Simulation-based Techniques for Automated Service Composition

Fabio Patrizi
Simulation-based Techniques for Automated Service Composition
Author's address:

Fabio Patrizi
Dipartimento di Informatica e Sistemistica
Sapienza Università di Roma
Via Ariosto 25, I-00185 Roma, Italy
E-MAIL: patrizi@dis.uniroma1.it
WWW: http://www.dis.uniroma1.it/~patrizi
# Contents

## 1 Introduction
1.1 Service Composition: an Overview .................................. 2  
1.1.1 Describing Service Behavior ................................... 3  
1.1.2 Composing Services .............................................. 5  
1.1.3 Solution Techniques ............................................. 7  
1.2 Further Issues ...................................................... 11  
1.2.1 Dealing with Data ............................................... 11  
1.2.2 Distributed Orchestrators ...................................... 13  
1.3 Contributions ...................................................... 14  
1.4 Thesis Structure ................................................... 15

## 2 Related Work
2.1 Service Modeling .................................................. 17  
2.1.1 Literature Review ............................................... 18  
2.2 Service Composition ............................................... 23  
2.2.1 Literature Review ............................................... 24  
2.3 Related Research Areas ............................................ 30  
2.3.1 AI Planning & Reasoning and Multi-Agent Systems .......... 30  
2.3.2 System Verification & Synthesis ................................ 30  
2.3.3 Data Integration ............................................... 31  
2.4 Summary of Literature Results ................................... 32

## 3 The “Roman Model”
3.1 The Framework ................................................. 35  
3.2 Service Composition ............................................... 40  
3.3 Composition and Simulation ...................................... 44  
3.4 Synthesizing Compositions via Simulation ...................... 49

## 4 The Extended Roman Model
4.1 The Framework .................................................. 60  
4.2 Nondeterministic Service Composition .............................. 66
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>Composition Synthesis via ND-Simulation</td>
<td>70</td>
</tr>
<tr>
<td>4.4</td>
<td>Discussion</td>
<td>83</td>
</tr>
<tr>
<td>5</td>
<td>Composition Robustness</td>
<td>85</td>
</tr>
<tr>
<td>5.1</td>
<td>The Framework</td>
<td>86</td>
</tr>
<tr>
<td>5.2</td>
<td>Just-in-Time Orchestrators</td>
<td>87</td>
</tr>
<tr>
<td>5.3</td>
<td>Service Failures</td>
<td>91</td>
</tr>
<tr>
<td>5.4</td>
<td>Reactive Adaptability</td>
<td>92</td>
</tr>
<tr>
<td>5.5</td>
<td>Parsimonious Refinement</td>
<td>93</td>
</tr>
<tr>
<td>6</td>
<td>Services with Multiple Operations</td>
<td>103</td>
</tr>
<tr>
<td>6.1</td>
<td>The Framework</td>
<td>104</td>
</tr>
<tr>
<td>6.2</td>
<td>Composing Services with Multiple Operations</td>
<td>108</td>
</tr>
<tr>
<td>6.3</td>
<td>Composition Synthesis via MND-Simulation</td>
<td>113</td>
</tr>
<tr>
<td>7</td>
<td>Distributed Service Composition</td>
<td>121</td>
</tr>
<tr>
<td>7.1</td>
<td>The Framework</td>
<td>122</td>
</tr>
<tr>
<td>7.2</td>
<td>Distributed Orchestrators</td>
<td>125</td>
</tr>
<tr>
<td>7.3</td>
<td>Local Orchestrator Synthesis</td>
<td>129</td>
</tr>
<tr>
<td>7.4</td>
<td>Discussion</td>
<td>135</td>
</tr>
<tr>
<td>8</td>
<td>Composition and Safety Games</td>
<td>137</td>
</tr>
<tr>
<td>8.1</td>
<td>Simulation and Safety Games</td>
<td>137</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Safety-Game structures</td>
<td>138</td>
</tr>
<tr>
<td>8.1.2</td>
<td>From Composition Problem to Safety Games</td>
<td>145</td>
</tr>
<tr>
<td>8.2</td>
<td>Using \textsf{TILV} for computing compositions</td>
<td>152</td>
</tr>
<tr>
<td>8.2.1</td>
<td>The \textsf{SMV} encoding</td>
<td>153</td>
</tr>
<tr>
<td>9</td>
<td>Implementation</td>
<td>169</td>
</tr>
<tr>
<td>9.1</td>
<td>General Architecture and Typical Scenario</td>
<td>169</td>
</tr>
<tr>
<td>9.2</td>
<td>WSCE</td>
<td>170</td>
</tr>
<tr>
<td>9.2.1</td>
<td>System Overview</td>
<td>171</td>
</tr>
<tr>
<td>9.2.2</td>
<td>Service Representation</td>
<td>171</td>
</tr>
<tr>
<td>9.2.3</td>
<td>System Details</td>
<td>174</td>
</tr>
<tr>
<td>9.3</td>
<td>Current Extensions</td>
<td>177</td>
</tr>
<tr>
<td>10</td>
<td>Conclusions and Future Work</td>
<td>179</td>
</tr>
<tr>
<td>10.1</td>
<td>Conclusions</td>
<td>179</td>
</tr>
<tr>
<td>10.2</td>
<td>Future Work</td>
<td>181</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Services are software artifacts, possibly distributed and built on top of different technologies, that export a description of themselves, are accessible to external clients and communicate through a commonly known, standard, interface which enables inter-operability. More in general, Service Oriented Computing (SOC, cf., e.g., [4]) is a computing paradigm whose basic elements are services, that can be used as building blocks to devise other services. A classical example of such paradigm is provided by Web Services, i.e., applications published over the Internet and self-described, usually built by different companies and relying on different technologies, which share a same communication protocol \(^1\). For instance, online travel agencies such as Expedia \(^2\) integrate different web services offered by hotels, airlines, restaurants, etc., and provide final users with a complete service, combining all functionalities of its basic components. To do so, no constraint is required over the *internal* structure of each web service, whereas they are all required to be published, compliant with the same communication protocol and to export a description of their interface, so to facilitate clients’ access and communication.

Abstracting from this example, services can be thought of as generic programs, publicly available and *wrapped* so to mutually interact and communicate over a common platform (often referred to as *middleware*). As such, the SOC paradigm makes easier code re-use and extension, as, in a sense, each service is interpreted as a usual *method* in programming languages and, thus, a set of services, usually referred to as a *community*, as a sort of *programming library*. This similarity can be taken as the basis of *service composition*: as exactly as in a programming language methods are combined to produce more complex methods, so services can be combined to build more complex services.

\(^1\)SOAP built on top of HTTP.

\(^2\)www.expedia.com
This thesis focuses on automated service composition, that is, the problem of automatically combining a set of available services, so to meet a desired specification. Such a topic has a lot in common with other research areas, the most closely related being, probably, System Verification & Synthesis (e.g., [123]), but also others provide theoretical frameworks that service composition can be cast over, such as Planning in Artificial Intelligence (e.g., [120]), Reasoning About Actions (e.g., [103]) and even Data Integration (e.g., [140]), thus making available some research achievements that SOC research can benefit from. Due to this interdisciplinarity, a variety of approaches have been proposed to model services, e.g., as atomic actions [103], finite state machines [42, 25, 24] or views over data [142], and to solve the corresponding composition problem.

1.1 Service Composition: an Overview

We take a closer look at service composition, starting from the classical architecture for Web Services. Although based on such a specific (service oriented) architecture, our discussion is general enough to highlight the central issues arising when services are composed.

In Figure 1.1, the parties involved in a web service-based session are depicted. In addition to a client, which can be a service itself, two entities are shown, namely: a UDDI directory and a set of Service Providers. The former is a central, well-known, registry which stores services’ descriptions and is accessed by clients searching for services that meet desired requirements; the latter are service providers, such as a companies, which provide actual running services that clients actually interact with. More in general, UDDI (Universal Description Discovery and Integration) directories are known as service brokers, as their main purpose is publishing providers’ services over the Internet (or a generic network) and providing search functionalities over them. On their side, providers register their services to brokers and, when invoked, serve clients’ requests. A typical session is as follows: (i) a client that needs some service contacts one (or more) broker(s) and searches for a service that meets some specification, e.g., a weather forecasting service; (ii) if such a service is found, the broker redirects the client to the service provider actually deploying the service; (iii) the client, which located the service based on broker’s information, contacts the desired service and interacts with it.

This example raises two classical questions about SOC:

1. How are services described?
2. What if the desired service is not found?

The first question concerns service modeling, i.e., the definition of a suitable abstraction of services, able to capture aspects that can be relevant to client;
the second one raises the problem of finding a constructive alternative to the trivial answer: “client’s request cannot be fulfilled, unless someone develops and deploys a new service that meets the desired requirements”. As one may expect, rather than a correct answer to each question, there exist many reasonable ones. This work addresses both problems. On the one hand, we propose a description model, that substantially enriches existing ones, able to capture services’ behavior, that is, which provides an abstraction of services’ evolution over time, representing their possible conversations with clients; on the other hand, on top of this model, we propose sound and complete techniques for building a solution that fulfills client’s needs, when possible, by combining other available services. In particular, we show that such techniques correspond to the best one can do, in that they return the most general solutions, while being optimal with respect to worst-case time complexity.

1.1.1 Describing Service Behavior

In the literature, several approaches to service modeling have been proposed. Rather than actual languages, such as WSDL ([49, 48]), widely used to describe Web services, we focus on their conceptual model. For example, a WSDL (Web) description exports a functional specification of a service, that is, from an abstract standpoint, the set of operations provided by the service, along with the corresponding format of messages exchanged. We can say that WSDL has an underlying atomic conceptual model, specified in terms of input-output requirements. For instance, a service providing stock quotes of some market can be successfully described this way, with a single operation that returns the list of quotes. Such a model is useful in many situations, as its popularity over the Web witnesses. Indeed, WSDL is a current standard for Web service description. However, when more complex specifications need to be exported, it shows severe limitations. For instance, think of the same web
service for stock quotes as above and assume that it provides quotations only to authenticated clients. In an input-output approach, one would describe two operations, say, \texttt{auth} and \texttt{quote}, as well as the respective data format necessary for interaction. Unfortunately, the input-output approach does not allow for conversation specification, i.e., for putting constraints on the order that operations should be executed in. A very natural constraint would be, e.g., requiring clients to authenticate before requesting quotes. Observe also that cases may exist where two services export a same set of operations but allow different execution sequences. Since this last constraint is not captured by input-output approaches, such services would appear to clients as the same.

In a word, atomic conceptual models export services’ interface but not their behavior.

Our Approach The need for a behavioral description of services is widely recognized (cf. e.g., [21, 4, 83, 25]), yet, the community suffers from a lack of standard languages for this purpose. In this work, we follow the same approach as the so-called Roman Model ([82]), which was originally introduced in [25, 24]. Inspired by that, we propose a rich model, oriented to describe all conversations supported by services, that includes relevant features, such as nondeterminism and shared memory, and, thus, increases the set of actual scenarios that are captured.

In our model, services export their behavioral specifications, by means of a language that represents transition systems, i.e., Kripke structures whose transitions are labeled by service’s operations, under the assumption that each legal run of the system corresponds to a conversation supported by the service. To clarify this, consider Figures 1.2(a) and 1.2(b). The former is a graphical representation of an input-output description of the stock quote service, with authentication, described above, which provides information about operations that can be requested; the latter is a behavioral representation of the same service, providing more information: indeed, it tells clients that they \(i\) must authenticate before requesting a \texttt{quote} operation and, then, \(ii\) may request any number of quotes. Of course, more sophisticated examples do exist, where several operations, even nondeterministic, can be executed in a state, with nondeterminism modeling partial knowledge about service’s internal logic. Also, there are settings relying on the same approach, where operations have parameters and are able to exchange data with other clients and even with an underlying database (cf. e.g., [23]).

A first advantage brought by such a model is its generality with respect to service integration, in the sense that it is abstract enough to serve as conceptual model for several classes of scenarios. As an example, it can be used to model web service applications as well as multi-agent system ones. As a
1.1. SERVICE COMPOSITION: AN OVERVIEW

Figure 1.2: Two approaches to service description: 1.2(a) Input-output model; 1.2(b) Behavioral model.

consequence, results obtained on this model are also relevant to areas different from SOC. Second, from the SOC viewpoint, it provides a behavioral, stateful, service representation, which allows for describing those inter-operation (temporal) constraints that current languages, e.g., WSDL, do not capture. We remark the importance of such feature in a perspective of composition automatization: indeed, composition engines are intended to replace human operators, who compose services based on their informal description, often provided in natural language, which include behavioral information.

Importantly, when dealing with a behavioral model, we can look at services as high-level descriptions of software artifacts. Indeed, they are characterized by states and state transitions triggered by inputs, which, specifically, represent requested operations. This interpretation suggests, hence, to see service (possibly finite) runs as computation fragments, that can be suitably combined to generate more complex services.

1.1.2 Composing Services

Many works exist which deal with automated composition of conversational services, where service behavior is abstracted by various kinds of transition systems, e.g., [22, 42, 146] (see Chapter 2 for a related work). The closest one to this thesis is [22], that we take as a starting point, where the problem of automatically composing a set of services, described as possibly nondeterministic transition systems, is addressed. Although here we propose a significant extension of [22], in that we devise a novel solution technique which relies on effective technologies and yields great advantages, the basic problem has not changed. It can be informally stated as follows:

Consider a set of available services, a.k.a. community, and an additional target service, all exporting their conversational behavior. Is it possible to coordinate the available services so to support, at execution time, all conversations supported by the target service?
In other words, the problem amounts to realize a (virtual) target service, by resorting only to (actual) available services. Obviously, how services are combined in the practice depends on the exported behavioral model. To see how this can be done under our model, consider the following example.

**Example 1.1.1** Figure 1.3 shows a service composition problem instance in the Roman Model, which includes two available services, represented in Subfigures 1.3(a) and 1.3(b), and a target one, in Subfigure 1.3(c). The one in Subfigure 1.3(a), say $S_a$, provides login/logout capabilities, allowing a client to be authenticated and to close an authenticated session, whereas the one in Subfigure 1.3(b), say $S_b$, provides market stock quotes from all over the world. Clients willing to interact with $S_b$ are, first, required to input the market country of their interest and, then, are allowed to request either stock quotes or currency rates (versus, e.g., euro and dollar) for that market. As for the target service, say $S_c$, it provides stock quotes of a selected market only to authenticated clients. Specifically, clients of such service need first to login, then to select a market country, then are allowed to request quotes and, finally, to logout.

As we said, target services are virtual, that is, only their specification exists, whereas their implementation is missing. However, it is easily seen that, by resorting to available services, this example’s target service can be built. Indeed, it is enough delegating login/logout operations to $S_a$ and country selection and stock requests to $S_b$. Observe that the target service not only provides a set of operations, but imposes a set of constraints over their executions, e.g., stock can be requested only after country has been executed. Since, on their side, also available service operations are subject to such kind of constraints, when a target service is to be realized, they must be met. For instance, had not $S_c$ required operation country be executed before stock, it would be not
1.1. SERVICE COMPOSITION: AN OVERVIEW

realizable, as $S_b$ is the only service that provides stock and it requires country to be executed first.

In Example 1.1.1, the composition can be realized by a machine which, on one side, receives client’s operation requests and, on the other side, forwards them to an appropriate available service which executes it and, consequently, changes its state, where a new set of operations becomes available. Such a machine, similar to a Mealy machine but that can be, in general, infinite-state, is called orchestrator. One that solves the problem of Example 1.1.1 is shown in Figure 1.4. Each state of the machine corresponds to a state of the target service and each transition is labeled by a pair of the form operation/service, with an intuitive semantics: the requested operation is assigned to the output service. For instance, operation login is delegated to service $S_a$.

The example above shows how the existence of temporal constraints among operation executions makes the problem non-trivial: each time an operation is to be delegated to some available services, one needs to check whether all constraints are fulfilled, i.e., whether the service chosen for delegation is in a state where the operation is actually executable. This makes the orchestrator construction a hard task, akin to an advanced form of conditional planning [72]. Indeed, in the Roman Model, the service composition problem is shown to be EXPTIME-complete [24, 108].

More complex scenarios can be considered. For instance, nondeterministic available services are also conceivable, where nondeterminism over operation execution represents partial knowledge about service’s internal logic. Also, one could think of services communicating through a common blackboard or even exchanging data. All these scenarios require different notions of composition and, hence, different kind of orchestrators. The aim of this thesis is to provide a formal model for them and to propose a respective solution technique for the resulting composition problem.

### 1.1.3 Solution Techniques

Once the service composition problem and its solution have been defined, the problem of finding solution techniques, that can be automatized, becomes
PDL-based solutions

Previous work [22] addresses the problem of building a single orchestrator that is a solution to the problem. In such work, a technique was developed, able to deal with nondeterministic, finite-state, services, based on an encoding of the problem as a Propositional Dynamic Logic (PDL) [64] formula. Although such technique is only able to build finite-state orchestrators, it is actually made effective by a crucial result showing that if an orchestrator exists then there exists one which is finite [22].

In a nutshell, PDLs constitute a family of logics that allow for specifying evolution of propositional properties over time, in response to events. PDL models are Kripke structures, as often happens when dealing with dynamic systems. Importantly, for each PDL logic there exists an equivalent Description Logic (DL) –i.e., a logic used for static knowledge representation, expressed in terms of classes and relationships among them [12]– such that each model of the former is a model of the latter and vice versa [132, 53]. So, since PDLs capture the Roman Model and being DL’s effective reasoning tools available (e.g., FACT [81], RACER [75], Pellet 3), one can exploit DLs to represent and actually solve composition problems.

A conceptual schema of the PDL-based approach adopted in [22] is depicted in Figure 1.5. Starting from a description of the problem, where all involved services are described by transition systems, first, an abstract PDL formula is generated such that (i) each of its models (which are finite-state) corresponds to an (finite-state) orchestrator that is a solution to the original problem and, vice versa, (ii) each composition problem’s (finite-state) solution has a corresponding model of the PDL-formula. Then, such formula is translated into a DL knowledge base, represented in an actual format suitable for a DL reasoner which, finally, generates a model of the knowledge base.

3http://clarkparsia.com/pellet

Figure 1.5: Conceptual schema of PDL-based approach to service composition
if consistent. By construction, such a model is also a model of the original PDL formula which, in turn, corresponds to a composition problem’s solution. Based on this approach, an actual tool, ESC (E-Service Composer), has been devised ([22]) which is able, with some limitations, to actually compute an orchestrator that realizes a target service. Unfortunately, this approach has three major drawbacks:

- only finite-state orchestrators are returned;
- the obtained solution is not flexible, that is, if a solution has been built which relies on an available service and such service becomes unavailable at runtime, then the solution is no longer valid and the best one can do, with this approach, is to re-compute a new solution;
- on the practical side, due to implemented DL reasoners’ limitations, ESC is actually able to synthesize a model only for some particular inputs, though it is complete with respect to checking for the existence of a model.

These limitations constitute the essential motivations behind this thesis.

**Simulation-based solutions**

We propose a novel solution technique based on the formal notion of simulation relation between transition systems [106]. Informally, given two transition systems, $S_1$ and $S_2$ we say that $S_1$ simulates $S_2$, if it shows, at least, the same behaviors as $S_2$. For example, considering Figure 1.6, assume that $S_1$ is the transition system shown in Subfigure 1.6(b) and $S_2$ is the one in Subfigure 1.6(a). Seen as services, $S_1$ simulates $S_2$ as each conversation supported by $S_2$, i.e., runs of the form $(ab)^*$, is also supported by $S_1$. Of course, the viceversa does not hold: for instance, $(b)^*$ is supported by $S_1$ but not by $S_2$.

Essentially, our technique amounts to reduce the composition problem to the search for a simulation relation between the target service and the available services’ asynchronous product, which is itself a transition system. Precisely, it is the transition system which describes all the possible interleaved executions of available services or, in other words, represents all potentialities of the community services, seen as a whole. Recalling above informal definition of simulation relation, this corresponds to answer the question: “can available

---

4Although, here, we resort to regular languages as a mean for describing service conversations, in our approach we do not adopt FSMs as service models. Indeed, generic transition systems are better suited for our purposes, as we are interested in which choices a service actually provides in each state. This issue will be clarified in Chapter 3.
services be combined so to include the same behaviors as target service?”}, which is precisely our problem.

A fundamental advantage brought by our technique is that simulation-based solutions are, in fact, universal solutions, or composition generators, i.e., finite structures that represent all possible, even infinite-state, orchestrators that realize a target service. Importantly, this does not affect worst-case time complexity: indeed, it is known that searching for a simulation between two transition systems is polynomial in the size of the systems, hence, since an asynchronous product result has exponential size with respect to its factors’ size, it comes out that our technique requires exponential time with respect to the size of (original) input systems. This, along with the observation that the service composition problem is proven EXPTIME-complete, yields that also our simulation-based technique is optimal with respect to worst-case time complexity. In fact, with respect to the PDL-based approach, we obtain a complexity characterization refinement.

But our approach yields two additional benefits.

First, by using composition generators, we obtain flexible solutions, that is, able to change their behavior, based on information available at runtime. The PDL-based approach does not provide this feature, since it returns a single, rigid, solution that cannot be modified at runtime. For example, if, during execution, an available service that the executing orchestrator relies on becomes unavailable, but can be replaced by another one, then, with our solution, we can change, without need for recomputation, the available service used to realize the target service. Differently put, we are able to switch orchestrators during execution.

Second, a set of effective tools, such as T1V [124], Lily [84] or Anzu [85], for computing simulation becomes available. Precisely, such tools are aimed at synthesizing finite-state dynamic systems that meet desired temporal properties. In particular, they can be used to compute winning strategies for safety
1.2. Further Issues

1.2.1 Dealing with Data

Data-Management capability is a desirable service property to capture: as a matter of fact, real services deal with data and including such feature in service representation would result in a more complete model, thus, yielding the possibility of facing even more realistic problems. However, as several works witness, e.g., [23, 56, 55], the presence of data poses a major obstacle to service composition and verification: data are infinite by their nature and make services infinite-state. Unfortunately, the techniques described above, as well as other proposed in the literature (cf., e.g., Chapter 2), are effective only when dealing with finite-state systems, whereas infinite-state system verification (and synthesis) is known as a hard task for most non-trivial properties, and undecidable for general ones. Consequently, introducing data in service composition frameworks has a great impact on the solution techniques.
that can be adopted, thus making the problem a major challenge for the SOC community.

In this work, we propose a model that allows for dynamic and finite-state data structures’ representation. Precisely, in addition to available services, a community contains a so-called finite-state data box, i.e., an additional transition system modeling the evolution of a shared data structure whose state is affected by available services’ execution, and that can be used to realize a basic form of inter-service communication. From a modeling perspective, a data box can be seen as a database with a finite-state behavior, used to capture some situations where services deal with data over finite domains. This way, we keep dealing with finite-state systems, thus making simulation-based techniques still applicable, but introduce a simple form of data-awareness.

