $\#H > 1$, the execution time decreases while $\#H$ increases).

In fact, above a given threshold, adding hybrid states increases the number of accepting states making the complete conversation (i.e., accepted words) shorter.

7.2.6 Test 4.

Test 4 studies the impact of the level $\#L$ of nested loops in the available protocols. For this test set, we generate 4 variants with a fixed set of 100 services and 10 message labels. We distinguish 4 varaiants with respect to the values of $\#L$ (ranging from 0 to 3). We executed 1000 instances of each variant and reported the average execution time in figure 7.4. As it can be

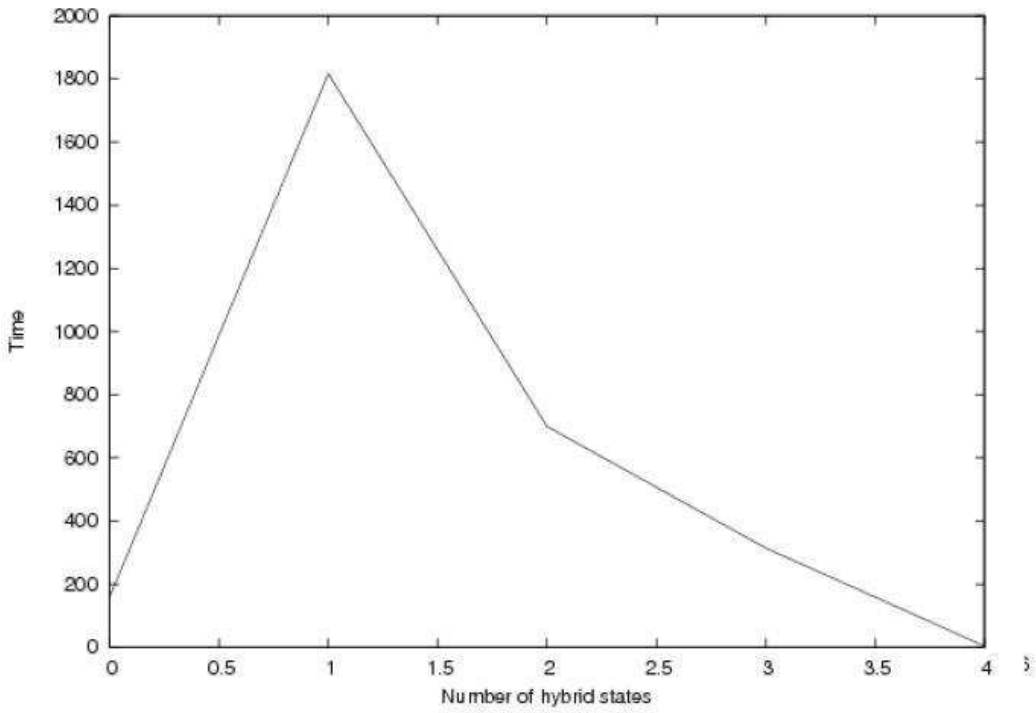

Figure 7.3: Experimental results of Test 3.

expected, it turned out that the level of nesting leads to an exponential blow up in the performance of the algorithm.

7.3 Discussion

Performance evaluation results showed the influence numbers of states, transitions and hybrid states, and the level of nested loops can have on the composition synthesis time. As expected, all these parameters plays an important role. In particular, the nesting level of loops and the hybrid state number can alter severely the composition time much more than the two other parameters. This evaluation allowed also to deduce the shape of 'easy instances' of the composition synthesis problem. Indeed, such instances are characterized