Such extension, besides increasing the set of scenarios that can be captured and, hence, making the model appealing also for other areas, provides the bases for a more ambitious research, i.e., devising techniques for solving composition problem instances over generic data-aware services. The basic idea is abstracting over actual data, by building symbolic representations of actual database instances using only a finite set of symbols and, hence, describing the infinite-state actual evolution of available services and database as a finite-state symbolic one. This way, one can reduce the original problem to one defined over finite-state symbolic structures, thus making above techniques still applicable. One attempt towards this direction, in a synthesis context, is the Colombo framework described in [23], where services are able to exchange data from infinite domains and to interact with a relational database, by reading/writing scalar values. As for service verification, [56, 55] propose two similar abstraction techniques allowing for data-aware services’ verification.

We observe that [23] tackles the composition problem by relying on a PDL-based approach. However, under the same model, one can recast the problem in terms of (data-aware) simulation, that is, by defining a relation between two data-aware services that interact with a common underlying data structure, whose data content may come from an infinite domain. This way, one would get the advantages brought by a simulation-based approach, though the actual resolution would be more complex, due to state space infiniteness, which calls for some abstraction procedure.

\(^5\)Such a feature was first introduced in [54], in a different context.

\(^6\)Cf., e.g., [129, 52], where multi-agent scenarios have been formulated as service composition ones—
1.2. FURTHER ISSUES

1.2.2 Distributed Orchestrators

In SOC, centralized scenarios, though conceivable, are rare. Even when all community services actually reside in a same repository, they communicate through a common middleware that enables communication and interoperability. Hence, the resulting abstraction is still a distributed community, where services are seen as programs, located at different places, that interoperate through some protocol. In such distributed context, there can be cases where a central coordinating entity, such as an orchestrator, is not realizable nor convenient. In [130, 52], two examples of such scenarios are reported. In particular, [52] shows one where all services execute on distributed peer devices, communicating via a wireless network. Clearly, in such situations one cannot rely on a single, central, coordinating unit: indeed, network disconnections are possible and can dramatically impact the overall system effectiveness. If, for instance, the coordinator permanently looses connection, peers are no longer able to cooperate. Also less catastrophic events, such as temporary disconnections, may affect system's efficiency and/or effectiveness.

In this thesis, we build a peer-based framework for distributed orchestrators' execution, that is, for execution in a scenario where each service has an attached local orchestrator, able to communicate with other ones to cooperate in achieving a common goal.

As discussed above, orchestrators make their choices, i.e., select available services for delegation, based on community current state, provided they have full observability on available services’ state. However, in this case, such assumption is no longer valid as, on the one hand, services reside at different places and, on the other hand, more importantly, there is no central entity able to observe their state. In fact, local orchestrator have full observability only on the service they are attached to, whereas know nothing about other services. Therefore, even if a technique for distributed execution of a centralized orchestrator were given, since, in general, services are nondeterministic, a problem of state reconstruction would arise. To solve this, we rely on services’ communication ability, which is exploited to broadcast messages to other peers. Essentially, our technique amounts to building a usual orchestrator, as if it were to be executed on a central coordinating device, and then, from this, generating one local orchestrator for each available (distributed) service. Local orchestrators replicate centralized one’s behavior, except for two features: (i) they are able to send/receive messages that are useful for community state reconstruction –i.e., they make their decisions based on received messages and current state of the service they are attached to– and (ii) they issue commands only to the service they are attached to, while using received messages to keep track of current community state. Differently put, each local orchestrator, through received messages, reconstructs community current state, and
history, until it is its turn, and only then activates the respective service by issuing the command requested by the client.

This technique, along with the possibility of executing services in parallel, and not only under an interleaving policy, contributes to make the service-based approach quite general and, thus, appealing also for other research areas than SOC, like, e.g., Multi Agent Systems (MAS).

1.3 Contributions

The main goal of this work is to address the drawbacks associated with a PDL-based solution approach to the service composition problem. In doing this, the achieved results introduce a set of novelties with respect to the service composition state-of-the-art (described in Chapter 2):

- we propose a rich conceptual model for service behavioral description, that makes services abstract enough to represent a variety of dynamic, possibly communicating, entities, such as web services, physical devices or agents.

- We devise a novel approach to service composition that exploits the formal notion of simulation, thus bringing the following major advantages:
  
  - a huge literature about simulation, and simulation-related problems, is available, as it is a standard notion, widely and very well studied in Computer Science and related areas, thus providing access to a set of useful results, such as efficient procedures for computing simulation relations (e.g., [68]);
  
  - the notion of simulation relation is associated with System Verification & Synthesis, thus making effective tools, based on Model Checking approach, available. In fact, we resort to one of such tools to build our composition engine.

- the proposed techniques overcome the obstacles met by the PDL-based approach:
  
  - they return the whole (in general, infinite) set of solutions (including infinite-state ones), represented as a finite orchestrator generator, without requiring additional computational efforts, with respect to previous approaches. This is undoubtedly the most relevant achievement, which shows the optimality of the approach.
  
  - Just-in-time solutions, able to adapt to run-time exceptional situations, can be built. With respect to the PDL-based approach, this corresponds to the possibility of switching orchestrators during
target service realization. Importantly, one can take advantage of information available at runtime, such as unexpected faults, which were considered *catastrophic* in previous approach;

– we get a refinement of the complexity bound obtained in [22]. Precisely, we identify the exponential term as the number of available services, rather than their size.

• We provide actual techniques that take advantage of flexible solutions to efficiently recompute an orchestrator generator, when some exceptional situations, such as temporal/permanent unavailability of available services, arise;

• we propose a formulation of the composition problem in a distributed scenario and show how it can be solved. The problem is not introduced to face a *pure* service composition scenario, though it might be conceivable, but, rather, finds natural application in the Multi Agent System field, where several agents are described as possibly nondeterministic, communicating, finite-state transition systems (i.e., the same as services), operating in a common environment and communicating through a distributed shared memory;

• we show how the problem can be encoded into a safety-game structure, then written in an actual language, SMV, and, finally, processed by an implemented system, Tlv, to compute the whole set of solutions;

• based on above game structure encoding, an actual system, Web Service Composition Engine (WSCE) is devised, which exploits Tlv for efficient problem solution;

All achievements have been published in international journals ([29, 44]), proceedings of international conferences ([130, 52, 129]) or international workshops ([36]).

1.4 Thesis Structure

The rest of this thesis is organized as follows:

• in Chapter 2, we review literature’s achievements that our work is based upon;

• Chapters 3 and 4 introduce our model, starting from the basic Roman Model, as described in [22]. In addition, the simulation-based solution technique is developed and related results are proven.
• Chapter 5 shows how simulation-based solutions can be actually exploited to build robust solutions that react to unexpected situations. We consider several kinds of faults that available services may experience, such as temporal or permanent unavailability, but also take into account the possibility of unexpected service resuming.

• In Chapter 6, a further model extension is proposed which provides the possibility of executing several operations per step (rather than one only), and

• in Chapter 7, such model is adopted to formalize and solve the composition problem in a distributed context.

• Chapter 8 shows how the service composition problem can be reduced to a safety game, and provides a methodology to translate abstract problem instances into actual ones, in the SMV language, that can be resolved by the well-known synthesis tool TgV.

• Chapter 9 describes an implemented system, WSCE (Web Service Composition Engine), which uses theoretical achievements in order to solve the service composition problem by resorting to TgV.

• Finally, in Chapter 10, our conclusions are drawn.

In addition, we provided an Appendix section, where ongoing work related to data-aware services is reported.
Chapter 2

Related Work

This work follows one of the research directions opened by Daniela Berardi’s Ph.D. thesis [22], that is, the possibility to build a formal framework for composing services, based on the formal notion of simulation relation [106]. On one hand, this work shares a lot of basic literature with [22] while, on the other hand, refers to later works and introduces new contributions. In this chapter, we review the literature on service composition that is relevant, and sometime complementary, to our work, so to provide a state-of-the-art’s big picture and to highlight the main contributions of this thesis.

2.1 Service Modeling

Services can be described in a variety of ways. At a conceptual level, they are programs that export their description, where such description, or model, is not usually defined by means of formal requirements. Since not responding to a strict, formal, definition, one can describe a service based on the properties that are relevant to his/her purposes. As a result, the literature proposes several service models, each corresponding to a different perspective and all equally reasonable. For instance: a service that provides weather information can be seen as a stateless program whose only relevant property is the data format of the information produced, while a service for online payments might export a more complex, stateful, behavior that guarantees the correctness of transactions.

Clearly, when it comes to build automatic procedures, such as those for service composition, the service model assumes a central importance, as it defines the form that the input to procedures may assume and, consequently, the form of reasoning that one needs to adopt. In the following, we report the main approaches to service modeling, present in the literature, that are relevant to our work. In doing so, we adopt a classification similar to the
one proposed in [22], where a service can be described based on the following features:

- interaction model;
- controllability;
- state observability;
- data-awareness;

As for the interaction model, it represents the form of interaction with the client. We say that a service has an atomic interaction model if it is described in terms of the operations that it is able to execute, without restricting the order that they can be performed in, e.g., if one cannot be executed after another; in a (ii) run-based service model, all the sequences of legal operation executions are described, thus introducing a linear model of time (like in LTL); finally, the (iii) tree-based model refers to a branching model of the time: the relevant properties of a service are the choices that it presents after each operation execution (like in CTL). Here, run- and tree- based models are also referred to as behavioral models.

Controllability refers to the possibility of fully control a service, based on the operations it executes. Precisely, a fully-controllable, or deterministic, service is such that for each operation that can be executed in a given situation (or service state), one knows which choices will be available next, before actual operation execution; otherwise, a service is said nondeterministic, or not fully-controllable.

State observability is the characteristic of allowing service internal state to be observed by external services or components.

Finally, we say that a service is data-aware if its operations explicitly deal with data, through input or output parameters.

### 2.1.1 Literature Review

The proposals for service modeling that we take into account here usually refer to Web services, i.e., programs that are accessible over the Web. However, their basic concepts remain valid for services in general, independently of the underlying technology used for implementation, deployment and communication. In other words, we look at such proposals from a high-level perspective, focusing more on the abstraction they provide rather than their actual implementation.
The classical proposals from industry for Web service description, mentioned below, have been published as W3C notes or recommendations to face different issues.

**Web Service Description Language** (WSDL [49, 48]) is a language for static interface specification, that has become a W3C Recommendation in 2007. It allows to declare the set of operations offered by a service, incoming and outgoing messages and data types, based on the XML Schema language ([144, 33, 39]). In addition, it provides (i) the possibility to locate the service (e.g., through a URI), (ii) a protocol to exchange messages, and (iii) the concrete mapping between the abstract method definition and the real protocol and data format. Specifications written in WSDL provide a static representation of a service, in that they do not describe its behavior, e.g., how it reacts to messages or which operations are allowed after others took place.

**Web Service Conversation Language** (WSCL [16]) documents can be used to describe the conversation supported by a Web Service. A WSCL document specifies which (XML) documents can be exchanged and their legal sequences.

**Web Service Choreography Interface** (WSCI, [8]) describes the observable behavior of a Web Service, from clients’ perspective, by representing temporal and logical dependencies among the exchanged messages.

As it comes out, WSDL provides a static representation of a web service, or, according to the above classification, it encodes an atomic interaction model, whereas WSCL and WSCI provide a behavioral representation, or a tree-based interaction model. Moreover, they explicitly deal with data and, importantly, allow only fully-controllable services’ description.

Besides these, other languages have been devised in the industrial world, such as XLANG, WSFL, XPDL, WS-BPEL 2.0 or BPEL4WS, some of which will be discussed later on in this chapter, each with a specific aim (for instance, the BPEL family is aimed at representing service orchestrations). However, they suffer from a “lack of well-defined semantics” [74] that avoids a unique, unambiguous, interpretation of their specifications. Nonetheless, some of them, e.g., WSDL or the BPEL family, represent a widely used standard in actual implementations.

As for the academic world, it is worth mentioning the First-Order Logic Ontology for Web-Services (FLOWS [18, 17]) (also known as the Semantic Web Service Ontology (SWSO)), which has become a W3C submission in 2005. It is the result of the efforts, started in 2001, of the academic world to build a
CHAPTER 2. RELATED WORK

representational framework that is comprehensive with respect to web service features and that may facilitate the automation of several tasks such as service discovering, verification or composition, by providing a well-defined, formal, semantics. Towards this direction, one of the earliest results achieved is the language OWL-S [98, 99] (formerly DAML-S). However, such language was developed in OWL [102] (the Web Ontology Language) which is not expressive enough to represent all and only intended interpretations of the OWL-S process model. As a consequence, an OWL-S specification cannot be interpreted “as is” by a software agent but needs the mediation of a human interpreter to resolve the ambiguities that possibly arise. This is the main issue that motivated FLOWS.

A Semantic Web Service is a Web Service equipped with a formal, machine-interpretable, description of its capabilities, written according to an ontology. Such description can be published so that a client can understand several facets of the service behavior and, possibly, reason over them. In short, a specification compliant with the SWS Ontology includes the following parts:

- the service profile is a description of the service capabilities, oriented to discovery, including service inputs, outputs, preconditions and effects;

- the service model provides a specification of the service process model, i.e., its behavior in terms of control and data flow, and is aimed at enabling automated composition and execution;

- the service grounding focuses on how a SWS can be accessed, that is, describes the communication protocol, data marshaling and serialization. Essentially, service grounding relates the service message types to WSDL messages.

Finally, we mention the Web Service Modeling Ontology (WSMO) [51], which provides another ontology for web services and composition, based on a combination of Description and Horn logic for its semantics.

A lot of work has been done on the SWS architecture or related to the SWS Ontology (cf., e.g., [73, 50, 1, 99]). In [136] (see also [103, 104]), a framework for Web Service composition based on GOLOG [126] is adopted, where services are described as atomic interactions, while a behavioral description of the task to be composed is provided as a GOLOG procedure; a set of (hard [103] or soft [136]) constraints can be specified that are taken into account when the composition is computed. Though such work is not directly cast in OWL-S it has great relevance to it, since, as noted in [136], (i) GOLOG has been the inspiration source for OWL-S and (ii) each OWL-S construct has a GOLOG counterpart [103].
2.1. SERVICE MODELING

Other work that also considers an atomic interaction model for services includes [83, 116, 105, 58]. Besides these, we mention [150, 113, 112], where a common interface that wraps services is proposed. In such work, services are thought of as a kind of class in typical object-oriented languages, that can be extended and reused. The language proposed, SCSL (Service Composition Specification Language), does not allow to describe service behavior.

Another interesting Web service model is described in [141] and, earlier, in [143, 142, 140], where the interaction model is atomic and services are interpreted as views over data sources. Services are described by their input/output parameters, binding patterns and constraints on the data source, which characterize the output data that the service returns. Notably, this work creates an explicit link to the work on Data Integration Systems ([86, 67, 89, 94, 66, 93]).

Further, there are services that export their behavior, which are able to exchange messages with other (peer) services or which interact with a shared database. Behavior-exporting services have been object of great interest in the contest of service composition. Usually, in such context, their description focuses on conversations, i.e., the observable behavior, from a client’s viewpoint, shown through message exchanges. With respect to this, relevant service models are those adopted in [71, 42, 24, 56, 23], just to mention some.

In [56], a web application\footnote{Although the original work deals with Web applications, its relevance and applicability to Web services are straightforward.} is modeled as an automaton, namely an \(ASM^+\) \footnote{Cf. Spielmann’s work on \(ASM\) transducers [137].} (extended abstract state machine), whose behavior depends, step-by-step, on the data that a client –possibly another application– provides as input (hence \textit{data-driven}). In such model, the automaton itself proposes, at each step, a finite set of possible inputs, that are retrieved from an underlying database. In [57], the same model is adopted to verify the behavior of several, communicating, automata. Recently, in [55], a framework similar to the one adopted in [56] has been investigated, in the different context of artifact systems (cf., e.g., [111, 31]), where the user is allowed to introduce, along each system run, (a number of) arbitrary values from an infinitely dense ordered domain.

In [42], services are modeled as (possibly nondeterministic) Mealy machines able to send or receive messages, to or from other peer services, according to a predefined, fixed, communication network. Services communicate (and hence evolve) asynchronously and are equipped with a (bounded) queue which buffers incoming messages. If queues are zero-length, the sender-receiver communication is, clearly, synchronous. The behavior of each service can be summarized as follows: at each step, a service can \textit{send} a message, \textit{receive} a message, \textit{consume} a queue message, if any, or simply perform an empty (i.e.,
not involving messages) move. All pairs of actions are mutually exclusive and yield, in general, state transitions.

A framework which encompasses both message exchange among behavior-exporting services and interaction with a shared database is Colombo [23], where services: (i) are, essentially, represented as deterministic Mealy machines, (ii) are equipped with queues and asynchronously exchange messages with other peers, and (iii) interact with a database through atomic, possibly nondeterministic, processes (whose description language is inspired by OWL-S).

Similarly, from a modeling viewpoint, also [121, 118, 120, 119] propose a behavioral description of services as nondeterministic finite-state processes, actually specified in WS-BPEL and abstracted as finite-state transition systems, that are, in general, partially-observable, i.e., whose state is not completely known, at runtime. Each state represents a service internal state and is characterized by the set of operations that the service offers to clients, whereas transitions represent the state changes that a service performs when an operation is actually executed. On the one hand, nondeterminism models partial knowledge of domain details, which yields uncertainty about the possible outcomes of operation executions; on the other hand, partial observability models the inability to observe all properties of services’ internal state. Also, a language, EAGLE [91], for specifying the constraints, i.e., extended goals, that a composite service is required to meet, is adopted.

It is worth mentioning the work carried out in [13, 19, 20, 21], which focuses on service conversations, also referred to as Web service protocols. More precisely, services are characterized by the potential conversations they can have with other client services, described as a tree, whose edge labels correspond to allowed message send/receive operations and where possibly empty edges refer to internal operations that are not explicitly invoked by client services. Again (see above), finite state automata are well suited for compact representation of such conversational trees ([13, 21]), which can be nondeterministic, as in [20].

Finally, also Process Algebras [37, 128, 100, 101, 114] and Petri Nets [34, 95] have been used to describe tree-based service behaviors.

Summarizing, despite the great variety of actual languages for service behavioral specification, as remarked in [59], when it comes to formal analysis, and in particular synthesis, we find services abstracted by FSMs, Process Algebra or Petri Nets, according to the features of actual services one needs to take into account, the available analysis/synthesis tools, and the goals to be reached.

In this thesis, we adopt Transition Systems as underlying model for services, because they provide a natural, clear, and quite detailed description of services’ behaviors (and their respective conversations) and allow for describ-
2.2 Service Composition

In this section, we take a closer look at the main techniques developed for (Web) service composition, based upon the models presented so far. First of all, let us recall some basics about the problem. As we said, the service composition problem amounts to suitably combine a set, or community, of available services, with the purpose of building a desired service, sometimes referred to as target, not provided by the community. Even thought such a statement is very informal, it is general enough to include all problem specializations that we find in the literature, each of which provides a precise, formal, interpretation of what (i) combining a set of services and (ii) available and target services mean.

In a typical service composition scenario, a (possibly human) client needs a service to perform some task(s). For instance, one might need to access the list of used cars sold in Paris, pick one from the list, see its details and, if in good conditions, buy it or, if conditions are bad, pick another car, then check its conditions and so on, until a car is bought or the client stops the interaction. Or, in a simpler example, the client might simply need to get the current temperature and humidity of a city.

To find a service that meets the desired requirements, the client asks a service provider and, if the target service is available, starts an interaction with it, expected to be compliant with the original request (i.e., it does not requests trucks sold in Rome or a car for rent, in the former example, nor weather conditions, in the latter), to accomplish objectives. The question arises: what if the target service is not available? And the answer follows: check whether it can be obtained by composing the available services! Of course, if a composite service is returned by the provider, this needs to meet the desired requirements, as client’s interaction will be compliant with the original request.

As one may expect, the form that both available and target services are modeled strongly affects the procedures that one can define to find a composition: in a setting where all, including target, services are atomic, e.g., specified in WSDL, a composition based on service chaining seems to be appropriate; if
the target service has a behavioral description then one can reasonably expect a composition of atomic available services in a workflow, formalized, e.g., as a BPEL specification. In the former case, a composition approach which guarantees that the input/output and pre-condition/effects of services’ specifications are coherently composed is well-suited, whereas, in the latter, a planning-like approach seems more appropriate.

As shown in previous section, a great variety of approaches to service modeling and, correspondingly, composition exists. In the following, we review the literature which is relevant to the service composition techniques: a topic that the community is paying great attention to, as witnessed, e.g., by [138].

2.2.1 Literature Review

We start by looking at the industry proposals. In previous section, we mentioned the standard languages WSDL for atomic Web services description, WSCI for defining choreographies and WSCL, for conversation specification. The current standard for specifying service orchestrations and coordinations is WS-BPEL 2.0 [7], a recent improvement of BPEL4WS (Business Process Execution Language for Web Services). Those from BPEL family are XML-based languages, used to define an interaction schema among several services. BPEL specifications represent, essentially, workflows that contain variables and whose transitions represent activities, such as invoking services or sending/receiving messages. Also classical flow-control constructs are available, such as if-then-else or while, one for concurrent activities execution and one for exception management. BPEL implements a tree-based conceptual model, in the sense that, in each state, a set of activity can be executed, based on the evaluation of some conditions. We remark that in BPEL languages an atomic interaction model for component services is assumed. In other words, each available service is seen as an operation provider (described by a WSDL specification).

BPEL, and in particular the latest WS-BPEL 2.0, is the result of an industry effort. It was originally released by IBM, Microsoft and BEA, with the aim of merging the earlier IBM’s WSFL (Web Service Flow Language) and Microsoft’s XLANG, the former proposing Petri Nets as underlying conceptual model and the latter –suitable for specifying both services and orchestrations– based on process algebra. In 2003, it became an OASIS submission and then, in 2007, an official Open standard [7].

It is also worth mentioning the W3C Candidate Recommendation language WS-CDL [87] (Web Services Choreography Description Language) for description of web service choreographies. As we said, the main difference between orchestration and choreography is that the former is mediator-based, i.e., the orchestrator is an automaton that coordinates several services, by
suitably controlling their activations and message exchanges, whereas the latter is peer-based, in the sense that services are mutually controlled by message exchange, without resorting to any external coordinating entity. A WS-CDL specification can be seen as a contract that involves many services, described from an external point of view, thus referring to the services’ observable behavior. Analogously to BPEL, WS-CDL is also XML-based and its underlying abstract model is the $\pi$-calculus [107]: services are described by finite-state transition systems which interact with others and share resources, using predefined communication channels. More in depth, WS-CDL provides constructs for communication, choices, concurrency and iteration, whose formal semantics is the same as analogous $\pi$-calculus constructs.

Despite the efforts spent by the industry on representational aspects of service compositions, which represent, of course, important achievements, very few work has been done on the automated service composition problem, that is, from a practical viewpoint, how a BPEL, or a WS-CDL, specification can be automatically generated, given a description of the available services and a desired target service. To the best of our knowledge, no commercial product is available, yet.

On the other hand, the academic proposals include different techniques, whose theoretical bases span from planning to theorem proving [125], able to deal with both atomic and behavioral interaction models.

**Atomic interaction model**

In [105], a rule-based framework for composition of atomic web services is proposed, where services are classified based on the operations they allow (in an atomic interaction model), that are, in turn, classified according to a taxonomy based on three properties: purpose (what are operations intended to), category (area of interest of the operation) and quality (describing the quality of the operation, according to some measure). A composability model is also defined, needed to check whether services can properly interact and hence contribute to build a new, desired, service. Such service is specified by the client in the language CSSL (Composite Service Specification Language), which allows to describe the desired target conversations. Then, a composition technique, based on the composability models used as rules, is developed for semi-automated composition and a prototypical implementation is described which is able to return the solution in an actual orchestration language. Above mentioned quality parameters allow solutions ranking, when many are available.

The work carried out in [113, 112, 150], tackles the service composition problem by wrapping atomic services as web components. In the same way as
web services can be composite, being services themselves, also a Web component can be composed by basic Web component blocks, according to a defined composition logic. A language for describing composite services is also defined, that the client uses to represent the target service. We also mention the more recent work [151], where a framework for service composition and service life-cycle management is proposed, where composite services are built by reusing, specializing and extending existing ones.

[92] proposes an approach based on AI planning for the composition, execution and monitoring of web services, where the target service is specified in the language XSRL [115]. The planning module is essentially an adapted Model Based Planner [30] (MBP) which takes a description of the target service in input and returns a plan for the available services, that is a controller able to coordinate the execution of the available services so that the target specification is fulfilled.

Another work relying on AI planning techniques is [135], where the HTN (Hierarchical Task Network) planner SHOP2 [109] is used in combination with OWL-S as service description language, to find, compose and execute a target service requested by a client.

Again, from AI, we mention the series of works [104, 103, 136], based on the Situation Calculus. They are characterized by the use of Golog as the language available to clients for specifying their requests. [136] introduced soft constraints, also called preferences, that can be specified by the client in a variant of LTL [32]. In such framework, available services are modeled as actions, which can be either primitive or complex, the latter being combination of the former, described as an action theory [126], i.e., a representation of their preconditions and effects (on the world). For a given input specification, i.e., an action theory and a client request specified as a set of hard —i.e., mandatory—and soft —i.e., preferences— constraints, the objective is to build a plan, or web service composition, that meets the hard constraints and maximizes the preference satisfaction. This is done by the system GologPref, obtained as a modification of PPLAN [32], which executes, in fact, a planning algorithm.

Also techniques based on results from the data-integration area have been devised for automated service composition. In aforementioned works [143, 142, 140], the basic idea is to utilize existing web services as if they were data sources and then execute a plan that retrieves the information requested by the user. In particular, this approach has been applied to bioinformatics web services, which are, often, data retrieval services. The underlying algorithm takes in input the set of available services, represented as data sources, and a user query, consisting of some input data plus a representation of the requested output, and returns an integration plan implementing a template query, which is a query such that all other user input queries different only for the intensional input data can be answered by the same plan, i.e., the composite service.
Behavioral interaction model

Concerning services that export a behavioral interaction model, [120] adopts a model for available services which is analogous to the one used in this thesis, i.e., transition systems. The composition task consists in finding an interleaved execution of available services so that a client-defined target specification is satisfied. In particular, such specifications are expressed as branching temporal formulae requiring the system to reach states satisfying some condition, possibly passing through a desirable path (of states), and, in case of failure, reach a different set of states that satisfy another condition. Roughly speaking, client requests express a main desired system execution plus some side paths to execute for exceptions' handling. As anticipated in the modeling subsection, EaGLe is the language for target service (goal) specification, which is an extension of CTL ([91]), suitable for expressing extended planning goals. The composition technique is based on planning techniques. Specifically, a set of (nondeterministic) transition systems, derived from a corresponding set of BPEL specifications representing available services, is taken as input, along with an EaGLe specification representing the requirements of the target service. Then, a Model Based Planner (MBP), i.e., a planner which exploits BDD-based techniques to contain the state explosion, is used to actually compute the plan that realizes the composition. Such approach has been proven effective in [97]. Importantly, as already noted in [22], once the plan is synthesized, no interaction between the client and the composite service takes place.

Approaches from automata theory also exist. In particular, we mention [42], where a framework for modeling and analysis of global behavior of service compositions is proposed. Each available service is modeled as a Mealy machine, thus a behavioral description is provided, and the focus is on service conversations, i.e., sequences of observable messages exchanged by services during execution. Communication is made possible by a predefined, and fixed, channel topology. In [41] such model is taken as the basis to propose a different conversation model based on collaboration diagrams. As for the composition problem, in [70, 71], a composition algorithm that takes in input (i) a behavioral specification of the available services, formalized as Mealy machines with a cost function defined on transitions, and (ii) a description of the desired conversation, specified as a linear sequence of activities to be executed, is defined. The objective is to find a delegation of each activity to an available service, while minimizing the cost of the obtained solution (properly defined, by combining the cost of each available service transition). We remark that the conceptual model of compositions is run-based, conversations being, in fact, linear sequences of activities.

http://sra.itc.it/tools/mbp/
In [114], behavioral service representations in BPEL or WSCL are called *contracts* and are abstracted by fragments of the language (process algebra) CCS [110] (Calculus of Communicating Systems). A *compliance* relationship between a client and a service CSS specifications occurs if the client always terminates correctly when interacting with the service (in such case the client is *satisfied* by the service); a *subcontract* relation is also defined, which occurs between two services if every client satisfied by one service is also satisfied by the other. Then, a notion of orchestrator—defined as a *bounded*, *directional*, *controlled buffer*—which supervises client-service interaction is defined. It is, in fact, a client-server mediator able to send/receive and forward/filter messages from/to client and service. Finally, an algorithm for synthesizing a mediator that adapts client and service so to satisfy the compliance relation is built.

As we said, this thesis takes [22], and all directly related works (e.g., [24, 27, 26]), as a starting point. The common model adopted in such works abstracts behavior-based services as finite-state transition systems, that can be sometimes deterministic and other times nondeterministic. Importantly, the notion of “composition” is based on the observable behavior of the target service: a target service is realized by suitably combining the available services if the combination reproduces all executions (or possible behaviors) of the target service itself. The works provide a formal problem specification for several scenarios, e.g., when available services are deterministic or nondeterministic, and an actual tool based on logic programming, namely on Propositional Dynamic Logic [64] (PDL), has been built, able to compute an orchestrator that solves the problem. Complexity, with such formulation, was proven to be in EXPTIME, whereas EXPTIME-hardness was shown later by other authors in [108], thus obtaining an EXPTIME-completeness result. The abstract resolution technique adopted throughout [22] was based on the satisfiability of PDL-SAT formulae, implemented by the Description Logics [12] (OWL DL) reasoner Pellet, and the focus was on finding a single solution, i.e., an orchestrator. The major difference with this thesis is in the resolution technique: indeed, we adopt one based on the formal notion of simulation relation [106], which (i) allows the *whole set of solutions* to be computed at once, without increasing the worst-case time complexity, and (ii) yields a series of advantages, such as solution run-time flexibility and failure-tolerance, not present in other approaches.

We also mention the COLOMBO framework [23], which represents a first attempt to combine behavioral-based services with data. In the proposed framework, deterministic available services, essentially represented as Mealy machines, invoke nondeterministic atomic processes, inspired by OWL-S, which interact with an underlying, common, database. The composition problem

\[^{4}\text{http://clarkparsia.com/pellet/}\]
2.2. SERVICE COMPOSITION

consists in “combining” such services so to implement a target service, described the same way as available ones. In this context, available services’ combinations can be either mediator- or choreography-based. Since dealing with a database and infinite data, the general problem is likely to be undecidable (though a formal proof is not provided). However, the work proposes some restrictions which enable reasoning on finite, symbolic, database representations and, ultimately, allow PDL-based techniques as above to be adopted for problem resolution.

Another work relevant to data-aware services is [77], which focuses on model-checking data-aware services for first-order temporal properties that involve data ([76]), expressed as CTL-FO⁺. Although verification of such properties is in general PSPACE-complete, and even EXPTIME-hard if a naïve resolution strategy based on reduction to classical CTL model-checking is adopted, the authors present a reduction, based on “freeze quantification”, that is still exponential but seems to perform better in the practice. Strictly related to the same problem is [56], where the underlying logic is LTL-FO, which defines a technique for model checking web applications (though it can be straightforwardly recast in terms of web services), draws some decidability boundaries and shows that the approach is actually effective in the practice.

Also [15] deals with services that exchange data, though it does not consider an underlying database. The service model adopted in such work is very close, from an abstract viewpoint, to the one adopted in COLOMBO, as available services are seen as Mealy machines. Notably, the authors propose an orchestrator synthesis technique based on controller synthesis literature (e.g.,[9, 117]) and exploit the notion of bisimulation relation, which is an approach similar to the one adopted in this thesis (cf. Ch. 8). In [14], decidability and complexity issues related to the problem are addressed.

In [62], a uniform framework for web service specification, which allows execution of parallel activities, is defined. Specifications are referred to as Synthesized Web Services (SWS) and, essentially, serve as high-level representations of services’ observable behavior, resulting from the possible interactions with external users or other (client) services. Importantly, SWS’s include a local database (which remains unchanged during service execution), that services can access. The work shows how several models, including the Roman Model and ASM⁺, are captured by different SWS’s, which are classified on the basis of their input and query languages. Also, three decision problems, namely emptiness, validation and equivalence of SWS’s are studied, and decidability and complexity bounds are characterized. Finally, the composition problem is addressed, by resorting, among others, to relationships with query rewriting using views (cf., e.g., [3, 45]). The work is a comprehensive analysis of service modeling and composition, which, on the one hand, provides a complexity characterization of problems relevant to services’ analysis and
synthesis, and, on the other hand, generalizes the models of services proposed so far.

2.3 Related Research Areas

In this section, we provide a description of the research areas, different from SOC, that our work is related to.

2.3.1 AI Planning & Reasoning and Multi-Agent Systems

Many works about service composition, such as [136, 120, 135], witness a great relevance of planning in Artificial Intelligence. Roughly speaking, planning is the problem of automatically finding an appropriate combination of actions in a dynamic environment so that certain goals can be achieved (cf., e.g., [35]). This is also a central problem in Control Theory. Planning settings are classified on the basis of actions’ environment’s and goals’ description. So, in the literature, we have, e.g., deterministic vs. nondeterministic actions and environments, temporally extended goals vs. reachability goals, finite-state vs. infinite-state, fully-, partially- or non-observable environments, and so on. When dealing with non-deterministic actions and (partially) observable environment, planning is said to be conditional. Web service composition can be seen as a form of conditional planning, where available services constitute the environment, whose state can be (partially) observed by the orchestrator (i.e., the plan), in order to make decisions, and (ii) goals can be either temporally extended or describe a set of states desired to be reached. This opens the possibility to adopt effective solution approaches already devised in the AI area, successful examples being [135, 120].

In this thesis, we show an application of automated compositions technique to the Multi-Agent System area. As a matter of fact, the transition-system modeling can be exploited to represent the behavior of several agents (available service) and a task to be executed (target service) (cf. e.g., [129, 52]). Clearly, from the abstract viewpoint, nothing changes: target and available services are still abstracted as transition systems. However, when agents are distributed, a central entity that coordinates all behaviors might be no longer available. We show how distributed, communicating, agents can realize a distributed orchestrator, hence mutually coordinate so to execute the desired task.

2.3.2 System Verification & Synthesis

Similarly to AI planning, synthesizing a controller which satisfies a given description (e.g., as an LTL sentence) can be taken as the target problem in a service composition problem reformulation. More precisely, the controller
2.3. RELATED RESEARCH AREAS

represents the orchestrator, i.e., a program that interacts with the available services, on the one side, and, on the other side, with the client service. The requirement it needs to fulfill is to always correctly select an available service for delegation, given all the possible requests a client service could issue. This thesis shows how the problem can be reformulated (see Ch. 8). In this context, the relevant work comes from [10, 11, 123, 122]. In particular, we refer to the problem formulated as a safety game ([10]), i.e., a two-player game, with an environment facing a controller, where the latter wins if it is able to respond to each move of the former, while satisfying a given (safety) condition. In particular, in our case, such condition reflects the correctness of delegations: for each operation request that a client might issue, the controller must delegate the operation to a service that is actually able to execute it. A winning strategy for a safety game is a function which, for every possible game history –i.e., the sequence of states that the game passed through, up to a certain point–, selects the move that the system needs to execute so to satisfy the safety condition. The link between system verification & synthesis, temporal logics, game structures and strategy computation, has been object of many works, e.g., [5, 148, 149, 60, 147, 78, 131], thus building a solid literature which provides the bases for new techniques and efficient approaches (also) to automated service composition –as already done in some cases (see above).

Remarkably, in [15] a similar approach is also adopted, which refers to works for CTL-FO formulae synthesis [9, 117].

We notice that the system-synthesis approach is appealing also as a planning strategy. Indeed, in some situations, one can think of encoding a planning problem as a synthesis problem having the same properties as the one that would be obtained if the starting problem were a service composition one. Such approach is developed in this thesis (see Ch. 7). A similar idea, namely building a new system starting from a set of existing ones, i.e., a component library, is also developed in [96], which also studies the decidability and complexity bounds of the problem.

2.3.3 Data Integration

Data integration is the problem of combining different data sources so to provide users with a unified view of the data (cf., e.g., [93]). Informally put, given several databases, a data integration system allows users to see the data content from all sources as if contained in a single database. From a high-level perspective, service composition has a lot in common with data integration. On one side, we have a set of available data sources that need to be integrated so to provide users with a composite data source, accessible as any other source, whereas, on the other side, a set of available services is given, to be combined in a composite service, made available to users as any other
service. This similarity, along with the observation than in many cases web services play the role of information-retrieval services, suggest to see, as in works [141, 143, 142, 140], available services as data producer or, more precisely, as queries over single data sources, and target services as queries over all data sources, seen as a whole (the so-called global schema). A solution to the problem, then, consists of coordinating the accesses to all data sources, i.e., combining web service executions, so to fulfill the user’s request, i.e, answering a query. To the best of author’s knowledge, these are the only works in the literature \(^5\) that face the service composition problem with such an approach.

### 2.4 Summary of Literature Results

Here, we provide a summary of the main results achieved by the community, that this thesis relies on and integrates with original contributions.

**Service Modeling**

Several models have been proposed for available services:

- input/output specifications, as in WSDL, describe data formats that the service deals with, so to provide clients with information about (i) how service operations can be requested and (ii) what they should expect as output. These models export an atomic, message-oriented, service representation;

- data-unaware behavioral services, as in the Roman Model, focus on sequences of (atomic) operations, or conversations, without explicitly taking into account the presence of data. Such representation abstracts services as transition systems, which are different from FSMs, in that they consider also the options provided by the service, in each state;

- FSM representations, possibly dealing with data through operation parameters, exist, as, e.g., Mealy machines;

- Petri Nets;

- Process Algebras;

- behavioral representation as workflow, e.g., in BPEL, usually adopted for describing composite services, but sometimes exploited also for available service behavioral description;

\(^5\)Actually, other ones, co-authored by a subset of the same authors, do exist. In our opinion, however, the ones cited here are the most representative of the approach.
2.4. SUMMARY OF LITERATURE RESULTS

• services seen as query over data sources;

• in some cases, for instance in the COLOMBO framework and when ASM+ are used, services interact with an underlying database, besides dealing with data through operation parameters;

In this work, we start from the behavioral model based on transition systems proposed in [22], and enrich it with services’ ability to interact with an underlying database. Observe that such a choice is representative also of other models, since many representations are very similar to transition systems. Hence, in our opinion, results on this model can also be very useful, when not straightforwardly re-usable, for other approaches.

Automated Composition

As for the automated service composition, several techniques have been developed:

• from AI planning and reasoning community, techniques based on Golog, PDL and planning as model-checking;

• from database verification, though not actually developed, yet, composition techniques based on modeling services as ASM+s are conceivable;

• a composition framework for data-aware services with an underlying database has been built by resorting to PDL and abstraction techniques from database verification;

• techniques from data integration.

In this work, the approach we propose aims at exploiting, integrating and extending the existing results obtained on the Roman Model, with respect to automated composition techniques, so to overcome some of their shortcomings. Specifically, we propose a technique based on the notion of simulation relation that returns the whole set of solutions to a service composition problem, without increasing the worst-case time complexity of the solution algorithm. The advantages obtained with such approach are several, such as the flexibility of solutions, i.e., the possibility of adapting a solution to exceptional situations that may arise at runtime, and others that will be detailed throughout the thesis.
Chapter 3

The “Roman Model”

We start by introducing a basic service composition proposal which constitutes the core of our framework: the so called “Roman Model”\(^1\) \([24, 27]\), that will be incrementally extended throughout this thesis.

The Roman Model focuses on services’ conversational behavior, i.e., the sequences of atomic interactions that services can potentially have with clients and that, from a general perspective, can be seen as computation fragments. As will be shown, conversational behaviors are compactly represented as finite transition systems (TSs), where each transition label refers to a possible atomic interaction between a client and an available service, also referred to as operation. The goal of the composition task is to realize a new (non available) target service, specified by an external customer as a TS, whose transition labels are chosen among those appearing in available services. Realizing such target service means, in fact, delegating to available services, for actual execution, all operations a client requests. In other words, the purpose of the composition is to implement a service that is, from the client perspective, indistinguishable from the desired (virtual) target service.

3.1 The Framework

In this Section we provide a detailed description of the fundamental blocks of the Roman Model, along with their respective motivating ideas.

Services Following \([24, 27, 29]\), a service is intended as a software artifact characterized by its conversational behavior, that is, its potential evolutions resulting from the interaction with some external system, such as a client service. In other words, a service is a program intended to interact with a client, whose atomic interactions are expected to be conformant with a given,

\(^1\)As referred to by Rick Hull in \([82]\).
known, behavior description. Typically, an atomic interaction results from the following steps:

1. according to its current state, the available service proposes a choice of operations the client can ask for;

2. the client selects one of such operations;

3. the available service executes client’s selection, moves to a new state, according to its behavioral specification, and iterates the procedure.

Client-service conversations can be stopped (by the client), but do not have to, whenever the service is in a “final” state. Services that never terminate, thus offering a continuous interaction, are quite conceivable [27], as well as services starting in non-final states—which means that they need to perform some action in order to reach a final state and possibly terminate legally— or whose initial state is also final—meaning that the service can be not started at all.

In the Roman Model [24, 27], available services are deterministic, so as to capture their full controllability and, hence, the result of executing an operation in a given state is a certain successor state. In other words, by assigning operation execution, one can fully control available services’ transitions. Extensions of the Roman Model to nondeterministic, i.e., not fully controllable, available services have also been studied in [23, 54, 130, 129] and will be discussed in forthcoming Chapters.

In details, as mentioned above, a service (behavior) is represented by a finite deterministic transition system $S = \langle O, S, s_0, \delta, S_f \rangle$, where:

- $O$ is the finite alphabet of operations;
- $S$ is the finite set of states;
- $s_0$ is the initial state;
- $\delta : S \times O \rightarrow S$ is the transition function;
- $S_f \subseteq S$ is the set of final states.

When $\delta(s, a) = s'$ we equivalently write $s \xrightarrow{a} s'$ (is in $S$. Moreover, by a slight abuse of terminology, we will identify a service with its corresponding transition system.

Observe the strong similarity between a TS and a finite state machine (FSM) able to execute operations taken from alphabet $O$. Despite appearances, however, their semantics is profoundly different. Consider the transition systems in Figure 3.1. As FSMs, they would be equivalent, since both represent the language $\{ab, ac\}$, while, as TSs, they are indeed different:
3.1. THE FRAMEWORK

- $TS_1$ is deterministic and models the case in which after operation $a$ one can perform both $b$ and $c$;

- $TS_2$ is nondeterministic and models the case in which, after operation $a$, one is allowed to perform either $b$ or $c$, depending on the actual transition that takes place after executing $a$.

As it turns out, FSM (and language theory) nondeterminism is *angelic* and becomes just a compact way to represent the set of accepted operation sequences. On the other hand, TS nondeterminism is *devilish*, i.e., the client can ask for operation execution but the actual transition is chosen (in a devilish way) by the transition system, that is, the service. As anticipated, in this Chapter, we follow the original proposal of the Roman Model and focus on deterministic transition systems only.

**Available services** Existing programs are referred to as available services and are the only services directly available to clients. They cannot be modified, are defined once for all and evolve according to their behavior. The only way their evolution can be driven is by instructing them to execute legal operation sequences. In general, we deal with many available services $S_i (i = 1, \ldots, n)$, each modeled as a transition system $S_i = \langle O_i, S_i, s_{i0}, \delta_i, S_{if} \rangle$.

**Community and Target Service** A finite set of available services $C = \{S_1, \ldots, S_n\}$ forms a *community*. The available services of a community share the same set of operations $O$, which is, indeed, the result of joining the operation alphabets of all available services, even if some service might not be able to perform all of them. For technical convenience, we introduce the so called community TS –essentially, the *asynchronous product* of all transition systems associated to an available service in a community—, which is a formal descrip-
CHAPTER 3. THE “ROMAN MODEL”

...tion of all community evolutions obtainable by asynchronously executing the available services in all possible ways (see, e.g., [29, 108]).

Definition 3.1.1 Given a community \( C = \{S_1, \ldots, S_n\} \), where \( S_i = (O_i, S_i, s_{i0}, \delta_i, S_f^i) \), the community TS of \( C \) is the transition system \( \mathcal{S}_C = (\mathcal{O}, \mathcal{S}_C, s_{C0}, \delta_C, S_f^C) \), where:

- \( \mathcal{O} \) is the usual shared operation alphabet;
- \( \mathcal{S}_C = S_1 \times \ldots \times S_n \) is the set of \( \mathcal{S}_C \)’s states;
- \( s_{C0} = (s_{10}, \ldots, s_{n0}) \) is the initial state;
- \( S_f^C = S_f^1 \times \ldots \times S_f^n \) is the set of \( \mathcal{S}_C \)’s final states;
- \( \delta_C \subseteq \mathcal{S}_C \times \mathcal{O} \times \mathcal{S}_C \) is the \( \mathcal{S}_C \)’s transition relation, where \( \langle s_1, \ldots, s_n \rangle \xrightarrow{\alpha} \langle s'_1, \ldots, s'_n \rangle \) iff there exists an index \( k \in \{1, \ldots, n\} \) such that \( s_k \xrightarrow{\alpha} s'_k \) in \( S_k \) and for each \( i \neq k \), \( s'_i = s_i \).

Intuitively, given a state \( s_C = (s_1, \ldots, s_n) \in \mathcal{S}_C \), \( \mathcal{S}_C \) can execute an operation if and only if there exists one service \( S_k \) among \( S_1, \ldots, S_n \) that can do it in \( s_k \).

Despite all available services’ determinism, the community TS may happen to be nondeterministic. Precisely, it is the case when in a given community state there are many services able to execute the same operation.

Our goal is to synthesize, given a community \( C \), a new service that realizes a desired behavior. Such a service is called target service and, again, is described as a deterministic TS \( \mathcal{S}_t = (\mathcal{O}, \mathcal{S}_t, s_{t0}, \delta_t, S_f^t) \), in general not included in the community. It needs be realized by exploiting (computation) fragments of available services, since these are the only entities corresponding to actual programs.

The following example makes actual some of the notions just introduced.