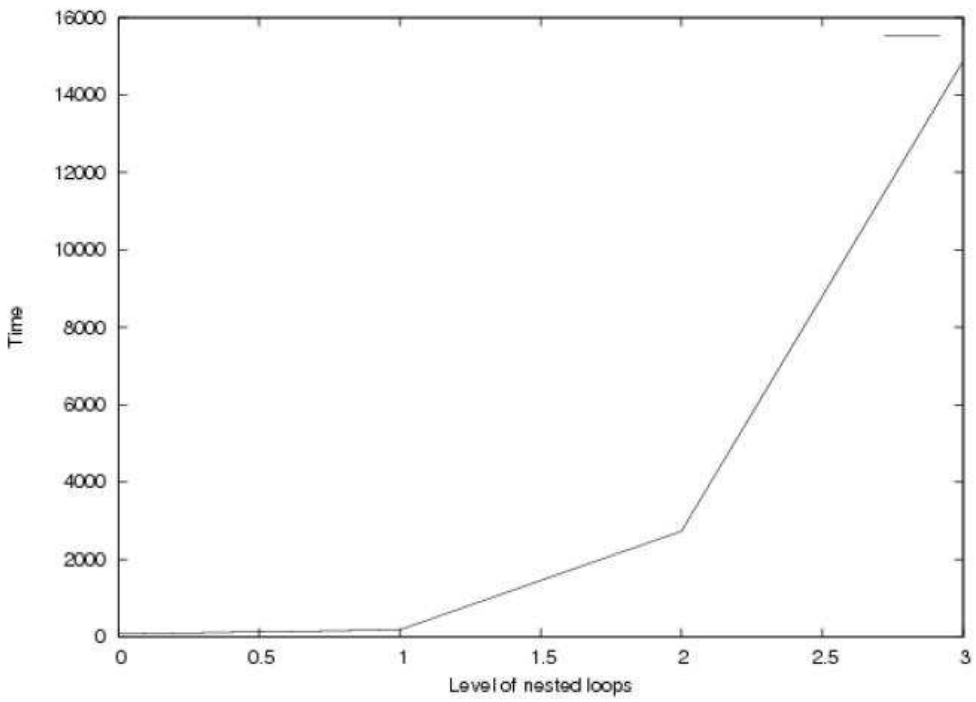

Figure 7.4: Experimental results of Test 4.

by a large alphabet, a medium number of states, a light hybrid states density and a low level of nesting loops. Whereas, the 'extremely hard instances' are characterized by a binary alphabet, large states and hybrid states number and finally, a high level of loops nesting in the target protocol (e.g. a target protocols with all possible transitions between each couple of states).

Eight

Conclusion

We conclude this work by summarizing the contributions and by drawing some perspectives for future work.

8.1 Summary

In this work, we investigated the composition synthesis problem. We provided a generic definition of protocol synthesis problem that holds for both the cases where the number of instances that can be used in composition is bounded or unbounded. The former case was widely investigated in literature (e.g. [BCG$^+$03, BDGL$^+$04, BCG$^+$05b]) and its exptime-completeness is known from [MW07]. Hence, we concentrated on the unbounded case of the composition that we formalized using the simulation preorder and the shuffle closure operator. More precisely, we proposed a new state machine, called PCSM, that allows to run an unbounded number of parallel instances of a finite state machine. Under settings of this framework, we proved the decidability of the underlaying problem and we study its complexity issues. We provided an Ackermannian upper bound and an exptime-hard lower bound. We dealt also with two particular cases of the generic composition synthesis problem, namely *(i)* the case where target services are without loops which is

NP-compete and *(ii)* the case where component services are without hybrid states which is exptime. Finally, a composition prototype and experimentations were detailed.

8.2 Perspectives

Several opportunities exist to pursue further research directions by building on top of the present work. Indeed, we can envision more theoretical refinements of our results as well as a richer composition model (yet useful in practice). We briefly detail those perspectives below.

1. **Complexity.** Right now, we have an Ackermannian upper bound that can not be improved as long as the decidability proof relies on the Dickson lemma. However, identification and complexity characterization of particular cases can be helpful to build a new decidability proof for the GCSP and hence a tighter upper bound can be derived. In addition, having such cases and their corresponding algorithms may allow to significantly increase the performances of a composition engine by running the adequate algorithm w.r.t each particular case.

2. **Towards a richer composition model.** One strong assumption of our approach is that the composition relies on a central delegator, which may turn out to be a limiting factor in practice (e.g., single point of failure or scalability issues). A first improvement would be to leverage choreographies by considering direct message exchanges between services which would get us rid of the delegator (e.g. the conversational model [BFHS03]). A second improvement would be to take into account operations effects on the real world, modeled as update queries on a relational database. Finally, we should investigate the extension of our work toward non-functional properties, including timing constraints

(e.g., time-bound message exchanges), transactions support, costs and security.