Example 3.1.1 (A multi-lingual community) Consider the service community depicted in Fig. 3.2 which provides, through available services, several translation functionalities. In particular, 3.2(a) and 3.2(b) can translate text from, respectively, French and German, into Italian. One can think of them as web services allowing users to send some (French or German) text and, then, to request its translation into Italian. According to their TSs, translations can be asked for only after text is sent. Similarly, available service 3.2(c) provides French-to-Italian translation functionalities, besides allowing for further operations—such as finding synonyms—when German text is introduced (actually, the “sub-behavior” associated to such operations has been compacted into a single state, \( s_1 \), since its details are not relevant to our purposes). Differently from previous services, an Italian translation can be obtained even if no text...
3.1. THE FRAMEWORK

Figure 3.2: Example 3.1.1: Available services for a multi-lingual community

Figure 3.3: Asynchronous product of available services in community of Example 3.1.1.
is explicitly introduced, as shown by the edge looping on state $s_0$ and labeled by action \texttt{output\_it}—in this case a buffer, initially filled out with some default text, is used to record last input.

We refer to TSs associated to services 3.2(a), 3.2(b) and 3.2(c) by means of subscripts $a$, $b$ and $c$, respectively. For instance, TS associated to service 3.2(a) is referred to as $S_a = \langle O_a, S_a, s_{0a}, \delta_a, S_f^a \rangle$. Fig. 3.3 shows the asynchronous product of available services in community $C = \{S_a, S_b, S_c\}$. Note that it is nondeterministic, differently from available services. Given such community, an instance of target service is depicted in Fig. 3.4, where a service that allows for translating either French or German text into Italian is described. □

**Orchestrator** The orchestrator is a component able to activate, stop and resume each of the available services, and select one to perform an (executable) operation. It has full observability on available service states, that is, it can keep track (at runtime) of the current state available services are in. As it will be soon clear, the orchestrator is, ultimately, the object of the composition procedure: it is the component intended to coordinate available service executions so to mimicking, from a client’s viewpoint, the (desired) target service behavior. A formal definition of orchestrator requires a formal background that will be set in the following.

### 3.2 Service Composition

Intuitively, the service composition problem can be stated as follows:

Given a target service and a community, synthesize a composition, i.e., a suitable function that delegates operations, requested by a client to the target service, to the available services in the community (which are the only services actually corresponding to existing programs).

As already discussed, both available and target services are represented by transition systems over a common operation alphabet $\mathcal{O}$. Recall that (i) before
3.2. SERVICE COMPOSITION

any interaction takes place, each available service is in its initial state and (ii) a service can be left (or stopped) only if it is in a final state. Composing a target service amounts to mimicking the desired (target) behavior by properly instructing, for each operation chosen by the client (coherently evolving with the target service), a particular available service for performing the requested operation itself. The orchestrator is responsible for such task and, of course, each of its service selections is constrained by the current state, i.e., the result of operations performed so far, of each available service. In addition, it depends on future operations that, coherently with the target service, can be requested later by the client. Indeed, examples can be found where an assignment that fulfills a client request leads, in the future, to, e.g., dead end situations. As a result, searching for a composition requires a form of “non local” reasoning, i.e., past and future possible evolutions need be taken into account, which makes this task akin to an advanced form of conditional planning [72].

In order to provide a formal statement of the problem, we introduce some preliminary notions.

Definition 3.2.1 Given a service \( S = \langle O, S, s_0, \delta, S_f \rangle \), a trace for \( S \) is a possibly infinite sequence of alternating states and operations, of the form \( s_0 \overset{\alpha}{\rightarrow} s_1 \overset{\alpha}{\rightarrow} \cdots \), such that (i) \( s_0 = s_0 \), and (ii) for all \( j > 0 \), \( s_j \overset{\alpha_{j+1}}{\rightarrow} s_{j+1} \) is in \( S \).

Similarly, let \( C = \{S_1, \ldots, S_n\} \) be a community, where \( S_i = \langle O_i, S_i, s_{i0}, \delta_i, S_{ifi} \rangle \) \((i = 1, \ldots, n)\), and let \( S_C \) be its corresponding TS. A community trace for \( C \) is a possibly infinite sequence of the form \( \langle s_{01}^1, \ldots, s_{0n}^1 \rangle \overset{\alpha^1}{\rightarrow} \langle s_{11}^1, \ldots, s_{1n}^1 \rangle \overset{\alpha^2}{\rightarrow} \cdots \), such that (i) \( \langle s_{01}^1, \ldots, s_{0n}^1 \rangle = \langle s_{10}^0, \ldots, s_{n0}^0 \rangle \) and (ii) for all \( j > 0 \), \( \langle s_{j1}^j, \ldots, s_{jn}^j \rangle \overset{\alpha_{j+1}}{\rightarrow} \langle s_{j+11}^j, \ldots, s_{jn}^j+1 \rangle \) is in \( S_C \).

We call (community) history every finite prefix of a (community) trace ending with a state.

Intuitively, a trace represents a possible service evolution. In particular, we are interested in target service traces, which represent possible target service evolutions induced by a client when issuing consecutive operation requests. Of course, client requests are assumed to be conformant with the target service capabilities (recall that the target service is the desired service a customer wants to obtain to make client services interact with).

Given a service history \( h \), \( \text{last}(h) \) denotes its last state and \( \text{length}(h) \) denotes the number of state-operation alternations it contains. The 0-length history is simply the initial trace state, which simply contains the initial state, i.e., \( s_{i0} \), of the corresponding service. In particular, the 0-length target service history is \( s_{t0} \) and is the same far all target service traces. Observe that states contained in service traces (and histories) are, in fact, redundant, as they can
be reconstructed from the sequence of operations only, each service being deterministic and starting from its respective initial state. Of course, this is not the case for the (nondeterministic, in general) community TS. Since explicit states will be required when extending the framework to deal with nondeterministic services, they are kept here, though redundant, to have a uniform definition for the deterministic and nondeterministic scenarios.

**Composition** Now, we can provide a formal definition of orchestrator.

**Definition 3.2.2** Let \( C = \{S_1, \ldots, S_n\} \) be a community of available services, \( H \) the respective set of all community TS histories and \( O \) the shared operation alphabet. An orchestrator for \( C \) is a possibly partial function \( P : H \times O \to \{1, \ldots, n\} \) such that if \( P(h, o) \) is defined then there exists a transition \( s_k \xrightarrow{o} s'_k \) in \( S_k \), where \( k = P(h, o) \), \( \text{last}(h) = \langle s_1, \ldots, s_k, \ldots, s_n \rangle \).

Intuitively, an orchestrator is a function that, given a community TS history and an operation, can be either defined or not. In the former case, it returns an index in \( \{1, \ldots, n\} \) corresponding to a service (index) that, in the last state of the argument history, is able to execute the operation. Observe that (i) no assumption is made over the community TS history \( h \) provided as function argument, (ii) just a local condition is states, in that nothing is required about past states, and (iii) no requirement is provided to describe “correct” orchestrators, i.e., that “mimick” a given target service. Such orchestrators are called composition and are formalized in the following.

**Definition 3.2.3** Let \( C = \langle S_1, \ldots, S_n \rangle \) be a community and \( S_t \) a target service, \( S_i = \langle S_i, s_{i0}, S_i^f, G_i, \varrho_i \rangle \) \((i = 1, \ldots, n, t)\). Let \( P : H \times O \to \{1, \ldots, n\} \) be an orchestrator for \( C \). Given an \( S_t \) trace \( \tau = s_0 \xrightarrow{o_1} s_1 \xrightarrow{o_2} \cdots \), we say that \( P \) realizes \( \tau \) if and only if:

- for all community TS histories \( h \in H_{\tau, P} \subseteq H \), \( P(h, o^{\text{length}(h)+1}) \) is defined, where \( H_{\tau, P} \) is the set of community service histories induced by \( \tau \) and \( P \), inductively defined as follows:
  - \( h^0 = \langle s_{10}, \ldots, s_{n0} \rangle \in H_{\tau, P} \);
  - \( h^{j+1} \in H_{\tau, P} \) iff \( h^{j+1} = h^j \xrightarrow{o^{j+1}} \langle s_1^{j+1}, \ldots, s_n^{j+1} \rangle \), where \( h^j \in H_{\tau, P} \) and:
    * \( \text{last}(h^j) = \langle s_1^j, \ldots, s_n^j \rangle \);
    * \( o^{j+1} \) is the same as in \( \tau \);
    * \( P(h^j, o^{j+1}) = k \), that is, after community history \( h^j \), the orchestrator instructs available service \( S_k \) to execute operation \( o^{j+1} \);
3.2. SERVICE COMPOSITION

![Mathematical expressions]

Intuitively, $\mathcal{H}_{\tau,P}$ contains all and only those community histories that the orchestrator $P$ generates when instructing available services so to execute/realize a prefix of trace $\tau$. Observe that the definition applies also if $\tau$ is finite. Indeed, for all histories $h$ such that $\text{length}(h) = \text{length}(\tau)$ operation $o_{\text{length}(h)+1}$ is not defined and, consequently, no constraint is required over $P(h,o_{\text{length}(h)+1})$.

We can now introduce the notion of composition.

**Definition 3.2.4** An orchestrator $P$ for $\mathcal{C}$ is a composition of target service $S_t$ by $S_C$ iff it realizes all $S_t$ traces.

The idea behind this definition is that an orchestrator realizes a target service by composing available services if and only if (i) it is always able to delegate, to some available service, each operation that the target service would be able to execute and (ii) each time the target service is allowed to stop, so are all available services. In other words, the orchestrator implements an available services’ controller that mimics target service behavior.

**The Composition Problem** We can now provide a formal statement of the composition problem in a deterministic setting:

Given a community $\mathcal{C} = \{S_1, \ldots, S_n\}$ and a deterministic target service $S_t$, synthesize an orchestrator for $\mathcal{C}$ that is a composition of $S_t$ by $S_C$.

**Computational complexity characterization** In [24, 27], a slightly different notion of composition, based on the so-called execution trees, is introduced, were the set of histories is explicitly represented as a tree and the orchestrator is a function of the target service histories, rather than community’s. However, such notion is just an equivalent syntactic variant of the one presented here, so the same complexity bounds apply. In particular, the upper bound was established in [24]:

**Theorem 3.2.1** Checking for the existence of a composition of $S_t$ by $S_C$ can be done in EXPTIME.

Interestingly, a matching lower bound was recently proved by Muscholl and Walukiewicz ([108]):
Theorem 3.2.2 Checking for the existence of a composition of $S_t$ by $S_C$ is EXPTIME-hard.

In other words, checking for the existence of a composition is an EXPTIME-complete problem. Also synthesizing an orchestrator that is a composition is an EXPTIME-complete problem ([24, 27, 108]).

Notably, in [24, 27] an actual synthesis technique is presented, based on a polynomial reduction to satisfiability in Propositional Dynamic Logic [79]. The aim of this work is to present a novel synthesis technique, based on the formal notion of simulation relation [106], that is still optimal with respect to worst-case time complexity and, yet, refines the complexity upper bound of the problem.

3.3 Composition and Simulation

The first step, and basic result, of our work is to show that checking for the existence of a service composition is equivalent to checking for the existence of a simulation relation between the target and the community TSs. Let us introduce the notion of simulation relation ([106]) in our context.

Definition 3.3.1 Given two transition systems $S_t$ and $S_C$, a simulation relation of $S_t$ by $S_C$ is a relation $R \subseteq S_t \times S_C$, such that:

\[ \langle s_t, s_C \rangle \in R \]  implies:

1. if $s_t \in S_f^t$ then $s_C \in S_f^C$;

2. for each transition $s_t \xrightarrow{o} s'_t$ in $S_t$ there exists a transition $s_C \xrightarrow{o} s'_C$ in $S_C$ and $R(s'_t, s'_C)$.

Informally, for a state $s_t \in S_t$ being in a simulation relation $R$ with state $s_C \in S_C$ means that:

- if $s_t$ is final then so is $s_C$;

- for every operation $o$ and state $s'_t$, if $s_t$ can perform a transition to $s'_t$ when executing operation $o$, then $s_C$ can perform a transition to some $s'_C$ when executing the same operation $o$, in a way such that $s'_t$ is still in the same simulation relation $R$ with $s'_C$.

Observe the coinductive nature of the definition: indeed, it is cyclic but with no base case. Based on this, we define when a community state simulates a target state:
3.3. COMPOSITION AND SIMULATION

Definition 3.3.2 Let $S_t$ be the transition system representing the target service and $S_C$ be the community transition system, as above. A state $s_t \in S_t$ is simulated by a state $s_C \in S_C$ (or $s_C$ simulates $s_t$), denoted $s_t \preceq s_C$, if and only if there exists a simulation relation $R$ of $S_t$ by $S_C$ such that $R(s_t, s_C)$.

Observe that $\preceq$ is itself a simulation relation and, precisely, the largest one since, by above definition, all simulation relations are contained in $\preceq$.

Finally, we define when the community TS simulates the target TS:

Definition 3.3.3 $S_t$ is simulated by $S_C$ (or $S_C$ simulates $S_t$) iff $s^0_t \preceq s^0_C$, where $s^0_t$ and $s^0_C$ are the initial states of the target and the community TSs, respectively.
Example 3.3.1 (Example 3.1.1, continued) Consider the target (Fig. 3.4) and the community (Fig. 3.3) TSs of Example 3.1.1. In Figure 3.5, a simulation of the former by the latter is shown. Dashed lines associate each state of the target TS to those states of the community TS it is simulated by. For instance, $S_C$ state $\langle s_1, s_0, s_0 \rangle$ simulates $S_t$ state $s_1$ as well as state $s_0$ is simulated by both $\langle s_0, s_0, s_0 \rangle$ and $\langle s_0, s_0, s_1 \rangle$. Note that, in general, there may exist several simulations. The one shown in Figure 3.5 represents the largest one, i.e., the relation $\preceq$.

Now, we can show how checking for the existence of a service composition can be reduced to checking whether the target transition system is simulated by the community transition system. We do so by showing two intermediate results.

**Lemma 3.3.1** Given a target and a community TSs, $S_t$ and $S_C$, if there exists an orchestrator $P$ that is a composition of $S_t$ by $S_C$ then $s_{i0} \preceq s_{C0}$.

**Proof:** We need to show that there exists a simulation relation, say $R$, such that $\langle s_{i0}, s_{C0} \rangle \in R$. To this end, define $R \subseteq S_t \times S_C$ as follows. For each $S_t$ trace $\tau$, let $\mathcal{H}_{\tau,P}$ be the set of community histories induced by $P$ and $\tau$, as in Definition 3.2.3, and $h^\ell$ be its (only) $\ell$-length history. Let $\langle s_t, s_C \rangle \in R$ iff there exists an $S_t$ trace $\tau = s_t^0 \xrightarrow{o_1} s_t^1 \xrightarrow{o_2} \ldots$ such that $s_t = s_t^i$ and $s_C = \text{last}(h^i)$, for some $i$. We show, by induction on the length of $\mathcal{H}_{\tau,P}$ histories, that $R$ is a simulation relation and $\langle s_{i0}, s_{C0} \rangle \in R$.

- Clearly, $\langle s_{i0}, s_{C0} \rangle \in R$, as $s_t^0 = s_{i0}$ and $h_{\tau} = s_{C0}$ for each trace $\tau$.

- Assume $\langle s_t, s_C \rangle \in R$. Then, by $R$ definition, there exists a trace $\tau = s_t^0 \xrightarrow{o_1} \ldots \xrightarrow{o_{i-1}} s_t^i$ and a corresponding $\mathcal{H}_{\tau,P}$ history $h^i$ such that, for some $i$, $s_t = s_t^i$ and $s_C = \text{last}(h^i)$. First, observe that $(\dagger)s_t \in S_t^\ell \Rightarrow s_C \in S_C^\ell$ is a consequence of Definition 3.2.3, $P$ being a composition and, hence, realizing all $S_t$ traces $\tau$. Now, for each operation $o \in \mathcal{O}$ (possibly $o^\ell$) such that $s_t^i \xrightarrow{o} s_t' \in S_t$, by definition of service trace, there exists an $S_t$ trace $\tau' = s_t^0 \xrightarrow{o_1} \ldots \xrightarrow{o_{i-1}} s_t^i \xrightarrow{o} s_t' \ldots$ (possibly $\tau$) and, $P$ being a composition, a corresponding $(i+1)$-length history $h_{\tau} = h_{\tau}^i \xrightarrow{o} s_C'$ in $\mathcal{H}_{\tau,P}$. Hence, $\langle s_t', s_C' \rangle \in R$, by $R$ definition. Moreover, by definition of community history, there exists a transition $s_C' \xrightarrow{o} s_C'$ in $S_C$. So, we get that for each operation $o \in \mathcal{O}$ such that $s_t^i \xrightarrow{o} s_t'$ in $S_t$, there exists a transition $s_C' \xrightarrow{o} s_C'$ such that $\langle s_t', s_C' \rangle \in R$. This, together with $(\dagger)$, completes the proof. \(\square\)
The above lemma states that, given $S_t$ and $S_C$, if there exists a composition of $S_t$ by $S_C$ then $S_t$ is simulated by $S_C$. The converse also holds, as shown in the following.

**Lemma 3.3.2** Given a target and a community TS’s, $S_t$ and $S_C$, if $S_t$ is simulated by $S_C$ then there exists an orchestrator $P$ for $C$ that is a composition of $S_t$.

**Proof:** Let $R \subseteq S_t \times S_C$ be a simulation relation containing $\langle s_{t0}, s_{C0} \rangle$. We inductively build (i) a set $H_{P,t}$ of sequences of the form $\langle s_i^0, s_i^C \rangle \xrightarrow{o^1} \langle s_i^1, s_i^C \rangle \xrightarrow{o^2} \cdots \xrightarrow{o^\ell} (s_i^\ell, s_i^C)$, where $o^i \in O$, $s_0^0 = s_{t0}, s_i^C \in S_C$, functions length and last are defined as usual, and (ii) an orchestrator $P$ for $C$ that is a composition of $S_t$.

First, define, for each sequence $h^m = (s_{t0}, s_{C0}) \xrightarrow{o^1} (s_1^0, s_1^C) \xrightarrow{o^2} \cdots \xrightarrow{o^n} (s_m^0, s_m^C)$, a corresponding sequence $\hat{h}^m = s_{C0} \xrightarrow{o^1} s_1^C \xrightarrow{o^2} \cdots \xrightarrow{o^n} s_m^C$. Now, we inductively define $H_{P,t}$ and $P$, as follows:

- (Base case) Let $H_{P,t}^0 = \{ (s_{t0}, s_{C0}) \}$. By hypothesis, $(s_{t0}, s_{C0}) \in R$.

- (Induction step) Let $H_{P,t}^m \subseteq H_{P,t}$ be a set of $m$-length sequences as above and assume that for all such sequences $last(h^m) = \langle s_i^m, s_i^C \rangle \in R$. Then, for each $o \in O$ such that $s_i^m \xrightarrow{o} s_i^{m+1}$ is in $S_t$, there exists a transition $s_i^C \xrightarrow{o} s_i^{C+1}$ in $S_C$ such that $\langle s_i^{m+1}, s_i^{C+1} \rangle \in R$, $R$ being a simulation relation. For each of such $o$’s, choose one respective transition in $S_C$ (in general, there are many) and let $\hat{h}^m \xrightarrow{o} (s_i^{m+1}, s_i^{C+1}) \in H_{P,t}^{m+1}$. By definition of community TS, if $s_i^C \xrightarrow{o} s_i^{C+1}$ in $S_C$ then there exists an index $k \in \{1, \ldots, n\}$ such that $s_k^{m} \xrightarrow{o} s_{k+1}^m$, where $s_i^C = \langle s_1^m, \ldots, s_k^m, \ldots, s_n^m \rangle$ and $s_i^{C+1} = \langle s_1^{m+1}, \ldots, s_{k+1}^m, \ldots, s_n^m \rangle$. So, for each $o \in O$ such that $s_i^m \xrightarrow{o} s_i^{m+1}$ in $S_t$, let $P(\hat{h}^m, o) = k$, where $k$ is chosen from those as above.

The proof is completed by showing that for each $\ell$-length target service history $h^\ell_t$, (i) there exists a corresponding $\ell$-length sequence $h^\ell \in H_{P,t}$ such that $h^\ell_t = \hat{h}^\ell$ and (ii) if $last(h^\ell_t) \in S_{t1}$ then $last(h^\ell) \in S_{tC}$. Both (i) and (ii) are direct consequences of $H_{P,t}$ definition and the fact that $R$ is a simulation relation.

As a direct consequence of Lemmas 3.3.1 and 3.3.2, we get the following, fundamental, theorem.

**Theorem 3.3.1** A composition of the target service $S_t$ for community $C = \{S_1, \ldots, S_n\}$ exists if and only if $S_t$ is simulated by $S_C$. 

**Proof:** Direct consequence of Lemmas 3.3.1 and 3.3.2.

So, based on Theorem 3.3.1, we get a straightforward method to check for the existence of a composition, namely:

- compute the maximal simulation relation $\preceq$ of $S_t$ by $S_C$;
- check whether $\langle s_{t0}, s_{C0} \rangle$ is in such a relation.

From the computational point of view, recall that checking for the existence of a simulation relation between two (states of two) transition systems can be done in polynomial time in the size of the transition systems—actually, optimized techniques exist for computing simulation, such as those in [80, 139, 68]. Since in our case the number of $S_C$ states is exponential in the size (i.e., the number of states) of $S_1, \ldots, S_n$, we can check for the existence of a composition using simulation in exponential time. Considering that the problem is EXPTIME-complete, we get the following result:

**Theorem 3.3.2** Checking for the existence of compositions via simulation is optimal with respect to worst-case complexity.

Notably, the EXPTIME-completeness of service composition yields the following result in the context of simulation, which closes a long standing open problem in the simulation literature:

**Theorem 3.3.3** Checking simulation from a single deterministic transition system to the asynchronous product of $n$ deterministic transition systems is an EXPTIME-complete problem.

**Proof:** The membership to EXPTIME is a direct consequence of Theorem 3.2.1 and Theorem 3.3.2. As for the EXPTIME-hardness, if a lower complexity technique would exist for the simulation problem above it could be applied to service composition as well, leading to a lower complexity technique for service composition itself and thus contradicting the EXPTIME-hardness result of Theorem 3.2.2.

In [134], the computational complexity characterization of checking simulation from the asynchronous product of $n$ concurrent deterministic transition systems to a single deterministic transition system was given. However, the computational complexity characterization of checking simulation in the converse direction remained open. We close it, by transferring EXPTIME-completeness result of service composition to simulation ([29]).
3.4 Synthesizing Compositions via Simulation

Theorem 3.3.1 relates the notion of simulation relation to the one of service composition showing, ultimately, that checking for the existence of a service composition is equivalent to checking for the existence of a simulation relation between the target and the community TS. However, no procedure is given for actually synthesizing an orchestrator that implements a composition, by properly assigning operation executions to available services. In this section, we show that if a simulation relation exists, it can be used to synthesize a composition. To this end, we refer to an abstract structure called composition generator, or CG for short. Intuitively, the CG is a program that returns, for each state the community may potentially reach while realizing a target history and for each operation the client may request in such state, the set of all available services able to perform the given operation in the given community state so that future client requests can be still fulfilled. The CG is directly obtained by the maximal simulation relation between the target and the community TSs, as required by its formal definition:

Definition 3.4.1 Let $S_t$ be a target service and $C = \{S_1, \ldots, S_n\}$ a community of available services such that $S_t$ is simulated by $S_C$. The composition generator for $S_t$ by $S_C$ is a tuple $CG = \langle O, \{1, \ldots, n\}, S_g, s_{g0}, \omega_g, \delta_g \rangle$, where:

1. $O$ is the (usual) finite set of community operations;
2. $\{1, \ldots, n\}$ is the set of available service indices;
3. $S_g = S_t \times S_1 \times \ldots \times S_n$ is the set of CG states;
4. $s_{g0} = \langle s_{t0}, s_{10}, \ldots, s_{n0} \rangle$ is the CG’s initial state;
5. $\omega_g : S_g \times O \rightarrow 2^{\{1, \ldots, n\}}$ is the service selection function. For $s_g = \langle s_t, s_1, \ldots, s_n \rangle \in S_g$, $\omega_g(s_g, o)$ is defined if and only if:
   - $s_t \preceq \langle s_1, \ldots, s_n \rangle$ and
   - $s_t \xrightarrow{a} s'_t$ in $S_t$ (for some $s'_t$);
   
   in such case,
   $$\omega_g(s_g, o) = \{k \mid s_k \xrightarrow{o} s'_k \text{ is in } S_k \text{ and } s'_t \preceq \langle s_1, \ldots, s'_k, \ldots, s_n \rangle\};$$
6. $\delta_g : S_g \times O \times \{1, \ldots, n\} \rightarrow S_g$ is the CG transition function. $\delta_g(s_g, o, k)$ is defined iff $k \in \omega_g(s_g, o)$, as follows:
   $$\delta_g(s_g, o, k) = s'_g,$$
   where
   - $s'_g = \langle s'_t, s_1, \ldots, s'_k, \ldots, s_n \rangle$, $s_t \xrightarrow{a} s'_t$ in $S_t$ and $s_k \xrightarrow{o} s'_k$ in $S_k$. 

Observe how the CG is essentially a TS, except for the presence of output function $\omega_g$, index $k$ labeling transitions and the absence of final states. Similarly to TSs, when $\delta_g(s_g,o,k) = s'_g$, we equivalently write $s_g \xrightarrow{\alpha_k} s'_g$ (is) in CG. In addition, as a notational convention, we refer to the community components of a CG state as simply $s_C$, when no ambiguity arise. So, for instance, we equivalently represent a CG state $\langle s_t, s_1, \ldots, s_n \rangle$ as $\langle s_t, s_C \rangle$, $s_C$ standing for $\langle s_1, \ldots, s_n \rangle$.

Informally, the CG is a finite state machine that, given an operation $o$, outputs (function $\omega_g$) the set of services which can perform $o$ next, according to the current community state. More in detail, observe that each CG state $s_g = \langle s_t, s_C \rangle$ represents a pair of target, $s_t$, and community, $s_C$, states such that $s_t \preceq s_C$. Then, for each operation the target service can execute in $s_t$, there exists an available service $S_k$ that can execute it (in $s_C$), according to the simulation relation. The CG outputs all indices of available services able to execute $o$, when the community is in $s_C$, so to guarantee that also future (target-compliant) operation requests can be fulfilled. Moreover, the CG evolves coherently with the community and the target service (function $\delta_g$) when an available service is selected to execute $o$. In other words, the CG describes the synchronous execution of $S_t$ and $S_C$ when the latter “realizes” the former. As anticipated by its name, the CG allows to generate (all) compositions of $S_t$ by $S_C$. Indeed, it is enough picking up, at each step, one among the services returned by $\omega_g$ while the CG “follows” $S_t$ execution.

In order to formalize how compositions are actually generated starting from the CG, we introduce some preliminary notions and results.

**Definition 3.4.2** Given a CG for $S_t$ and $S_C$, a CG trace is a possibly infinite sequence $s_0^g \xrightarrow{\alpha_1^t, k_1} s_1^g \xrightarrow{\alpha_2^t, k_2} \ldots$ such that (i) $s_0^g = s_{g0}$ and (ii) for each $i \geq 0$, $s_i^g \xrightarrow{\alpha_{i+1}^t, k_{i+1}} s_{i+1}^g$ in CG. A CG history is any finite prefix of a trace, ending with a state. We define $H_g$ as the set of all CG histories.

From Definitions 3.4.1 and 3.4.2 we get the following lemma.

**Lemma 3.4.1** If $s_0^g \xrightarrow{\alpha_1^t} s_1^g \xrightarrow{\alpha_2^t} \ldots$ is a CG trace, then for all $s_i^t = \langle s_t, s_C^i \rangle$ ($i \geq 0$), $s_i^t \preceq s_C^i$.

**Proof:** Comes directly from $\omega_g$ and $\delta_g$ in Definition 3.4.1. \qed

Observe that, by CG state and transition definitions, from a CG trace (history) one can derive an $S_t$ and an $S_C$ trace (history). Indeed, it is enough projecting out one (i.e., the non-relevant) component from each state $s_i^t = \langle s_t^i, s_C^i \rangle$ and remove the index $k$ from each transition. Formally:
Definition 3.4.3 Given a CG trace \( \tau_g = s^0_g \xrightarrow{\alpha^1,k^1} s^1_g \xrightarrow{\alpha^2,k^2} \ldots \), where \( s^i_g = \langle s^i_t, s^i_C \rangle \),

- the CG-derived community trace \( \tau_C \) is the sequence \( \tau_C = s^0_C \xrightarrow{\alpha^1} s^1_C \xrightarrow{\alpha^2} \ldots \), obtained by projecting out from each \( \tau_g \) state \( s^i_g \) the \( S_t \) component and by removing the index \( k^i \) from each transition;

- the CG-derived target service trace \( \tau_t \) is the sequence \( \tau_t = s^0_t \xrightarrow{\alpha^1} s^1_t \xrightarrow{\alpha^2} \ldots \), obtained by projecting out from each \( \tau_g \) state \( s^i_g \) the \( S_C \) component and by removing the index \( k^i \) from each transition;

- a CG-derived community (target service) history is any finite prefix of a CG-derived community (target service) trace, ending with a state.

It is easily seen, by CG definition, that

Lemma 3.4.2 Each CG-derived community (target service, respectively) trace is, in fact, a community (target service) trace, and analogously for histories.

We have also the converse for target service traces and histories.

Lemma 3.4.3 For each target service trace \( \tau_t \) (history \( h_t \), respectively) there exists a CG trace \( \tau_g \) (history \( h_g \), respectively) such that \( \tau_t \) (\( h_t \), respectively) is CG-derived from \( \tau_g \) (\( h_g \), respectively).

Proof: It is a consequence of \( \delta_g \) and \( \omega_g \) definitions in Definition 3.4.1, which yield that if \( s_g \xrightarrow{\alpha,k} s'_g \) in CG then \( s_t \preceq s_C \) and \( s'_t \preceq s'_C \), where \( s_g = \langle s_t, s_C \rangle \) and \( s'_g = \langle s'_t, s'_C \rangle \), and the fact that \( S_t \) is simulated by \( S_C \) (otherwise CG would be not defined), i.e., \( s_{t0} \preceq s_{C0} \).

In other words, the whole (and exact) set of target service traces corresponds to the set obtained by CG-deriving all CG traces. We can now define how a composition can be extracted from the CG.

Definition 3.4.4 A CG-composition is an orchestrator \( P : \mathcal{H} \times \mathcal{O} \rightarrow \{1, \ldots, n\} \) satisfying the following, inductively defined, requirement:

- let \( s^0_g \in \mathcal{H}_P \), where \( \mathcal{H}_P \subseteq \mathcal{H}_g \) and \( s^0_g = \langle s_{g0}, s_{C0} \rangle \);

- if \( h \in \mathcal{H}_P \) then for all \( o \in \mathcal{O} \) such that \( \omega_g(\operatorname{last}(h), o) \) is defined:
  - \( P(h_C,o) = k \in \omega_g(\operatorname{last}(h), o) \), where \( h_C \) is the community history CG-derived from \( h \);
  - \( h \xrightarrow{\alpha,k} s^\operatorname{length}(h)+1_g \in \mathcal{H}_P \), where \( \operatorname{last}(h) \xrightarrow{\alpha,k} s^\operatorname{length}(h)+1_g \) is in CG;
Recall that the CG represents the synchronous evolution of the target service and the community. The idea behind this definition is that, while the CG “follows” the evolution of the (virtual) target service and the (actual) community, a composition can be generated by choosing a service among those \( \omega_g \) returns in the current state, given the next operation requested by the client. In detail, starting from the initial state, the client requests an operation \( o \) (executable by the target service); then, the CG outputs, through \( \omega_g \), the set of available services able to execute \( o \). Observe that such services guarantee not only \( o \) executability, but also that however the client evolves, the community will be able to satisfy its future requests. At this point, any service among those returned by \( \omega_g \) can be chosen to satisfy the request; based on such choice and \( o \), the CG performs a transition, reaching a new state, where a new client request triggers a new set of services that can fulfill the request itself, the CG moves, and so on. Observe that \( P \) is required to satisfy the constraint only along the histories it “induces”, as others are never reached when realizing a target trace according to \( P \).

Example 3.4.1 (Example 3.3.1, continued) As a CG instance, consider Figure 3.6, where a graphical representation of \( CG = \langle O, \{1, \ldots, n\}, S_g, s_g, \omega_g, \delta_g \rangle \) obtained by the simulation relation of Figure 3.5 is shown. According to Definition 3.4.1, nodes are labeled by four components representing, respectively, states of \( S_t \), \( S_a \), \( S_b \) and \( S_c \). Edges are labeled by pairs of the form \( I/O \), where \( I \in O \times \{1, \ldots, n\} \) and \( O \in 2^{\{1, \ldots, n\}} \), with the following semantics: an edge \( e \) connects node \( s \) to node \( s' \) with label \( \langle o, i \rangle/O \) if and only if \( \omega_g(s, o) = O \), \( i \in O \) and \( \delta_g(s, o, i) = s' \). Even though the CG includes two disconnected components, only the one containing the initial state is relevant for composition generation. Indeed, when generating a composition, according to Definition 3.4.4, the CG essentially “follows” a target service trace, actually generated by the client, and, step by step, outputs the set of available services able to execute the operation currently requested. Since all services start in their initial state, so does the CG and, hence, it can never “jump” out of the connected component containing the initial state itself.

Starting from this CG, several compositions can be obtained, depending on the service selected for performing each operation. Intuitively, generating a composition corresponds to “unfolding” a CG and labeling the resulting edges by choosing one among the services proposed by the \( \omega_g \). For instance, in Figure 3.4.1 two different orchestrators are reported. Edges are labeled with pairs \( o/i \), where \( o \in O \) and \( i \in \{1, \ldots, n\} \) represent, respectively, the operation the client requested and the available service choice.

Since, by Lemmas 3.4.2 and 3.4.3, all target service traces can be derived from CG ones, all orchestrators generated by the CG are, in fact, compositions,
3.4. SYNTHESIZING COMPOSITIONS VIA SIMULATION

Figure 3.6: CG for Example 3.3.1

Figure 3.7: Two different generated orchestrators for CG of Example 3.4.1
as the following Theorem shows.

**Theorem 3.4.1** Let $C$, $S_t$, $S_C$ and $CG$ be as above. If $P$ is a CG-composition then $P$ is a composition of $S_t$ by $S_C$.

**Proof:** Let $P$ be a generated composition as in Definition 3.4.4 and consider a generic infinite (the finite case is a straightforward specialization) target service trace $\tau = s^0_1 \stackrel{\alpha_1}{\rightarrow} s^1_1 \stackrel{\alpha_2}{\rightarrow} \ldots$. We show that $P$ realizes $\tau$, i.e., (i) for all $\ell$-length community histories $h^\ell \in H_{\tau,P}$, where $H_{\tau,P}$ is as in Definition 3.2.3, $P(h^\ell, o^{\ell+1})$ is defined and (ii) for all $s^\ell_t \in \tau$ such that $s^\ell_t \in S^\ell_t$, history $h^\ell \in H_{\tau,P}$ is such that $last(h^\ell_C) \in S^\ell_C$.

For (i), we proceed by induction on the length of histories in $H_{\tau,P}$ and $H_P$ (as in Definition 3.4.4):

- Let $h^0 = s^0_C \in H_{\tau,P}$ be the community history CG-derived from $h^0_g = s^0_0 = \langle s^0_t, s^0_C \rangle \in H_P$. Since, by CG existence, $s^0_t \preceq s^0_C$ and, by $\tau$ definition, there exists a transition $s^0_t \xrightarrow{\alpha} s^1_t$ in $S_t$, then $\omega_g(last(h^0_g), o^1)$ is defined and so is $P(h^0_g, o^1)$;

- Assume that $P(h^l, o^{l+1})$ is defined for all $0 \leq l \leq m - 1$, where each $h^l_C \in H_{\tau,P}$ is CG-derived from the $\ell$-length CG history $h^\ell_g \in H_P$, according to Definition 3.4.4, and each $o^{\ell+1}$ is as in $\tau$. By Definition 3.4.4, $\omega_g(last(h^m_g), o^m)$ is defined as well and, hence, $s^{m-1}_t \preceq s^{m-1}_C$, for $last(h^{m-1}_g) = \langle s^{m-1}_t, s^{m-1}_C \rangle$. Moreover, by $\tau$ definition above there exists a transition $s^{m-1}_t \xrightarrow{\alpha} s^m_t$ in $S_t$. Let $P(h^{m-1}, o^m) = k^m \in \omega_g(last(h^m_g), o^m)$. By Definition 3.4.1, $last(h^m_g) \xrightarrow{o^m k^m} s^m_g$ is in CG with $s^m_g = \langle s^m_t, s^m_C \rangle$, for some $s^m_t \in S_t$. But then $h^m_g = h^{m-1}_g \xrightarrow{o^{m-1} k^m} s^m_g \in H_P$ and $h^m_C = h^{m-1} \xrightarrow{o^{m-1}} s^m_C \in H_{\tau,P}$, with $h^m$ CG-derived from $h^m_g$. By Lemma 3.4.1, $s^m_t \preceq s^m_C$, hence for $s^m_t \xrightarrow{o^m k^m} s^m_{t+1}$ in $S_t$ as in $\tau$, we get that $\omega(last(h^m), o^{m+1})$ is defined and, therefore, so is $P(h^m, o^{m+1})$.

For (ii), it is enough observing that for all histories $h^\ell_g \in H_P$, $s^\ell_t \preceq s^\ell_C$, with $last(h^\ell_C) = \langle s^\ell_t, s^\ell_C \rangle$. \hfill $\square$

Even more importantly, the vice versa holds: every composition can be obtained by suitably choosing, at each step, one element from those in CG’s selection function $\omega_g$. Formally, we have the following theorem.

**Theorem 3.4.2** Let $C$, $S_t$, $S_C$ and $CG$ be as above. If $P$ is a composition of $S_t$ by $S_C$ then $P$ is a CG-composition.
By contradiction, suppose \( \tau = s^0_1 \xrightarrow{o^1} s^1_1 \xrightarrow{o^2} \ldots \) be an infinite target trace (the finite case is a straightforward specialization). From this, consider the target history \( h^\ell \) obtained as \( \ell \)-length prefix of \( \tau \). Given \( P \) and \( \tau \), let \( h^\ell \in \mathcal{H}_{\tau,P} \) be the \( \ell \)-length community history as in Definition 3.2.3. Clearly, \( h^\ell \) is defined because \( s \) and \( s' \) exist: \( s = (s^0_t, s^t_C) \), \( s^t_C = \text{last}(h^t_C) \) and \( o^\ell + 1 \) is the same as in \( \tau \).

By contradiction, suppose \( \tau \) does not satisfy (\( \dagger \)) and let \( \bar{\ell} \) be the minimum index such that \( P(h^\bar{\ell}, o^{\bar{\ell} + 1}) \) is defined but (\( \ddagger \)) \( P(h^\bar{\ell}, o^{\bar{\ell} + 1}) = k^{\bar{\ell} + 1} \notin \omega_g(\text{last}(h^\bar{\ell}_g), o^{\bar{\ell} + 1}) \). Thus, considering \( \omega_g \) definition in Definition 3.4.1 and observing that there exists a transition \( s^\ell_i \xrightarrow{o^{\ell + 1}_i} s^{\ell + 1}_i \) in \( S_t \), we necessarily get that either (i) \( \omega_g(\text{last}(h^\bar{\ell}_g), o^{\bar{\ell} + 1}) \) is not defined because \( s^\ell_i \not\in s^\ell_C \) or (ii) \( s^{\ell + 1}_i \not\in s^{\ell + 1}_C \), where \( h^{\ell + 1}_g = h^{\bar{\ell}} o^{\ell + 1, k^{\bar{\ell} + 1}} s^{\ell + 1}_g \) and \( h^{\ell + 1}_g = \langle s^{\ell + 1}_t, s^{\ell + 1}_C \rangle \) (otherwise (\( \ddagger \)) would hold for \( \ell = \bar{\ell} \)). But, due to finiteness of state spaces, \( (s^\ell_i \not\in s^\ell_C) \lor (s^{\ell + 1}_i \not\in s^{\ell + 1}_C) \) yields that there must exist:

- a target service trace \( \tau' = s^0_1 \xrightarrow{o^1} \ldots \xrightarrow{o^{\ell + 1}_i} s^{\ell + 1}_i \xrightarrow{o^{\ell + 2}} \ldots \), where \( s^\ell_i \), \( o^\ell \) and \( s^t \) are the same as in \( \tau \) for \( 1 \leq i \leq \ell + 1 \), and
- for some \( p \geq \bar{\ell} \), a \( p \)-length community history \( h_g^p \in \mathcal{H}_{\tau,P} \), induced by \( P \) when realizing \( \tau' \) as in Definition 3.2.3, such that either:

\[- s^p_i \xrightarrow{o^{p + 1}_i} s^{p + 1}_i \text{ in } \mathcal{S}_t \text{ and there is no transition last}(h^p_g) \xrightarrow{o^{p + 1}_C} s^{p + 1}_C \text{ in } \mathcal{S}_C, \text{ or}\]

\[- s^p_i \in S^p_t \text{ and last}(h^p_C) \notin S^p_C. \]

But this means that \( \tau' \) is not realized by \( P \), thus contradicting the hypothesis that \( P \) is a composition.

So, by Theorems 3.4.2 and 3.4.1, we obtain the following, fundamental, result:

**Theorem 3.4.3** Let \( \mathcal{C} \), \( S_t \), \( S_C \) and \( CG \) be as above. \( P \) is a \( CG \)-composition if and only if \( P \) is a composition of \( S_t \) by \( S_C \).

**Proof:** Direct consequence of Theorems 3.4.2 and 3.4.1. \( \square \)

As already anticipated, Theorem 3.4.3 yields that, given a \( CG \), by non-deterministically choosing, at each step, a service among those proposed by the
selection function $\omega_{g'}$, we obtain all and only compositions of $S_I$ by $S_C$. This represents a first improvement to previous work on the Roman Model (see, e.g., [22]), where, by paying the same computational cost, only one composition is computed.

Interestingly, compositions are not required to be built before a client starts interacting with the community, but can be generated just-in-time, as the client issues operation requests. This means that one can select the available service to execute the operation at run-time, when additional, possibly non-predictable, information, such as the operations requested the client in the past, cost of execution, etc., are available, thus providing the orchestrator with great flexibility, as, in a sense, it can “switch” composition on the go as needed. This intuition will be taken, in the following, as the basis for developing ambient-aware compositions that are reactive to events occurring during execution: a novel feature that introduces a further improvement with respect to previous work.
Chapter 4

The Extended Roman Model

In this Chapter, we extend the framework presented in Chapter 3 with two features, obtaining a generalization of the Roman Model. First, we consider nondeterminism in available services, which models partial knowledge about service details and results in (partial) unpredictability of service behavior. Second, we introduce a new component, called data box, intended to model a common, dynamic, structure that services can interact with.

As a motivation for nondeterminism, observe that services essentially model software artifacts that are arbitrarily complex and that may react to unpredictable, or not observable, input sequences. In some cases, not all their details are known or accessible, so that a ground-grain, nondeterministic, description happens to be the best one is able to provide. Consider the following example about a service for taxi reservations.

Example 4.0.2 Typically, taxis observe a first-in-first-out policy, that is: there is a queue of waiting taxis; first incoming request is served by first taxi in queue; after serving a request, a taxi gets back and becomes the last in queue.

Now, assume that the queue hosts two categories of taxis, cheap and expensive. Due to traffic conditions, customer destinations and further real-world events, it is reasonable considering the round-trip time, i.e., the time needed for a taxi to serve a customer request and getting back to the queue, unpredictable. As a result, the state of the queue is unpredictable and, ultimately, so is the category that the first taxi belongs to. If we consider a service that, upon client request, simply offers the first taxi in queue, then it is clearly unpredictable whether a cheap or an expensive taxi will be offered (at least, from client’s perspective).

As for data box, in many cases, services interact with a shared repository or access a common structure, and possibly affect its state, in a way such that their effects become visible to other services. A typical example is a central database, which may also be used by services as a sort of communication channel. Similarly to above example, it is not always the case that details
CHAPTER 4. THE EXTENDED ROMAN MODEL

of such common structure are publicly available or relevant to services. So, in many cases, rather than providing a fully-detailed description of its behavior, one can abstract over actual databox properties thus obtaining, again, a nondeterministic representation. Consider the following example.

Example 4.0.3 Assume a community of services for theatre seat reservations. Each service interacts with a database (i) to check for seat availability, (ii) to record a reservation or (iii) to cancel a reservation. Assume there are, say, 2 seats. The behavior resulting from services-database interactions can be abstracted by the data box represented in Figure 4.1, where each time a community service executes a **reserve** operation, the number of currently available seats decreases by one and each time a **cancel** operation is performed it increases by the same quantity. Clearly, **reserve** can be executed only if there are available seats and **cancel** cannot be executed when the theatre is empty (i.e., there are 2 available seats).

If, for some reason, the actual number of available seats is not available—e.g., because the database owner wants to keep it hidden or not relevant to service purposes, then one can provide an abstract, possibly nondeterministic, description of the same service, such as the one in Figure 4.2. There are 3 states: “em” (empty) which models the situation where no seat is reserved, “av” which models the situation where some seats are available (thought it is not known how many) and “¬av” where no seat is available. As before, a **reserve** operation can be executed only if there are available seats (states em and av) whereas **cancel** can be executed in all states but “em”, where no seat is reserved. Independently on the state it is executed in, **reserve** may lead to a situation where no more (state “¬av”) or some (state “av”) seats are available (recall that the actual number of seats is, in general, unknown). Of course, since the actual number of seats is not public, the actual successor state cannot be foreseen. As for **cancel** operation, we have a specular behavior. Observe that such description general, i.e., it is valid no matter how many (at least one) seats are available, as long as **reserve** and **cancel** yield the same effects (increase or decrease by one) on the current number of available seats. This is why we refer to it as abstract representation.

Finally, it is important pointing out that operations reported in Figure 4.2 are not the only ones allowed by data box. In general, as it will be clear soon, a data box is part of a community. Thus, besides those operations that affect explicitly the DB, there are some that yield no state transitions and some that are not allowed at all in some states. By convention, in order to simplify the pictures, we assume that, if not differently specified, all operations not reported in data box graphical representation are allowed in all DB states and yield a self-loop.