3. **Decidability problem.** Beyond the web services composition context, an interesting theoretical problem that remains to be solved is the decidability of simulating FSMs by Petri nets with final markings. We recall that the same simulation problem where the Petri nets have no final markings was shown to be decidable in [Jan95]. As discussed in Chapter 5 of this thesis, a technique to solve the former problem relies on the definition of a new preorder on markings (e.g. cover). This preorder will allow cutting branches of simulation trees within finite time and will ensure a simulation as well.

Bibliography

[AC98] P.A. Abdulla and K. Cerans. Simulation Is Decidable for One-Counter Nets (Extended Abstract). *Proceedings of CONCUR'98, Lecture Notes in Computer Science 1466: 253–268*, pages 26–8, 1998.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi A. Kuno, and Vijay Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, 2004.

[AGK99] F.N. Afrati, M. Gergatsoulis, and T. Kavalieros. Answering queries using materialized views with disjunctions. *Lecture Notes in Computer Science*, pages 435–452, 1999.

[AKT81] Toshiro Araki, Toyohiko Kagimasa, and Nobuki Tokura. Relations of flow languages to petri net languages. *Theoretical Computer Science*, 15(1):51–75, 1981.

[BCDG+05] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella. Automatic composition of web services in Colombo. In *Proc. of 13th Itallian Symp. on Advanced Database Systems*, 2005.

[BCG+03] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. Automatic composition

of e-services that export their behavior. In *ICSOC*, pages 43–58, Dec. 2003.

[BCG+05a] D. Berardi, D. Calvanese, GD Giacomo, R. Hull, M. Lenzerini, and M. Mecella. Modeling data & processes for service specifications in colombo. *Proc. EMOI-INTEROP. CEUR-WS. org*, 2005.

[BCG+05b] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Richard Hull, and Massimo Mecella. Automatic composition of transition-based semantic web services with messaging. In *VLDB*, pages 613–624, 2005.

[BCG+05c] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. Automatic service composition based on behavioral descriptions. *IJCIS*, 14(4):333–376, 2005.

[BCT04a] B. Benatallah, F. Casati, and F. Toumani. Web service conversation modeling: a cornerstone for e-business automation. *Internet Computing, IEEE*, 8(1):46–54, 2004.

[BCT04b] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Analysis and management of web service protocols. In *ER*, pages 524–541, 2004.

[BCT+06a] B. Benatallah, F. Casati, F. Toumani, J. Ponge, and H.R.M. Nezhad. Service mosaic: A model-driven framework for web services life-cycle management. *IEEE Internet Computing*, pages 55–63, 2006.

[BCT06b] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Representing, analysing and managing web service protocols. *DKE*, 58(3):327–357, 2006.

[BDGL+04] D. Berardi, G. De Giacomo, M. Lenzerini, M. Mecella, and D. Calvanese. Synthesis of underspecified composite e-services based on automated reasoning. *Proceedings of the 2nd international conference on Service oriented computing*, pages 105–114, 2004.

[BFHS03] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW'03*. ACM, 2003.

[BH05] L. Bing and C. Huaping. Web Service Composition and Analysis: A Petri-net Based Approach. In *Semantics, Knowledge and Grid, 2005. SKG'05. First International Conference on*, pages 111–111, 2005.

[Blo89] B. Bloom. *Ready simulation, bisimulation, and the semantics of CCS-like languages*. Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1989.

[BPE07] Business process execution language for web services version 2.0, 2007. http://docs.oasis-open.org/wsbpel/2.0/.

[CDGLV99] D. Calvanese, G. De Giacomo, M. Lenzerini, and M.Y. Vardi. Rewriting of regular expressions and regular path queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 194–204. ACM New York, NY, USA, 1999.

[CPS+91] R. Cleaveland, J. Parrow, B. Steffen, Laboratory for Foundations of Computer Science, and University of Edinburgh. *The concurrency workbench: A semantics based tool for the verification of concurrent systems*. Swedish Institute of Computer Science (SICS), 1991.

[CS93] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal methods in system design*, 2(2):121–147, 1993.

[Dic13] Leonard Eugene Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *Amer. Journal Math.*, 35:413–422, 1913.

[DS05] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[Eng67] E. Engeler. Algorithmic properties of structures. *Theory of Computing Systems*, 1(2):183–195, 1967.