The target service is still assumed deterministic, thus modeling certainty about
client transitions. In other words, each time the client issues an operation request, it knows with certainty the set of operations it may potentially request next, before the requested operation is actually executed. In short, the client knows what it wants. Also nondeterministic target services can be considered, which may correspond to different semantics such as describing loosely specified services (see, e.g., [22]). Here, however, we only deal with deterministic ones.

On the other side, nondeterministic services are only partially controllable, this meaning that each time a service executes an operation, the set of operations it will present next is not, in general, predictable, this depending on the actual transition that takes place. As will be discussed soon, when data box is present, all services, and in particular the target one, contain transitions that are guarded by, i.e., subject to, current data box state. This way, a (target) service can adapt its behavior to current data box state and, in some sense, test conditions over actual state. Recall that data box is, in general, nondeterministic.

The framework proposed here is, in fact, a generalization of the previous one, in the sense that if data box is not present and all services are deterministic then all structures and procedures for checking and synthesizing a composition match exactly those presented in Chapter 3.
4.1 The Framework

In this Section, we detail the above mentioned extensions to the Roman Model, by following the same approach adopted in [24, 54, 129].

**Data box**  As we said, the data box, or DB for short, is a finite state transition system representing the behavior of a common component that services interact with. Even when the actual services (i.e., in the real world) interact with many such components (e.g., two databases), we assume a single data box that represents a suitable description of all their behaviors, seen as a whole. In general, the data box description is finite state, assuming that it provides a reasonable abstraction, i.e., a finite classification capturing all relevant aspects, of the configurations the common components can be in. On the other hand, nondeterminism is introduced to model the possible incomplete information one may have about their behavior, due to possible hidden details. We remark that state finiteness is a key feature, that allows synthesis techniques similar to the ones presented in Chapter 3 to be adopted. In many practical cases, even thought the actual common components happen to be infinite state (think, e.g., of a real database), one can provide a finite state description in terms of a data box, by abstracting over the actual system (see, e.g., [88, 56]) and considering only a finite set of properties. This intuition will be discussed in details in forthcoming chapters, when addressing the data abstraction issues.

Formally,

**Definition 4.1.1** Let $\mathcal{O}$ be the operation alphabet of a service community $\mathcal{C}$, possibly containing nondeterministic available services (see below for a formal definition). A data box for $\mathcal{C}$ is a tuple $DB = (\mathcal{O}, D, d_0, \rho)$ where:

- $\mathcal{O}$ is the finite set of shared operations, i.e., the whole set of operations services can perform, each of which may or may not affect data box state;
- $D$ is the finite set of data box states;
- $d_0 \in D$ is the initial state;
- $\rho \subseteq D \times \mathcal{O} \times D$ is the transition relation among states: $\langle d, o, d' \rangle \in \rho$, or $d \xrightarrow{o} d'$ (is) in $DB$, denotes that execution of operation $o$ in state $d$ may lead the data box to a successor state $d'$.

Essentially the DB is a transition system without final states, where transitions are nondeterministic. Clearly, this requires a transition relation rather than a function. The absence of final states is due to the idea that the DB is not an “active” component, in the sense that it is not able to take the initiative.
and its behavior depends on available services’. So, it is assumed that each
time an available service can stop, possibly after affecting the DB, the DB is
left in a consistent state and its interactions with any services can be stopped
as well. Remarkably, data box state transitions can be triggered only by
community services. In other words, no external events are allowed: all and
only components able to change data box state are part of the community.
This essentially prevents the possibility to model exogenous events that may
yield unexpected transitions. For an example of data box, consider Figure 4.2
and refer to Example 4.0.3 for its description.

Available Services Nondeterministic available services are analogous to
deterministic services in the spirit, but with two main differences. First, for
a given state and the operation to be executed, their evolution is not a priori
known. That is, due to hidden or unspecified details, possibly resulting from
the modeling step, their actual behavior is not fully captured thus possibly
obtaining a nondeterministic specification. However, it is assumed that the
provided description is detailed enough to capture all aspects relevant to the
purposes of service composition. Second, in addition to affecting the DB,
available services are, in turn, affected by its state. In other words, some of
their transitions are allowed to take place, though they are not required to
(due to nondeterminism), only if the DB is in some states. This is formalized
by putting boolean guards—i.e., conditions on current data box state— on
available service transitions. Let see how available services are defined in the
nondeterministic scenario, where a DB is possibly present.

Definition 4.1.2 A nondeterministic guarded available service over a data
box $DB = (O, D, d_0, \rho)$ is a tuple $S = (O, S, s_0, g, G, S_f)$, where:

- $O$ is the same set of operations as in $O$;
- $S$ is the finite set of service states;
- $s_0 \in S$ is the initial state;
- $S_f \subseteq S$ is the set of final states;
- $G$ is a set of boolean functions $g : D \to \{\top, \bot\}$ called guards;
- $\varrho \subseteq S \times G \times O \times S$ is the service transition relation.

When $\langle s, g, o, s' \rangle \in \varrho$, we say that transition $s \xrightarrow{g,o} s'$ (is) in $S$. Given a
state $s \in S$, if there exists a transition $s \xrightarrow{g,o} s'$ in $S$ (for some $g$ and $s'$) and
the data box is in a state $d$ such that $g(d) = \top$ then operation $o$ is said to
be executable in $s$. A transition $s \xrightarrow{g,o} s'$ in $S$ denotes that $s'$ is a possible
successor state of $s$, when operation $o$ is executed in $s$, provided $g(d) = \top$, $d$ being the current data box state. We say that a service $S$ over a data box $DB$ is deterministic iff there is no $DB$ state $d \in D$ for which there exist, in $S$, two distinct transitions $s \xrightarrow{g_1,o} s'$ and $s \xrightarrow{g_2,o} s''$ such that $s' \neq s''$ and $g_1(d) = g_2(d) = \top$. As in Chapter 3, given a deterministic service state and a legal operation in that state, unique next service’s state is always known. In other words, deterministic services are fully controllable by just selecting the operation to perform next.

Observe that guards are more expressive than preconditions, i.e., conditions necessary for an operation to be executed. Indeed, guards are not defined over operations but, rather, over transitions. Consequently, they affect operation execution in a different way in each state and not, as preconditions do, in the same way in all states. Moreover, guards have the power to select which transitions may take place when an operation is executed in a given state. Consider the following example.

**Example 4.1.1** Referring to data box $DB$ of Figure 4.2, described in Example 4.0.3, Figure 4.3(a) depicts an available service over $DB$, $S_1$, that allows to reserve seats for theatre performances. In particular, one can list all available performances, choose one of them, then check for seat availability. If some seats are available, one can first reserve and then either pay, to confirm reservation and buy electronic tickets, or simply cancel the reservation; otherwise, if no seat is available, the service simply gets back to its initial state (possibly informing the client), where the possibility to re-list the available performances or to stop the interaction (initial state is final, as well) is given. As a notational convention, when a transition outgoing from a state $s$ is labeled by a pair of the form $g:o$, this means that when executing operation $o$ in state $s$, if guard $g$ is $\top$ then the transition is allowed, tough not required, to take place. When a guard is omitted in a label, it is implicitly assumed to be $\top$. In general, guards are boolean functions over DB states. In this example, however, we simply replace the only guard $g$ with the state(s) where it assumes value $\top$. So, from state $s_2$, there are two outgoing transitions triggered by operation check: one allowed (only) when $DB$ is in state “av” (i.e., $g(\text{av}) = \top$) and one allowed (only) when $DB$ is in state “¬av” (i.e., $g(\text{¬av}) = \top$). Observe that this is not a nondeterministic transition as $DB$ cannot be in both “av” and “¬av” at the same time and, consequently, not both transitions can be allowed at the same time.

Clearly, data box of Figure 4.2 is affected by $S_1$: each time a reserve or cancel operation is performed, in general, a nondeterministic transition takes place. However, as already pointed out in Example 4.0.3, also other operations, such as list or check, formally affect the DB, though they yield no actual effect, as triggering self-loop transitions only.
4.1. THE FRAMEWORK

(a) Available (guarded) service $S_1$ for theatre seat reservations

(b) Available (nondeterministic) service $S_2$ for taxi reservations.

Figure 4.3: Nondeterministic available services.

As an example of service with nondeterministic operations, consider $S_2$ depicted in Figure 4.3(b). It describes a service for taxi reservation, where client requests are satisfied by picking up the first taxi in queue, according to the behavior described in Example 4.0.2. In particular, when operation `req_taxi` is executed, the service can move to either state $s_1$ or $s_2$, depending on whether a cheap or an expensive taxi, respectively, is available. As already discussed, from service (and hence client) perspective, this property, though observable, is not predictable, so the transition is correctly modeled as a nondeterministic one. When an expensive taxi is available, in addition to operation `pay`, the customer is given the possibility to share the trip with other customers, so to reduce the fare. After executing operation `share`, the service (deterministically) moves to state $s_1$ where, in order to confirm the reservation, the customer is required to pay the same amount as if served by a cheap taxi.

Target Service and Community TS  The notion of (nondeterministic) community is naturally extended, with respect to the one introduced in Chapter 3, so to include data box and nondeterministic services. It is a set $C = \{S_1, \ldots, S_n, DB\}$ containing:

- a data box $DB$;
- $n$ nondeterministic available services $S_1, \ldots, S_n$ over $DB$.

The notion of target service over a data box extends the one of Section 3.1 to deal with data box: a target service has the same form as any other service defined over $DB$, in particular it may include guards, with the only requirement of being deterministic.
Example 4.1.2 Consider the nondeterministic community \( C = \{S_1, S_2, DB\} \) where \( DB \) is as in Figure 4.2 and \( S_i \) \((i = 1, 2)\) are as in Example 4.1.1. Observe that \( S_2 \), in fact, has no (direct) influence over \( DB \). An example of target service defined over data box \( DB \) is shown in Figure 4.4. A client needs to reserve a theatre seat and a taxi to reach the place. Provided taxis are always available the client wants to reserve one, no matter whether cheap or expensive, only after the ticket for the show has been purchased. Operations have the same semantics as in services \( S_1 \) and \( S_2 \). In particular, focus on \( \text{req\_taxi} \) and observe that, for \( S_t \), it is deterministic. Recall that the target service is requested by a client, with the aim to carrying out some interaction. In this case, the client already knows that after operation \( \text{req\_taxi} \), it will ask for executing \( \text{pay\_taxi} \), independently on which state the service that actually executes \( \text{req\_taxi} \) reaches. As we said, we consider only deterministic target services, thus situations where the client has uncertainty about the next operations it may potentially request are, in general, avoided.

We remark that in this example, there is no ambiguity about which service should execute a given operation, as available services \( S_1 \) and \( S_2 \) have two disjoint operation sets. As a result, how services should be composed is quite straightforward. In general, however, further examples with features from this can be mixed with others similar to the ones presented in Chapter 3, so to obtain more complex examples with nondeterministic, guarded, available services whose operation sets intersect.

Also in the nondeterministic context, the (nondeterministic guarded) community \( TS_C \) is introduced as a useful structure that represents the “joint” behavior of available services over \( DB \). Since available services in a community are assumed to act asynchronously\(^1\) on the same data box, combining their

\(^1\)In fact, it is possible to extend the approach and results presented here to the case of several services acting synchronously, as done, e.g., in [130].
evolutions results in a new sort of TS which is, essentially, the asynchronous product of the available services, synchronously combined, in turn, with the data box. In details, it is defined as follows.

**Definition 4.1.3** Let \( C = \{S_1, \ldots, S_n, DB\} \) be a community, where \( S_i = \langle O, S_i, s_{i0}, g_i, G_i, S_i^f \rangle \) \( (i = 1, \ldots, n) \) and \( DB = \langle O, D, d_0, \rho \rangle \). The nondeterministic guarded community TS of \( C \), implicitly defined over \( DB \), is the tuple \( S_C = \langle O, \{1, \ldots, n\}, S_C, s_{C0}, \gamma_C, G_C, S_C^f \rangle \), where:

- \( O \) is, as usual, the shared operation alphabet;
- \( \{1, \ldots, n\} \) is the set of available service indices;
- \( S_C = S_1 \times \cdots \times S_n \) is the (finite) set of \( S_C \) states; let \( s_C = \langle s_1, \ldots, s_n \rangle \), we denote \( s_i \) by \( \text{com}_i(s_C) \) \( (i = 1, \ldots, n) \);
- \( s_{C0} \in S_C \), where \( \text{com}_i(s_{C0}) = s_{i0} \) \( (i = 1, \ldots, n) \) is the \( S_C \) initial state;
- \( S_C^f \subseteq S_C \) is the set of \( S_C \) final states, where \( s_C \in S_C^f \) if and only if \( \text{com}_i(s_C) \in S_i^f \) for all \( i = 1, \ldots, n \);
- \( G = \bigcup_{i=1}^n G_i \) is the set of \( S_C \) guards, where guard \( g_{ij} \) stands for \( g_j \in G_i \) (i.e., \( g_{ij} \) is the \( j \)-th guard of \( G_i \));
- \( \gamma_C \subseteq S_C \times G \times O \times \{1, \ldots, n\} \times S_C \) is the \( S_C \) transition relation, where \( \langle s_C, g, o, k, s_C' \rangle \in \gamma_C \), or \( s_C \xrightarrow{g,o,k} s_C' \) is in \( S_C \), if and only if:
  - \( \text{com}_k(s_C) \xrightarrow{g,o} \text{com}_k(s_C') \) in \( S_k \);
  - \( \text{com}_i(s_C) = \text{com}_i(s_C') \), for \( i = 1, \ldots, n \) such that \( i \neq k \).

Observe that the community TS \( S_C \) has a structure very similar to any other guarded TS, except for the presence of index \( k \) in transitions, representing the available service chosen to actually perform the operation that the transition refers to. We remark that the community TS is an abstract structure well suited to represent the state and the behavior of a service community, but it does not correspond to any actual component. For the sake of brevity, the following example reports only a fragment of the community TS representing the evolution of the community considered in Example 4.1.2

**Example 4.1.3** Figure 4.5 represents a portion of the community TS \( S_C \) obtained from community \( C = \{S_1, S_2, DB\} \) of Example 4.1.2. Dashed edges represent those transitions for which successors or source states have been omitted. Observe how, due to data box nondeterminism, operation cancel introduces a nondeterministic transition outgoing from state \( \langle\langle s_4, s_0\rangle, av\rangle \). Indeed, when the data box is in state \( av \), a cancel operation may leave the DB in \( av \) itself or lead it to state em. Also operation req:taxi introduces nondeterminism, but in this case, differently from cancel, it is due to available service \( S_2 \).
Orchestrator  Also the orchestrator is an extension of the notion presented in Chapter 3. As before, it is a component able to activate, stop and resume all the available services. In addition, it is important to highlight that it has full observability on current available service and data box states, that is, it can keep track, at runtime, of the current state available services and data box are in. Recalling that services are in general nondeterministic, this feature allows, after each operation is executed, to know, with certainty, the actual effects of the operation and, ultimately, the actual history that the community executed up to a given (non-future) instant.

4.2 Nondeterministic Service Composition

In this section, we reformulate the composition problem for the nondeterministic scenario and show a solution that is, again, optimal with respect to worst-case time complexity. Results in this section have been presented in [129] and are extended here with formal details.

We start by defining traces and histories, as already done for the deterministic case. Definitions are very similar to their deterministic counterparts, being essentially a representation of potential service evolutions, but show a major difference: since services, in general, affect and are affected by data box,
4.2. NONDETERMINISTIC SERVICE COMPOSITION

this component needs be taken into account. Indeed, data box state constrains the set of operations a service can execute and, consequently, a trace (history) is valid only if all the operations it contains are coherent with the data box specification. Let us formalize this intuition.

**Definition 4.2.1** Given a nondeterministic service $S = \langle O, S, s_0, g, G, S^f \rangle$ over data box $DB = \langle O, D, d_0, p \rangle$, a trace for $S$ on $DB$ is a possibly infinite sequence $\tau = \langle s^0, d^0 \rangle \xrightarrow{a^1} \langle s^1, d^1 \rangle \xrightarrow{a^2} \ldots$, such that:

- $\langle s^i, d^i \rangle \in S \times D$ for all $i \geq 0$;
- $\langle s^0, d^0 \rangle = \langle s^0_0, d^0_0 \rangle$;
- for all $j > 0$, there exists some $g \in G$ such that $g(d^j) = \top$, $s^j \xrightarrow{g^{a_{j+1}}} s^{j+1}$ is in $S$ and $d^j \xrightarrow{a_{j+1}} d^{j+1}$ is in $DB$.

A nondeterministic service history is any finite prefix of a trace, ending with a pair $\langle s, d \rangle \in S \times D$.

Intuitively, a trace of a nondeterministic available service over a data box represents a possible “joint”, or, better said, synchronous, evolution of the service and the data box. As we mentioned in Chapter 3, given a sequence of (legal) operations for a deterministic available service, one can reconstruct the states it traverses, in fact a whole trace, since each state has only one possible successor under a given operation. In a nondeterministic scenario this is no longer true for two reasons. First, services are, in general, nondeterministic, thus the only sequence of operations does not allow to get a unique trace. Second, even when a (guarded) service is deterministic (in the above sense), as it is the case for the target service, the data box introduces, again, uncertainty about which operations will be available in successor state. As a result, the data box state is an ingredient necessary to make a trace actually representing a possible service evolution, even for deterministic guarded services.

We can now formally define traces and histories for nondeterministic communities.

**Definition 4.2.2** Let $C = \{S_1, \ldots, S_n, DB\}$ be a community, $DB$ as above and $S_C = \langle O, \{1, \ldots, n\}, S_C, s_{C0}, \gamma_C, G_C, S^f_C \rangle$ the respective community TS. A nondeterministic community trace for $S_C$ is a possibly infinite sequence $\tau = \langle s^0_C, d^0 \rangle \xrightarrow{a^1} \langle s^1_C, d^1 \rangle \xrightarrow{a^2} \ldots$, such that:

- $\langle s^i_C, d^i \rangle \in S_C \times D$ for all $i \geq 0$;
- $\langle s^0_C, d^0 \rangle = \langle s_{C0}, d_0 \rangle$;
for all \( j > 0 \), there exists some \( g \in G_C \) such that \( g(d^j) = \top \), \( s_C^{j+1} \) is in \( S_C \) and \( d^j \stackrel{\alpha_{j+1}}{\rightarrow} d^{j+1} \) in \( DB \).

A nondeterministic community history is any finite prefix of a trace, ending with a pair \( (s_C, d) \) in \( S_C \times D \).

Observe how, similarly to available services, a community trace represents a possible synchronous evolution of the community TS—that is, the asynchronous product of available services and the data box. As we said, data box state needs be considered as it constrains the set of operations each service, and hence the whole community, can perform at a given point. The set of all and only (nondeterministic) community histories is referred to as \( H \). Moreover, we refer to the number of alternations between states and operations in service or community histories \( h \) as \( \text{length}(h) \) and to last \( h \)'s state as \( \text{last}(h) \).

**Nondeterministic Service Composition**

Similarly to the deterministic case,

**Definition 4.2.3** Let \( C = \{S_1, \ldots, S_n\} \) be a nondeterministic community of available services, \( \mathcal{H} \) the respective set of all community TS histories and \( \mathcal{O} \) the shared operation alphabet. An orchestrator for \( C \) is a possibly partial function \( P : \mathcal{H} \times \mathcal{O} \rightarrow \{1, \ldots, n\} \) such that, if \( P(h, o) \) is defined, the following holds, where \( k = P(h, o) \) and \( s_C = \text{last}(h) = \langle s_1, \ldots, s_k, \ldots, s_n, d \rangle \):

- there exists a transition \( d \xrightarrow{o} d' \) in \( DB \).
- there exists a guard \( g \in G_C \) and a transition \( s_C \xrightarrow{g \circ o \circ k} s_C' \) in \( S_C \) such that \( g(d) = \top \) and \( s_C' = \langle s_1, \ldots, s'_k, \ldots, s_n, d' \rangle \);

We are now ready to define when an orchestrator is a composition of \( S_t \) by \( S_C \) in a nondeterministic setting.

**Definition 4.2.4** Let \( C = \{S_1, \ldots, S_n, DB\} \) be a community, \( DB \) as above and \( S_C \) the respective community TS. Moreover, let \( S_t \) be a target service over \( DB \) and \( P : \mathcal{H} \times \mathcal{O} \rightarrow \{1, \ldots, n\} \) an orchestrator for \( C \). Given a target service trace \( \tau = \langle s_t^0, d_t^0 \rangle \xrightarrow{o_1} \langle s_t^1, d_t^1 \rangle \xrightarrow{o_2} \ldots \) on \( DB \), \( P \) realizes \( \tau \) if and only if:

- for all community TS histories \( h \in \mathcal{H}_{\tau, P} \subseteq \mathcal{H} \), \( P(h, \text{length}(h) + 1) \) is defined, where \( \mathcal{H}_{\tau, P} = \bigcup_{\tau} \mathcal{H}_{\tau, P} \) is the set of community TS histories induced by \( P \) and \( \tau \), inductively defined as follows:
  - \( \mathcal{H}_{\tau, P}^0 = \{s_C^0, d_0\} \);
  - \( \mathcal{H}_{\tau, P}^{j+1} \) is the set of \((j + 1)\)-length community histories of the form \( h \xrightarrow{\alpha_{j+1}, k_{j+1}} \langle s_C^{j+1}, d^{j+1} \rangle \), where:
4.2. NONDETERMINISTIC SERVICE COMPOSITION

* $h \in \mathcal{H}_{\tau,P}^\ell$, with last$(h) = (s_\ell^j, d^j)$;
* $o^{j+1}$ and $d^{j+1}$ are the same operation and data box state as in $\tau$;
* $P(h, o^{j+1}) = k^{j+1}$, that is, the orchestrator states that operation $o^{j+1}$ in trace $\tau$, after community history $h$, is to be executed by available service $S_{k^{j+1}}$;
* there exists a guard $g \in G_C$ such that $g(d^j) = \top$ and $s_C^j \xrightarrow{g, o^{j+1}, k^{j+1}} s_C^{j+1}$ is in $S_C$, that is, the community TS $S_C$ is able to execute the transition prescribed by $P$.

- for all $(s^\ell_t, d^\ell_t)$ in $\tau$ such that $s^\ell_t \in S^\ell_t$ ($\ell \geq 0$), all $\ell$-length histories $h \in \mathcal{H}_{\tau,P}^\ell$ for which last$(h) = (s_\ell^j, d^j)$ are such that $s^\ell_C \in S^\ell_C$.

As it comes out, the main difference between deterministic and nondeterministic trace realization is due to operation effects. Indeed, here, after executing an operation, many states are, in general, reached by the community TS and consequently a given history is possibly “extended” with several different states (induced by $\tau$ and $P$). On the other hand, in Definition 3.2.3 exactly one community histories is generated at each step, given $\tau$ and $P$. Obviously, dealing with a stateless data box and only deterministic available services would yield all $\mathcal{H}_{\tau,P}^\ell$ to be singletons and, essentially, turn the framework to be the same as in the deterministic scenario.

The definition of trace realization provides, as expected, the basic notion to define when a controller is a composition.

**Definition 4.2.5** An orchestrator $P$ for $C$ is a composition of the target service $S_t$ by the nondeterministic services in $C$ over $DB$ iff it realizes all traces of $S_t$ on $DB$.

As before, the orchestrator realizes a target service if for all target service traces on $DB$, at every step, it returns the index of an available service that can indeed perform the requested operation. Differently from the deterministic scenario, due to available service and data box nondeterminism, the orchestrator must be always able to execute the next operation, no matter how the activated service and the data box happen to evolve after each step. Again, note that the orchestrator can observe available service and data box states (in fact, the whole community service history) in order to decide which available service to select next. So, the characterization of orchestrators as an advanced form of conditional plans ([72]), mentioned in Chapter 3 remains valid also in the general nondeterministic framework.
The Composition Problem  The formal statement of the composition problem in presence of nondeterminism is as follows.

Given a nondeterministic community \( C = \{S_1, \ldots, S_n, DB\} \) and a deterministic target service \( S_t \) over \( DB \), synthesize an orchestrator for \( C \) that is a composition of \( S_t \) by \( S_C \).

The computational complexity characterization of (i) checking for the existence of a composition and (ii) composition synthesis in presence of nondeterministic operations and data box is part of the original contributions of this work and will be discussed in the following.

4.3 Composition Synthesis via ND-Simulation

Here, we present a composition technique able to deal with nondeterministic communities that follows the approach introduced in Chapter 3 (see also [29]) in that it is based on the formal notion of simulation [106, 80]. However, similarities are in the spirit only, as operation nondeterminism prevents the possibility to exploit the off-the-shelf notion of simulation relation and, thus, calls for a an extension of the definition (see, e.g., [129]) and, ultimately, yields a technical construction slightly different from the one proposed for the deterministic scenario.

We start by adapting the notion of simulation relation to deal with nondeterministic community and data box.

Definition 4.3.1 Let \( C = \{S_1, \ldots, S_n, DB\} \) be a nondeterministic community and \( S_t \) a target service over \( DB \). An ND-simulation relation of \( S_t \) by \( S_C \) on \( DB \) is a relation \( R \subseteq S_t \times S_C \times D \) such that \( \langle s_t, s_C, d \rangle \in R \) implies:

1. if \( s_t \in S_t^f \) then \( s_C \in S_C^f \);
2. for each transition \( s_t \stackrel{g, o}{\rightarrow} s_t' \) in \( S_t \) with \( g(d) = \top \), if there exists a transition \( d \stackrel{o}{\rightarrow} d' \) in \( DB \) then there exists a \( k \in \{1, \ldots, n\} \) such that the following holds:
   (a) there exists a transition \( s_C \stackrel{g, o, k}{\rightarrow} s'_C \) in \( S_C \) with \( g(d) = \top \);
   (b) for all transitions \( s_C \stackrel{g, o, k}{\rightarrow} s'_C \) in \( S_C \) with \( g(d) = \top \) and \( d \stackrel{o}{\rightarrow} d'' \) in \( DB \), \( \langle s_t', s'_C, d'' \rangle \in R \).

An ND-simulation is intended to lift the standard notion of simulation relation to deal with operation nondeterminism and the data box. To catch its essence, consider a stateless data box whose only transitions are self-loops on its only state, \( d_0 \), triggered by all operations, and assume \( S_t \) and \( S_C \) have only
One guard identically \( \top \). This basically corresponds to removing the data box, thus obtaining a data box-free ND-simulation. Now, assume that \( (s_t, s_C, d_0) \in R \), for \( R \) ND-simulation relation, and, referring to Definition 4.3.1, focus on the following:

- from requirement (1), analogously to a standard simulation relation, if \( s_t \) is a final state for \( S_t \), then the corresponding (according to \( R \)) community state \( s_C \) is final for \( S_C \);

- from requirement (2a), recalling the nondeterministic community TS Definition 4.1.3, for any operation the target service can execute (when serving a client) in \( s_t \), there exists an available service, namely \( S_k \), that can execute the same operation in state \( \text{com}_k(s_C) \);

- more importantly, from requirement (2b), observing that \( k \) is fixed once \( o \) is fixed (and consequently \( g \), as \( S_t \) is deterministic), \( \forall \) possible \( s_C \) successors, generically referred to as \( s'_C \), are still able to ND-simulate the (only) \( s_t \) successor \( s'_t \); this, in turn, recalling again Definition 4.1.3, corresponds to requiring that no matter how (nondeterministic) service \( S_k \) evolves after executing operation \( o \), the community TS will be able to ND-simulate \( s'_t \).

As it turns out, the ND-simulation relation captures the same idea behind the standard simulation relation and extends it to the nondeterministic case. Indeed, it is not hard to see that, under the same assumptions as above, i.e., absence of data box, if all available services were deterministic, an ND-simulation relation would actually represent a simulation relation, except for the presence of a third, constant, component which brings no useful information and that can be soundly removed.

Now, remove the assumptions over data box and guards and observe that this affects only requirement (2), as (1) involves no transition. As we already discussed, the data box constrains both the available and the target services, in that the possibility to execute an operation no longer depends only on service current state (and capabilities), but also on data box'. Intuitively, the data box can be thought of as a transition system whose states “restrict” service transition relations/functions through service guards, and requirement (2) simply as encoding the ND-simulation property seen above, when the data box was not relevant, on the resulting “restricted” TSs. Since different data boxes yield, in general, different restrictions over the involved services, the data box can be seen as a parameter which sets the restrictions that services are subject to.

**Example 4.3.1** Consider the community \( C = \{S_1, S_2, DB\} \) depicted in Figure 4.6, where both services \( S_1 \) and \( S_2 \) are nondeterministic and \( S_1 \) has a
guarded transition. Figure 4.7 represents the community TS obtained from \( S_1 \) and \( S_2 \). To see an ND-simulation relation \( R \) of \( S_t \) by \( S_C \) on \( DB \), consider Figure 4.8, where each node represents a tuple in \( R \) (and each tuple in \( R \) corresponds to a node in figure). Each node is labeled by a tuple whose components represent (in the order they appear):

- the community TS state;
- the target service state;
- the data box state.

In addition, some edges have been added, each one witnessing the existence of an index \( k \), for each action executable by the target service, as in ND-simulation definition in Definition 4.3.1. In particular, given a node \( t = \langle \langle s_1, s_2 \rangle, s_t, d \rangle \):

- for each operation \( a \) that \( S_t \) can execute in \( s_t \), an index \( k \in \{1, 2\} \) is chosen. Indeed, observe that, for a fixed action, all edges outgoing from a given node are labeled with the same index \( k \);
- a transition with label \( \langle a, k \rangle \) is present, with successor node \( t' = \langle \langle s'_1, s'_2 \rangle, s'_t, d' \rangle \), if and only if all involved components, i.e., community TS, target service and DB, evolve coherently with the transition label.

So, for instance, focus on tuple \( t_0 = \langle \langle s_{10}, s_{20} \rangle, s_{10}, d_0 \rangle \). From \( s_{10} \), the target service can execute only operation \( a \). Now, choose \( k = 1 \) and observe that no matter how the community TS and the data box (nondeterministically) evolve—the former according to label \( \langle a, 1 \rangle \) and the latter according to \( a \), for each of their possible successor states, say \( \langle s'_{10}, s'_{20} \rangle \) for \( S_C \) and \( d' \) for \( DB \), on next step \( t' = \langle \langle s'_{10}, s'_{20} \rangle, s_{11}, d' \rangle \in R \) still holds. Also, observe the role played by guards in node \( \langle \langle s_{10}, s_{20} \rangle, s_{10}, d_1 \rangle \). If no guard were defined on \( S_1 \) transition \( s_{10} \xrightarrow{a, d_0} s_{12} \), then on label \( \langle a, 1 \rangle \) it would have two outgoing edges labeled by \( \langle a, 1 \rangle \), as it is the case for \( t_0 \). But, since the data box is in state \( d_1 \), then transition \( s_{10} \xrightarrow{a, d_0} s_{12} \) in \( S_1 \) cannot take place, while only \( s_{10} \xrightarrow{a, d_0} s_{11} \) does which, in fact, corresponds to the one in Figure 4.8.

**Definition 4.3.2** Given a target service \( S_t \) and a nondeterministic community TS \( S_C \) over data box \( DB \), a state \( s_t \in S_t \) is ND-simulated by \( s_C \in S_C \) on \( d \in D \), denoted \( \preceq (s_t, s_C, d) \), iff there exists an ND-simulation \( R \) of \( S_t \) by \( S_C \) on \( DB \) such that \( (s_t, s_C, d) \in R \). If \( \preceq (s_{00}, s_{C0}, d_0) \) then we say that \( S_t \) is ND-simulated by \( S_C \) on \( DB \) or, equivalently, that \( S_C \) ND-simulates \( S_t \) on \( DB \).

Observe that, like in the deterministic setting, this is a coinductive definition and, thus, the relation \( \preceq \) is itself an ND-simulation, in fact the largest one.
4.3. COMPOSITION SYNTHESIS VIA ND-SIMULATION

Figure 4.6: A nondeterministic service community, data box and target service.

Figure 4.7: Nondeterministic community TS.
Figure 4.8: An ND-simulation with explicit index $k$.

**Computational Characterization** Considering Definition 4.3.1, it is not hard to see that the largest ND-simulation can be computed by Algorithm 1 (NDS). It is enough to show that the algorithm terminates as correctness and completeness come directly by comparing Algorithm 1 to Definition 4.3.1. To this end, observe that at each iteration no element is added to $R$. Since the initialized version of $R$, say $R^0$, is finite (due to finiteness of $S_t$, $S_C$ and $D$) then at most $|R^0| + 1$ steps are sufficient for the algorithm to return. As for time complexity, since $|R^0| = |S_t| \cdot |S_C| \cdot |D|$, we get the following result:

**Theorem 4.3.1** Given a target service $S_t$ and a nondeterministic community $TS S_C$ over data box $DB$, computing the largest ND-simulation relation of $S_t$ by $S_C$ on $DB$ is a PTIME problem.

Now, based on the following two Lemmas, we can formulate and prove the key connection theorem between ND-simulation and composition in the nondeterministic scenario.

**Lemma 4.3.1** Let $C = \{S_1, \ldots, S_n, DB\}$ be a community and $S_t$ a target service over $DB$. If there exists an orchestrator $P$ for $C$ that is a composition of $S_t$ by $S_C$ on $DB$ then $\leq (s_{t0}, s_{C0}, d_0)$.

**Proof:** We need to show that, given $P$, there exists an ND-simulation relation $R \subseteq S_t \times S_C \times D$ such that $(s_{t0}, s_{C0}, d_0) \in R$. For each $S_t$ trace $\tau$, let $\mathcal{H}_\tau, P$ be the
Algorithm 1 \(NDS(S_t, S_C; DB)\) – Computes Largest ND-Simulation of \(S_t\) by \(S_C\) on \(DB\)

1: \(R := S_t \times S_C \times D\)
2: \(R := R \setminus \{(s_t, s_C, d) \mid (s_t \in S_t^f \land s_C \notin S_C^f)\}\)
3: repeat
4: \(R := R \setminus D\), where \(D\) is the set of \((s_t, s_C, d) \in R\) for which there exist
(i) a transition \(d \xrightarrow{o} d'\) in \(DB\) and (ii) a transition \(s_t \xrightarrow{g(o)} s'_t\) in \(S_t\) with
\(g(d) = \top\) such that for each \(k \in \{1, \ldots, n\}\) one of the following holds:
1. there is no transition \(s_C \xrightarrow{g_c, o, k} s'_C\) in \(S_C\) with \(g_C(d) = \top\) (for all
\(g_c \in G_C\));
2. there exists a transition \(s_C \xrightarrow{g_c, o, k} s'_C\) in \(S_C\) with \(g_C(d) = \top\) (for some
\(g_c \in G_C\) but \((s'_t, s'_C, d') \notin R\).
5: until \((D = \emptyset)\)
6: return \(R\)

set of community histories induced by \(\tau\) and \(P\), and \(\mathcal{H}^\ell_{\tau, P}\), the set of its \(\ell\)-length histories, as in Definition 4.2.4. Now, let \((s_t, s_C, d) \in R\) iff there exists an \(S_t\) trace \(\tau = (s_t^0, d^0) \xrightarrow{o^1} (s_t^1, d^1) \xrightarrow{o^2} \ldots \) on \(DB\) and, for some \(\ell\), a community
history \(h^\ell \in \mathcal{H}^\ell_{\tau, P}\) on \(DB\), \(h^\ell = (s_C^0, d_h^0) \xrightarrow{o^1_{t_1}} \ldots \xrightarrow{o^\ell-1_{t_{\ell-1}}} (s_C^\ell, d_h^\ell)\), where each
\(o^i\) is the same as in \(\tau\) (and each \(k^i\) depends on \(P\), as in Definition 4.2.4), such that:

1. \(s_t = s_t^\ell\);
2. \((s_C, d) = (s_C^\ell, d_h^\ell) = \text{last}(h^\ell)\);
3. for all \(0 \leq i \leq \ell\), \(d^i = d_h^i\), where each \(d^i\) is as in \(\tau\).

We show, by induction on the length of \(\mathcal{H}^\ell_{\tau, P}\) histories, that \(R\) is an ND-
simulation relation such that \((s_{t_0}, s_{C_0}, d_0) \in R\).

- Clearly, \((s_{t_0}, s_{C_0}, d_0) \in R\), as \((s_t^0, d^0) = (s_{t_0}, d_0)\) and \(\mathcal{H}^\ell_{\tau, P} = \{h^0\} = \{(s_{C_0}, d_0)\}\) for each \(S_t\) trace \(\tau\) on \(DB\).

- Assume \((s_t, s_C, d) \in R\). By \(R\) definition, there exists an \(S_t\) trace on \(DB\),
\(\tau = (s_t^0, d^0) \xrightarrow{o^1} \ldots \xrightarrow{o^{\ell-1}} s_t^\ell \xrightarrow{o^{\ell+1}} \ldots\) and a corresponding \(\ell\)-length history
\(h^\ell = (s_C^0, d_h^0) \xrightarrow{o^1_{t_1}} \ldots \xrightarrow{o^\ell-1_{t_{\ell-1}}} (s_C^\ell, d_h^\ell)\) such that requirements (1)-(3) above
hold. In particular, \((s_t^\ell, s_C^\ell, d^\ell) = (s_t, s_C, d)\).
Since $P$ is a composition, $s_t^l \in S_t^l \Rightarrow s_t^e \in S_t^e$ comes directly from Definition 4.2.4. Hence $\langle s_t, s_c, d \rangle \in R$ satisfies requirement (1) of Definition 4.3.1.

By definition of nondeterministic service trace, for each $o^\ell + 1 \in O$ and transitions (i) $d^\ell \xrightarrow{o^\ell + 1} d^{\ell + 1}$ in $DB$ and (ii) $s_t^l \xrightarrow{g, o^\ell + 1} s_t^{\ell + 1}$ in $S_t$ such that $g(d^\ell) = T$, there exists an $S_t$ trace of the form $\tau' = \langle s_t^0, d^0 \rangle \xrightarrow{o^1} \ldots \xrightarrow{o^\ell} \langle s_t^{\ell + 1}, d^{\ell + 1} \rangle \ldots$. In particular, all such $\tau'$ are realized by $P$, by definition of composition (observe that $\tau$ and $\tau'$ are two $S_t$ traces, possibly the same, which share the same $\ell$-length prefix history).

Therefore $P(h^\ell, o^\ell + 1)$ is defined. Let $h^{\ell + 1} = h^\ell \xrightarrow{o^\ell + 1} P(h^\ell, o^\ell + 1)$ be the generic $(l + 1)$-length community history induced by $P$ and $\tau'$. Clearly, $(s_t^{\ell + 1}, s_c^{\ell + 1}, d^{\ell + 1}) \in R$, according to $R$ definition above. Moreover, by definition of community history, for each $h^{\ell + 1}$ there exists a transition $s_C^l \xrightarrow{g, o, k^l} s_C^{l + 1}$ such that $g(d^\ell) = T$, where $k^l = P(h^\ell, o^\ell + 1)$.

But then, for each operation $o^\ell + 1$, $k = k^l$ is the service index such that $\langle s_t^l, s_c^l, d^l \rangle = (s_t, s_c, d) \in R$ fulfills requirement (2a) of Definition 4.3.1.

Not only. Since $h^{\ell + 1}$ is generic, for all transitions $s_C^l \xrightarrow{g, o, k^l} s_C^{l + 1}$ such that $g(d^\ell) = T$, $(s_t^{\ell + 1}, s_c^{\ell + 1}, d^{\ell + 1}) \in R$. In other words, requirement (2b) of Definition 4.3.1 is fulfilled and, consequently, so is (2).

(†) and (‡) show that $R$ is an ND-simulation relation.

The viceversa also holds, as shown in the following.

**Lemma 4.3.2** Let $C = \{S_1, \ldots, S_n, DB\}$ be a community and $S_t$ a target service over $DB$. If $\preceq (s_0, s_C, d_0)$ then there exists an orchestrator $P$ for $C$ that is a composition of $S_t$ by $S_C$ on $DB$.

**Proof:** Let $R \subseteq S_t \times S_C \times D$ be an ND-simulation relation such that $(s_0, s_C, d_0) \in R$. We inductively build:

- a set $H_{P,t}$ of sequences of the form $\langle s_t^0, s_C^0, d_0 \rangle \xrightarrow{o^1, k^1} \ldots \xrightarrow{o^\ell, k^\ell} \langle s_t^l, s_C^l, d^l \rangle$, where: $o^i \in O$, $k^i \in \{1, \ldots, n\}$, $s_t^l \in S_t$, $s_C^0 = s_0$, $s_C^l \in S_C$, $s_C^0 = s_C^0$, $d^0 \in D$, $d_0 \in D$ and functions $\text{length}()$ and $\text{last}()$ are as usual;

- an orchestrator $P$ for $C$ that is a composition of $S_t$.

First, for each sequence $h^m = \langle s_t^0, s_C^0, d_0 \rangle \xrightarrow{o^1, k^1} \ldots \xrightarrow{o^m, k^m} \langle s_t^m, s_C^m, d^m \rangle$ define a corresponding sequence $\hat{h}^m = \langle s_C^0, d_0 \rangle \xrightarrow{o^1, k^1} \ldots \xrightarrow{o^m, k^m} \langle s_C^m, d^m \rangle$. Then, inductively define $H_{P,t}$ and $P$ as follows:
4.3. COMPOSITION SYNTHESIS VIA ND-SIMULATION

• (Base case) Let $H^0_{P,t} = \{\langle s_{t0}, s_{C0}, d_0 \rangle \}$. Clearly, by hypothesis $\langle s_{t0}, s_{C0}, d_0 \rangle \in R$;

• (Induction step) Let $H^m_{P,t} \subseteq H_{P,t}$ be a set of $m$-length sequences $h^m$ as above and assume that for all such sequences $\text{last}(h^m) = \langle s^m_t, s^m_C, d^m \rangle \in R$. Then, by definition of ND-simulation relation, for each target transition $s^m_t \xrightarrow{g \circ o} s^{m+1}_t$ with $g(d^m) = \top$, if there exists a transition $d^m \xrightarrow{o} d^{m+1}$ in $DB$ then there exists a $k \in \{1, \ldots, n\}$ as in Definition 4.3.1 (recall that the target service is deterministic). For each transition $s^m_t \xrightarrow{g \circ o} s^{m+1}_t$ as above, choose one of such indices $k$ and let:

- $P(\hat{h}^m, o) = k$;
- $h^m \xrightarrow{o,k} \langle s^{m+1}_t, s^{m+1}_C, d^{m+1} \rangle \in H^{m+1}_{P,t}$ if and only if $\langle s^m_C, d^m \rangle \xrightarrow{g \circ o,k}$ $\langle s^{m+1}_C, d^{m+1} \rangle$ is in $S_C$ for some $g \in G_C$ such that $g(d^m) = \top$. By definition of ND-simulation relation and the choice of $k$, we get that $\langle s^{m+1}_t, s^{m+1}_C, d^{m+1} \rangle \in R$.

The proof is completed by showing that for each $\ell$-length target service history $h^\ell_t$ there exists a corresponding $\ell$-length sequence $\hat{h}^\ell_t$ in $H^{\ell}_{P,t}$ such that (i) $h^\ell_t = \hat{h}^\ell_t$ and (ii) if $\text{last}(h^\ell_t) \in S^\ell_t$ then $\text{last}(\hat{h}^\ell_t) \in S^\ell_t$. Both (i) and (ii) are direct consequence of $H^\ell_{P,t}$ definition and the fact that $R$ is an ND-simulation relation.

So, from Lemmas 4.3.1 and 4.3.2, we get the following result:

**Theorem 4.3.2** Let $C = \{S_1, \ldots, S_n, DB\}$ be a community and $S_t$ a target service over $DB$. An orchestrator $P$ for $C$ that is a composition of $S_t$ by $S_C$ on $DB$ exists if and only if $\langle s_{t0}, s_{C0}, d_0 \rangle \leq R$.

**Proof:** Direct consequence of Lemmas 4.3.1 and 4.3.2.

Based on this, a straightforward method to check for the existence of a composition is obtained:

1. compute the largest ND-simulation relation $\leq$ of $S_t$ by $C$;
2. check whether $\langle s_{t0}, s_{C0}, d_0 \rangle$ is in such relation.

From the computational point of view, the largest ND-simulation relation $\leq$ between $S_t$ and $C$ can be computed in polynomial time wrt the size of $S_t$ and $C$, as Theorem 4.3.1 shows. Since the number of states in $C$ is exponential in the number of available services $n$, $\leq$ can be computed in exponential time. More precisely, it is polynomial wrt the size of $S_t$, $DB$ and each service $S_i$, but exponential in the number of available services $n$. Thus, observing that the
problem is EXPTIME-hard \[108\], we get that this technique is optimal wrt worst-case complexity.

Once the ND-simulation is computed, a composition can be synthesized like already done in the deterministic scenario, again by relying on the composition generator, which is obviously extended in order to take into account nondeterminism and data box.