[FGG+08] W. Fan, F. Geerts, W. Gelade, F. Neven, and A. Poggi. Complexity and composition of synthesized web services. In *Principals on Database Systems*, pages 231–240. ACM New York, NY, USA, 2008.

[FL79] M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *JCSS*, 18(2):194–211, 1979.

[FLW06] X. Feng, Q. Liu, and Z. Wang. A Web Service Composition Modeling and Evaluation Method Used Petri Net. *LECTURE NOTES IN COMPUTER SCIENCE*, 3842:905, 2006.

[FS04] K. Fujii and T. Suda. Dynamic service composition using semantic information. In *Proceedings of the 2nd international conference on Service oriented computing*, pages 39–48. ACM New York, NY, USA, 2004.

[Gal91] Jean H. Gallier. What's so special about kruskal's theorem and the ordinal γ_0? a survey of some results in proof theory. *Annals of Pure and Applied Logic*, 53:199–260, 1991.

[Gis81] Jay Gischer. Shuffle languages, Petri nets, and context-sensitive grammars. *Comm. of the ACM*, 24(9):597–605, 1981.

[GKOS08] O. Golubitsky, M. Kondratieva, A. Ovchinnikov, and A. Szanto. A Bound for Orders in Differential Nullstellensatz. *Arxiv preprint arXiv:0803.0160*, 2008.

[GR63] S. Ginsburg and G.F. Rose. Some Recursively Unsolvable Problems in ALGOL-Like Languages. *Journal of the ACM (JACM)*, 10(1):29–47, 1963.

[GT04] M. Ghallab and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[HAR67] J. HARTMANIS. Context-free languages and Turing machine computations. *Mathematical Aspects of Computer Science*, 1967.

[HB03] R. Hamadi and B. Benatallah. A Petri net-based model for web service composition. *Australasian Database Conference*, pages 191–200, 2003.

[Hfi94] H. Hfittel. Undecidable equivalences for basic parallel processes. volume 789. Springer, 1994.

[HHK95] Monika R. Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. pages 453–462. IEEE Computer Society Press, 1995.

[HMU01] J.E. Hopcroft, R. Motwani, and J.D. Ullman. Introduction to automata theory, languages, and computation. 2001.

[HNT08a] Ramy Ragab Hassen, Lhouari Nourine, and Farouk Toumani. Protocol-based web service composition. In *ICSOC*, pages 38–53, 2008.

[HNT08b] Ramy Ragab Hassen, Lhouari Nourine, and Farouk Toumani. Web services composition is decidable. In *WebDB*, 2008.

[HS05] R. Hull and J. Su. Tools for composite web services: a short overview. *ACM SIGMOD Record*, 34(2):86–95, 2005.

[HU69] J.E. Hopcroft and J.D. Ullman. Formal languages and their relation to automata. *ACM Classic Books Series*, 1969.

[Hul04] Richard Hull. Web services composition: A story of models, automata, and logics. A tutorial for the 2004 EDBT Summer School in Sardinia, 2004.

[Iwa82] Kazuo Iwama. The universe problem for unrestricted flow languages. *Acta Informatica*, 19(1):85–96, 1982.

[Jan95] P. Jančar. Undecidability of bisimilarity for Petri nets and some related problems. *Theoretical Computer Science*, 148(2):281–301, 1995.

[Jan03] P. Jancar. Strong bisimilarity on basic parallel processes in PSPACE-complete. In *Logic in Computer Science*, pages 218–227, 2003.

[Jed87] J. Jedrzejowicz. Nesting of shuffle closure is important. *Information Processing Letters*, 25(6):363–367, 1987.

[Jed99] Joanna Jedrzejowicz. Structural properties of shuffle automata. *Grammars*, 2(1):35–51, 1999.

[Jke96] Joanna Jkedrzejowicz. Undecidability results for shuffle languages. *Journal of Automata, Languages and Combinatorics*, 1(2):147–159, 1996.

[JM95] P. Jancar and F. Moller. Checking regular properties of petri nets. *ICCT*, pages 348–362, 1995.

[JS01] Joanna Jedrzejowicz and Andrzej Szepietowski. Shuffle languages are in P. *TCS*, 250(1-2):31–53, 2001.

[KJ06] A. KUCERA and P. JANCAR. Equivalence-checking on infinite-state systems: Techniques and results. *Theory and Practice of Logic Programming*, 6(03):227–264, 2006.

[KM69] R.M. Karp and R.E. Miller. Parallel Program Schemata. *JCSS*, 3(2):147–195, 1969.

[KM02a] A. Kucera and R. Mayr. Simulation preorder over simple process algebras. *Information and Computation*, 173(2):184–198, 2002.

[KM02b] A. Kucera and R. Mayr. Why is Simulation Harder Than Bisimulation? *Lecture Notes in Computer Science, CONCUR*, pages 594–610, 2002.

[KM02c] Antonín Kucera and Richard Mayr. On the complexity of semantic equivalences for pushdown automata and bpa. In *In Proceedings of the 27th International Symposium on Mathematical Foundations of Computer Science (MFCSŠ02), volume 2420 of LNCS*, pages 433–445. Springer-Verlag, 2002.

[KS05] M. Kot and Z. Sawa. Bisimulation equivalence of a BPP and a finite-state system can be decided in polynomial time. *Elec-*

tronic Notes in Theoretical Computer Science, 138(3):49–60, 2005.

[Lip76] R.J. Lipton. *The Reachability Problem Requires Exponential Space*. Dept. of Computer Science, Yale University, 1976.

[Mea55] G.H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[MJL07] Chan May, Bishop Judith, and Baresi Luciano. survey and comparaison of planning techniques for web services composition. *Technical Report.*, 2007.

[MNSPB+07] H. Motahari-Nezhad, R. Saint-Paul, B. Benatallah, F. Casati, J. Ponge, and F. Toumani. ServiceMosaic: Interactive analysis and manipulation of service conversations. *ICDE*, 2007.

[MS02] S. McIlraith and T.C. Son. Adapting Golog for Composition of Semantic Web Services. *KR'02*, pages 482–493, 2002.

[MSZ01] S.A. McIlraith, T.C. Son, and H. Zeng. Semantic Web Services. 2001.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[MW07] Anca Muscholl and Igor Walukiewicz. A lower bound on web services composition. In *FOSSACS*, volume 4423 of *LNCS*, pages 274–287. Springer, 2007.

[NM02] S. Narayanan and S.A. McIlraith. Simulation, verification and automated composition of web services. *WWW'02*, pages 77–88, 2002.

[ORR78] WF Ogden, WE Riddle, and WC Round. Complexity of expressions allowing concurrency. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 185–194. ACM New York, NY, USA, 1978.

[OWL04] OWL-S: Semantic markup for web services, 2004. http://www.w3.org/Submission/OWL-S/.

[PBB+04] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and monitoring web service composition. *Lecture Notes in Computer Science*, pages 106–115, 2004.

[Pet73] C.A. Petri. Concepts of net theory. In *Proceedings of MFCS*, volume 73, pages 137–146, 1973.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1981.

[PG03] M.P. Papazoglou and D. Georgakopoulos. Service-Oriented Computing. *Communications of the ACM*, 46(10):25–28, 2003.

[PMBT05] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated composition of web services by planning at the knowledge level. *International Joint Conference On Artificial Intelligence*, 19:1252, 2005.

[PTB05] M. Pistore, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. *Proc. ICAPS05*, 2005.

[RNT] R. Ragab, L. Nourine, and F. Toumani. Web services composition is decidable. http://www.isima.fr/ragab/RNTReport08.pdf.

[RV82] G. Rozenberg and R. Verraedt. Subset Languages of Petri Nets. *Informatik-Fachberichte; Vol. 66*, pages 250–263, 1982.

[SOA07] SOAP version 1.2, 2007. http://www.w3.org/TR/soap12-part1/.

[Soc91] G.M. Socías. An Ackermannian Polynomial Ideal. In *Proceedings of the 9th International Symposium, on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 269–280. Springer-Verlag London, UK, 1991.

[SPW+04] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN planning for Web Service composition using SHOP2. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):377–396, 2004.

[Sti98] C. Stirling. The joys of bisimulation. *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*, pages 142–151, 1998.

[Tho93] W. Thomas. On the Ehrenfeucht-Fraisse game in theoretical computer science. *Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 559–559, 1993.

[UDD01] UDDI executive white paper, 2001. http://uddi.org/pubs/UDDI_Executive_White_Paper.pdf.

[UH79] J.H.J. Ullman and JE Hopcroft. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley„ 1979.

[WH84] Manfred K. Warmuth and David Haussler. On the complexity of iterated shuffle. *JCSS*, 28(3):345–358, 1984.

[Wic76] BA Wichmann. Ackermann's function: A study in the efficiency of calling procedures. *BIT Numerical Mathematics*, 16(1):103–110, 1976.

[WSC] Web service conversation language (WSCL), 1.0. http://www.w3.org/TR/wscl10/.

[WSD07] Web services description language (wsdl) version 2.0, 2007. http://www.w3.org/TR/wsdl20/.

[YTX05] Y.P. Yang, Q.P. Tan, and Y. Xiao. Verifying web services composition based on hierarchical colored petri nets. In *Proceedings of the first international workshop on Interoperability of heterogeneous information systems*, pages 47–54. ACM New York, NY, USA, 2005.

[ZAAM03] R. Zhang, I.B. Arpinar, and B. Aleman-Meza. Automatic Composition of Semantic Web Services. *Proc. of the 2003 Int. Conf. on Web Services*, 2003.

Résumé

Les services web permettent l'intégration flexible et l'interopérabilité d'applications autonomes, hétérogènes et distribuées. Le développement de techniques et d'outils permettant la composition automatique de ces services en prenant en compte leurs comportements est une question cruciale.

Cette thèse s'intéresse au problème de la composition automatique de services web. Nous décrivons les services web par leurs protocoles métiers, formalisés sous la forme de machines d'état finies. Les principaux travaux autour de cette problématique se focalisent sur le cas particulier où le nombre d'instance de chaque service est fixé a priori. Nous abordons le cas général du problème de synthèse de protocoles où le nombre d'instances de chaque service disponible et pouvant intervenir lors de la composition n'est pas borné à priori. Plus précisément, nous considérons le problème suivant : *'étant donné un ensemble de n protocoles de services disponibles $P_1, \ldots, P_n$ et un nouveau protocole cible P_T, le comportement de P_T peut-il être synthétiser en combinant les comportements décrits par les protocoles disponibles?'*. Pour ce faire, nous proposons dans un premier temps un cadre formel de travail basé à la fois sur le test de simulation et la fermeture shuffle des machines d'états finis. Nous prouvons la décidabilité du problème en fournissant un algorithme de composition correct et complet. Ensuite, nous analysons la complexité du problème de la composition. Plus précisément, nous fournissons une borne supérieure et inférieure de complexité. Nous nous intéressons également aux cas particuliers de ce problème général. Enfin, nous implémentons un prototype de composition dans le cadre de la plateforme ServiceMosaic.

Mots clès Services web, Composition de services web, Automate shuffle, Simulation.

Abstract

Web services enable flexible integration and interoperability of autonomous, heterogeneous and distributed applications. A core challenge for the web services technology is the development of techniques and tools for automatically generating composite servicess by taking into account their behavioral properties (e.g. business protocols).

In this thesis, we focus on the problem of automatic composition of web services. We consider web services described by their business protocols which are formalized as finite states machines. Previous works on this problem dealt with the particular case where the number of instances of each component service is bounded and fixed a priori. We tackle the general case of the protocol synthesis problem where the number of instances of each component service that can be used in a composition is not bounded a priori. More precisely, we consider the following problem: *'given a set of n available web service protocols $P_1, \ldots, P_n$ and a new target protocol P_T, can the behavior described by P_T be synthesized by combining the behaviors described by the available protocols?'*. In order to cope with this problem, we first propose a formal framework for the composition synthesis based on both the simulation preorder and the shuffle closure of finite states machines. We prove its decidability through a sound and complete composition algorithm. Then, we conduct a complexity analysis of the composition problem. More precisely, we provide upper and lower bounds on the problem complexity. We also focus on several particular cases of this general problem. Finally, we implement a composition prototype within the framework of the ServiceMosaic platform.

Key words Web services, Web services composition, Shuffle automata, Simulation.