**Definition 4.3.3** Let \( C = \langle S_1, \ldots, S_n, DB \rangle \) be a community and \( S_t \) a target service over \( DB \) such that \( S_t \) is ND-simulated by \( S_C \). The nondeterministic composition generator (NCG) of \( C \) for \( S_t \) is a tuple \( NCG = \langle O, \{1, \ldots, n\}, S_g, s_{g0}, \omega_g, \delta_g \rangle \), where:

1. \( O \) is the finite set of community operations;
2. \( \{1, \ldots, n\} \) is the set of available service indices;
3. \( S_g = S_t \times S_C \times D \) is the set of NCG states;
4. \( s_{g0} = (s_{t0}, s_{C0}, d_0) \) is the NCG initial state;
5. \( \omega_g : S_g \times O \rightarrow 2^{\{1, \ldots, n\}} \) is the service selection function. For \( s_g = (s_t, s_C, d) \), \( \omega_g(s_g, o) \) is defined if and only if:
   - \( \preceq (s_t, s_C, d) \);
   - there exists a transition \( s_t \xrightarrow{g,o} s_t' \) in \( S_t \), for some guard \( g \) such that \( g(d) = \top \);
   - there exists a transition \( d \xrightarrow{o} d' \) in \( DB \),

in such case:

\[ \omega_g(s_g, o) = \{ k \mid s_C \xrightarrow{g,o,k} s_C' \text{ is in } S_C \text{, for } g \in G_C \text{ such that } g(d) = \top \text{, and } \preceq (s_t', s_C', d') \}; \]

6. \( \delta_g \subseteq S_g \times O \times \{1, \ldots, n\} \times S_g \) is the NCG transition relation, where

\[ \langle s_g, o, k, s_g' \rangle \in \delta_g, \text{ or } s_g \xrightarrow{o,k} s_g' \text{ is in NCG, if and only if } k \in \omega_g(s_g, o) \text{ and } s_g = (s_t, s_C, d) \text{ and } s_g' = (s_t', s_C', d') \] and
   - there exists a transition \( s_t \xrightarrow{g,o} s_t' \) in \( S_t \) with \( g(d) = \top \);
   - there exists a transition \( s_C \xrightarrow{g,o,k} s_C' \) in \( S_C \) with \( g(d) = \top \);
   - there exists a transition \( d \xrightarrow{o} d' \) in \( DB \);

Analogously to its deterministic counterpart, the NCG is a finite state transducer that, given an operation \( o \) (compliant with the target service), outputs, through \( \omega_g \), the set of all available services able to perform \( o \) next,
4.3. COMPOSITION SYNTHESIS VIA ND-SIMULATION

According to the largest ND-simulation relation \( \preceq \). However, in this context we make crucial use of the full observability hypothesis. Indeed, both \( \omega_g \) and \( \delta_g \) depend on the current state of all community services and data box, and none of them can be, in general, reconstructed by just looking at the operation history. As a result, after each operation execution, in order for \( \omega_g \) to return, the current community and data box states are needed to be known. Even thought not considered in this work, a scenario where the current state is only partially observable (i.e., only some of its properties can be known with certainty) is perfectly conceivable, and left open as a future research step. Observe that computing the NCG from \( \preceq \) is easy since it involves checking for local conditions only.

Now, we are ready to introduce the notion of ND generated compositions, i.e., compositions “extracted” from the NCG, and show that a composition is such if and only if it is obtained in this way. To this end, some preliminary definitions and results are needed.

**Definition 4.3.4** Given an NCG for \( S_t \) and \( S_C \), a NCG trace is a possibly infinite sequence \( s_0^g \xrightarrow{o_1, k_1} s_1^g \xrightarrow{o_2, k_2} \ldots \) such that (i) \( s_0^g = s_g^0 \) and (ii) for each \( i \geq 0 \), \( s_i^g \xrightarrow{o_{i+1}, k_{i+1}} s_{i+1}^g \) in NCG. An NCG history is any finite prefix of a trace, ending with a state. We define \( H_{ng} \) as the set of all NCG histories.

From Definitions 4.3.3 and 4.3.4, we immediately obtain the following.

**Lemma 4.3.3** If \( s_0^g \xrightarrow{o_1, k_1} s_1^g \xrightarrow{o_2, k_2} \ldots \) is an NCG trace then, for all \( s_i^g = \langle s_{i, t}^C, s_{i, C}^t, d_i^0 \rangle \) (\( i \geq 0 \)), \( \preceq (s_{i, t}^C, d_i^0) \).

In the same way as in the deterministic scenario, \( S_t \) and \( S_C \) traces (histories) can be derived from NCG traces (histories), by just projecting out one (i.e., the non-relevant) component from NCG states. Formally:

**Definition 4.3.5** Given a NCG trace \( \tau_g = s_0^g \xrightarrow{o_1, k_1} s_1^g \xrightarrow{o_2, k_2} \ldots \), where \( s_i^g = \langle s_{i, t}^C, s_{i, C}^t, d_i^0 \rangle \),

- the NCG-derived community trace \( \tau_C \) is the sequence \( \tau_C = \langle s_0^0, d_0^0 \rangle \xrightarrow{o_1} \langle s_1^C, d_1^0 \rangle \xrightarrow{o_2} \ldots \), obtained by projecting out from each \( \tau_g \) state \( s_i^g \) the \( S_t \) component;

- the NCG-derived target service trace \( \tau_t \) is the sequence \( \tau_t = \langle s_0^0, d_0^0 \rangle \xrightarrow{o_1} \langle s_1^t, d_1^0 \rangle \xrightarrow{o_2} \ldots \), obtained by projecting out from each \( \tau_g \) state \( s_i^g \) the \( S_C \) component and by removing the index \( k_i \) from each transition;

- an NCG-derived community (target service) history is any finite prefix of an NCG-derived community (target service) trace, ending with a state.
It is easily seen, by NCG definition, that

**Lemma 4.3.4** Each NCG-derived community (target service, respectively) trace is a community (target service) trace, and analogously for histories.

We have also the converse for target service traces and histories.

**Lemma 4.3.5** For each target service trace $\tau_t$ (history $h_t$, respectively) there exists a NCG trace $\tau_g$ (history $h_g$, respectively) such that $\tau_t$ ($h_t$) is NCG-derived from $\tau_g$ ($h_g$).

**Proof:** It is a consequence of $\delta_g$ and $\omega_g$ definitions in Definition 4.3.3, which yield that if $s_g \xrightarrow{o,k} s'_g$ in NCG then $\preceq (s_t, s_C, d)$ and $\preceq (s'_t, s'_C, d')$, where $s_g = (s_t, s_C, d)$ and $s'_g = (s'_t, s'_C, d')$, and the fact that $S_t$ is ND-simulated by $S_C$ (otherwise NCG would be not defined), i.e., $\preceq (s_{t0}, s_{C0}, d_0)$.

That is, the whole (and exact) set of target service traces corresponds to the set obtained by NCG-deriving all NCG traces.

We can now define how a composition can be generated from the NCG.

**Definition 4.3.6** Let $C$, $DB$, $S_t$ and $S_C$ be as above. Assume $S_t$ is ND-simulated by $S_C$ on $DB$ and let NCG be the corresponding nondeterministic composition generator. An ND generated composition is an orchestrator $P : H \times O \rightarrow \{1, \ldots, n\}$ satisfying the following, inductively defined, requirement:

- let $s_g^0 \in H_P$, where $H_P \subseteq H_{ng}$ and $s_g^0 = (s_{t0}, s_{C0}, d_0)$;
- if $h \in H_P$ then for all $o \in O$ such that $\omega_g(last(h), o)$ is defined:
  - $P(h_C, o) = k \in \omega_g(last(h), o)$, where $h_C$ is the community history NCG-derived from $h$;
  - for all transitions $last(h) \xrightarrow{o,k} s_g^{length(h)+1}$ in NCG, $h \xrightarrow{o,k} s_g^{length(h)+1} \in H_P$.

ND generated compositions are obtained in a way very similar to the deterministic case, that is, by picking up, at each step, one among the service indices returned by $\omega_g$, when NCG follows the actual history the community executed up to the current state while realizing a target service history. However, since after each operation execution many community and data box successor states are possible, and the actual one is not, obviously, known in advance, a composition is required to take into account all possibilities. Precisely, it requires that if $P$ is defined over a given $\ell$-length community history $h_C$ and operation $o$, then it is defined over all possible $(\ell + 1)$-length community histories obtained by executing $o$ after $h_C$ and over all operations the target service can
execute next. Observe that all mentioned community histories are, in fact, NCG-derived from NCG-histories contained in \( \mathcal{H}_P \).

We can now relate ND generated compositions to compositions, through the following characterizing theorems.

**Theorem 4.3.3** Let \( \mathcal{C} \), \( \mathcal{DB} \), \( \mathcal{S}_t \) and \( \mathcal{S}_C \) be as above. Assume \( \mathcal{S}_t \) is ND-simulated by \( \mathcal{S}_C \) on \( \mathcal{DB} \) and consider the corresponding NCG. If \( P \) is an ND generated composition, then \( P \) is a composition of \( \mathcal{S}_t \) by \( \mathcal{S}_C \) on \( \mathcal{DB} \).

**Proof:** Let \( P \) be a (ND) generated composition as in Definition 4.3.6 and consider a generic infinite (the finite case is a straightforward specialization) target service trace on \( \mathcal{DB} \) \( \tau = (s^0_0, d^0)^\tau (s^1_1, d^1)^\tau \ldots \) We show that \( P \) realizes \( \tau \), i.e., \( (i) \) for all \( \ell \)-length community histories \( h^\ell \in \mathcal{H}_{\tau,P} \), where \( \mathcal{H}_{\tau,P} \) is as in Definition 4.2.4, \( P(h^\ell, o^{\ell+1}) \) is defined and \( (ii) \) for all \( (s^\ell_t, d^\ell) \) in \( \tau \) such that \( s^\ell_t \in S^\ell_t \), all histories \( h^\ell \in \mathcal{H}_{\tau,P} \) for which \( \text{last}(h^\ell) = (s^\ell_C, d^\ell) \) are such that \( s^\ell_C \in S^\ell_C \).

For \( (i) \), we proceed by induction on the length of histories in \( \mathcal{H}_{\tau,P} \) and \( \mathcal{H}_P \) (as in Definition 4.3.6):

- Let \( h^0 = (s^0_C, d^0) \in \mathcal{H}_{\tau,P} \) be the community history NCG-derived from \( h^0_g = s^0_g = (s^0_t, s^0_C, d^0) \in \mathcal{H}_P \). By hypothesis, \( s^0_t \in S^0_t \). Moreover, by \( \tau \) definition above, there exist two transitions \( (i) s^0_t \overset{g}{\rightarrow} s^1_t \in \mathcal{S}_t \) such that \( g(d_0) = \top \) and \( (ii) d_0 \overset{o^1}{\rightarrow} d_1 \) in \( \mathcal{DB} \) for some \( d_1 \) as in \( \tau \). So, by Definition 4.3.3, \( \omega_g(\text{last}(h^0_g), o^1) \) is defined and, by Definition 4.3.6, so is \( P(h^0_g, o^1) \);

- Assume \( P(h^\ell, o^{\ell+1}) \) is defined for all \( 0 \leq \ell \leq m \), where each \( h^\ell \in \mathcal{H}_{\tau,P} \) is NCG-derived from the \( \ell \)-length NCG history \( h^\ell_g \in \mathcal{H}_P \), according to Definition 4.3.6, and each \( o^{\ell+1} \) is as in \( \tau \). By Definition 4.3.6, \( \omega_g(\text{last}(h^{m-1}_g), o^m) \) is defined as well and, hence, \( s^{m-1}_t, s^{m-1}_C, d^{m-1} \) for \( \text{last}(h^{m-1}_g) = (s^{m-1}_t, s^{m-1}_C, d^{m-1}) \). Moreover, by \( \tau \) definition above, there exist two transitions \( (i) s^{m-1}_t \overset{o^m}{\rightarrow} s^m_t \in \mathcal{S}_t \) such that \( g(d_m) = \top \) and \( (ii) d^{m-1} \overset{o^m}{\rightarrow} d^m \in \mathcal{DB} \) for some \( d^m \) as in \( \tau \). Let \( P(h^{m-1}_g, o^m) \) be a (ND) generated composition as in Definition 4.3.6 and \( (i) s^m_t \overset{o^m}{\rightarrow} s^m_t \in \mathcal{S}_t \) such that \( g^m(d^{m-1}) = \top \) and \( (ii) d^{m-1} \overset{o^m}{\rightarrow} d^m \) in \( \mathcal{DB} \) for \( d^m \) as in \( \tau \). Let \( P(h^{m-1}_g, o^m) \) be a (ND) generated composition as in Definition 4.3.6 and \( (i) s^m_t \overset{o^m}{\rightarrow} s^m_t \in \mathcal{S}_t \) such that \( g^m(d^{m-1}) = \top \) and \( (ii) d^{m-1} \overset{o^m}{\rightarrow} d^m \) in \( \mathcal{DB} \) for \( d^m \) as in \( \tau \). We get that \( \omega(\text{last}(h^m_g), o^{m+1}) \) is defined and, therefore, so is \( P(h^m_g, o^{m+1}) \).
For (ii), it is enough observing that for all histories \( h^\ell_g \in \mathcal{H}_P \), \( \preceq (s^\ell_t, s^\ell_C, d^\ell) \), where \( \text{last}(h^\ell_g) = (s^\ell_t, s^\ell_C, d^\ell) \).

Analogously to the deterministic case, we get that the viceversa also holds.

**Theorem 4.3.4** Let \( \mathcal{C}, DB, S_t \) and \( S_C \) be as above. If \( P \) is a composition of \( S_t \) by \( S_C \) on \( DB \) then it is an ND generated composition.

**Proof:** Let \( \tau = (s^0_t, d^0) \xrightarrow{o_1} (s^1_t, d^1) \xrightarrow{o_2} \ldots \) be an infinite target trace (the finite case is a straightforward specialization). From this, consider the target history \( h^\ell_0 \) obtained as \( \ell \)-length prefix of \( \tau \). Given \( P, \tau \) and \( h^\ell_0 \), consider an \( \ell \)-length community history \( h^\ell \in \mathcal{H}_{\tau,P} \) as in Definition 4.2.4. Clearly, for each \( h^\ell_0 \), at least one (but, in general, several) history \( h^\ell \) exists as, by hypothesis, \( P \) is a composition.

Referring to Definition 4.3.6, it is enough, to our purposes, showing that for all target service traces \( \tau \), if \( P(h^\ell, o^{\ell+1}) \) is defined then

\[
P(h^\ell, o^{\ell+1}) = k^{\ell+1} \in \omega_g(\text{last}(h^\ell), o^{\ell+1}),
\]

where: \( h^\ell_0 = s^0_t \xrightarrow{o_1} s^1_t \xrightarrow{o_2} k^2 t \ldots \), \( s^\ell_t = (s^\ell_t, s^\ell_C), s^\ell_C = \text{last}(h^\ell) \) and \( s^\ell_t = \text{last}(h^\ell_0) \) are the same as in \( \tau \).

By contradiction, suppose \( \tau \) does not satisfy (†) and let \( \ell \) be the minimum index such that \( P(h^\ell, o^{\ell+1}) \) is defined but (†) \( P(h^{\bar{\ell}}, o^{\bar{\ell}+1}) = k^{\bar{\ell}+1} \not\in \omega_g(\text{last}(h^{\bar{\ell}}), o^{\bar{\ell}+1}) \). Thus, considering \( \omega_g \) definition in Definition 4.3.3 and observing that, by \( \tau \), there exists a transition \( s^\ell_t \xrightarrow{g,o^{\ell+1}} s^{\bar{\ell}+1}_t \) in \( S_t \), for some \( g \in G_t \), such that \( g(d^{\bar{\ell}}) = \top \) with \( d^{\bar{\ell}}, o^{\bar{\ell}+1}, s^\ell_t \) and \( s^{\bar{\ell}+1}_t \) as in \( \tau \), we necessarily get that either (i) \( \omega_g(\text{last}(h^{\bar{\ell}}), o^{\bar{\ell}+1}) \) is not defined because \( \not\in (s^\ell_t, s^\ell_C, d^\ell) \), or (ii) \( \omega_g(\text{last}(h^{\bar{\ell}}), o^{\bar{\ell}+1}) \) is defined but \( \not\in (s^{\bar{\ell}+1}_t, s^{\bar{\ell}+1}_C, d^{\bar{\ell}+1}) \), where \( h^{\bar{\ell}+1} = h^{\bar{\ell}} \xrightarrow{o^{\bar{\ell}+1}, k^{\bar{\ell}+1}} s^{\bar{\ell}+1}_y \) and \( s^{\bar{\ell}+1}_y = (s^{\bar{\ell}+1}_t, s^{\bar{\ell}+1}_C, d^{\bar{\ell}+1}) \) (otherwise (†) would hold for \( \ell = \bar{\ell} \)). So \( \not\in (s^\ell_t, s^\ell_C, d^\ell) \lor \not\in (s^{\bar{\ell}+1}_t, s^{\bar{\ell}+1}_C, d^{\bar{\ell}+1}) \) holds. But, recalling the formal notion of simulation relation in Definition 4.3.1, due to finiteness of state spaces, this yields that there must exist:

- a target service trace (possibly \( \tau \)) \( \tau' = (s^0_t, d^0) \xrightarrow{o_1} \ldots \xrightarrow{o_{\bar{\ell}+1}} (s^{\bar{\ell}+1}_t, d^{\bar{\ell}+1}) \xrightarrow{o_{\bar{\ell}+2}} \ldots \), where \( s^0_t, d^0, o^i, d^i \) and \( s^i \) are the same as in \( \tau \), for \( 1 \leq i \leq \bar{\ell} + 1 \), and

- for some \( p \geq \bar{\ell} \), a \( p \)-length community history \( h^p \in \mathcal{H}_{\tau,P} \), induced by \( P \) when realizing \( \tau' \) as in Definition 4.2.4, with \( \text{last}(h^p) = (s^p_t, s^p_C, d^p) \), such that either:

\[ s^p_t \notin S^C_t \text{ and } \text{last}(h^p) \notin S^C_t, \text{ or} \]
4.4 DISCUSSION

- there exist two transitions (i) \( s'_t \xrightarrow{g,o} s'_{t+1} \) in \( S_t \) (for some \( g \in G_t \))
  such that \( g(d_t) = \top \) and (ii) \( d'_t \xrightarrow{o'_{t+1}} d'_{t+1} \) in \( DB \), but there is
  no transition \( last(h'_t) \xrightarrow{g,o'}_{t+1} k'_{t+1} \xrightarrow{r} s'_{C,t+1} \) in \( S_C \) with \( g(d'_t) = \top \) for
  \( g \in G_C \), where \( k'_{t+1} \) is defined by \( P \).

But this means that there exists a target service trace \( \tau' \) that is not realized
by \( P \), thus contradicting the hypothesis that \( P \) is a composition. \( \square \)

Finally, previous Theorems 4.3.4 and 4.3.3 yield the following basic result
for composition of nondeterministic services operating on a data box.

**Theorem 4.3.5** Let \( C, DB, S_t \) and \( S_C \) be as above. An orchestrator \( P \) of \( S_C \)
for \( S_t \) on \( DB \) is a composition of \( S_t \) by \( S_C \) on \( DB \) if and only if it is an ND
generated composition.

We close this Chapter by observing that, similarly to what pointed out in
Chapter 3, also in the nondeterministic scenario compositions can be generated
just-in-time, by relying on the NC\( G \) and exploiting the available service and
data box state observability. Next chapter is intended to show in detail the
advantages one gets when computing simulation-based compositions.

### 4.4 Discussion

In the Roman Model, and its extended version, an *extensional* state space
characterization is provided, in the sense that all service states are explicitly
listed. This may yield modeling difficulties, in particular when services are
modeled in an actual language. For instance, assume that, in a service, an
operation \( o \) can be executed, say, in states \( s_1, \ldots, s_n \) and that each execution
nondeterministically leads the service to states \( s_{n+1}, \ldots, s_{n+m} \). In this case,
one needs to explicit list out \( n \times m \) transitions. Clearly, the mathematical
abstraction aimed at capturing web service evolution, but is not very well-
suited for actual description. An alternative would be representing states by
means of their (propositional) properties, such as internal variables (ranging
over finite domains), and then defining operations based on *pre-* and *post-*
conditions, expressed as conditions over such properties. For example, one
can easily state that operation *reduce (i)* is executable only if the value of
an internal variable ranging over \( \{0, \ldots, 100\} \) is less than 50, and *(ii)* that its
effect is to make the value of the same variable less than 50. This is a very
convenient representation, commonly used in AI planning domains. From an
abstract point of view, both descriptions are equivalent, as long as the latter
yields a finite number of states. Notice that, this way, one can easily refer to
sets of states by expressing conditions over properties, e.g., \( v \leq 20 \) refers to all service states whose variable \( v \) is assigned a value less than 20.

This observation is useful to introduce partially observable services. In a partially observable service, the current state cannot be always observed or, in general, the result of an observation is uncertain. As an example, consider a service whose state is described by two boolean variables, say \( v_1 \) and \( v_2 \), which contains, of course, four states, and assume that only \( v_1 \) is fully observable, whereas \( v_2 \) is not accessible at all. This means that, at any point, one can only know whether \( v_1 = \top \) or \( v_1 = \bot \), that is, only a possible set of states that the service is in is knowable. This issue has a major impact on the possibility of composing web services when dealing with nondeterministic available services. Indeed, like we said, the current community state is, in general, strictly needed by the orchestrator generator, in order to make its decision. To see this, consider the service depicted in Figure 4.9. After executing nondeterministic operation \( op_1 \) on initial state, if \( v_2 \)'s value is not observable, then the orchestrator generator cannot know whether the service moved or remained still in the initial state and, consequently, can neither known whether the service is actually able to execute operation \( op_2 \), and hence can be considered for operation delegation, or not.

In actual systems, partial observability arises very naturally, as, in general, available services do not export explicitly their state, and, hence, orchestrator generators can only infer it indirectly, e.g., through received messages. Service composition under partial observability is currently object of ongoing work. The main idea is to group states that are equivalent under a same observations and consider only all operations allowed by all such states which, therefore, are considered as certain.
Chapter 5

Composition Robustness

In this chapter, we face the problem of building robust compositions in the general, nondeterministic, scenario. Assume a composition has been built that realizes some target service. Essentially, it coordinates the set of (partially controllable) available services by delegating to one of them, at each step, the operation requested by a client that behaves as if conversing with the target service. Once the composition is built, it is expected to work correctly, provided, of course, available services do, too. In other words, the main assumption upon which the synthesis techniques are built is that available services respect a contract, i.e., (i) they work as specified by their respective transition system and (ii) are always available to be delegated for operation execution (according to their specification and current state).

Now, we relax this assumption and assume that some services can, for external reasons, unexpectedly stop their activity, either temporarily or permanently. When this happens, one can adopt a naive strategy and trivially re-compute a new composition by simply ignoring the missing service(s). This is required, for instance, when a PDL-based technique such as those proposed in [27, 23], is used. Indeed, with such technique, one composition only is found by a black-box procedure, which amounts to (i) encoding the problem as a PDL-SAT formula and (ii) extracting the (only) composition from the obtained formula model, if any. Importantly, there are no additional information one can store while searching for the model, a fact that essentially prevents the possibility to modify, even slightly, the obtained composition so to react to unexpected events. Therefore, if any of the available services needed to compose the target service is no longer available, a new composition needs to be computed from scratch. Clearly, re-planning is not, in general, an optimal strategy. In addition, recall that a (composed) target service is intended to conversing with a client. If the interaction is interrupted at some point (because, e.g., an available service is no longer available) and then restarted by
relying on a different composition, it is supposed to continue the interaction from the same point it was left, as client behavior is based on target service specification.

Here, we show how the simulation-based approach opens the possibility to construct compositions that are reactive to unexpected events, in a smart way, that is, by avoiding, when possible, full re-computation of compositions.

5.1 The Framework

The framework we refer to here is the same as in Chapter 4. We assume a nondeterministic community \( C \) formed by a data box \( DB \) and several nondeterministic available services \( S_1, \ldots, S_n \) over it, plus a target service \( S_t \) which describes the desired service to be implemented. To show how the SOC paradigm applies to different applicative scenarios, we show an example taken from the AI world ([129]), where a set of robotic appliances, called behaviors and here formalized as available services, act on a common environment, represented as data box. The goal of the application is to automatically generate a controller, in fact an orchestrator, which coordinates the appliances so to realize a desired behavior, that is, the target service.

**Example 5.1.1** Figure 5.1 depicts a scenario\(^1\) where some robotic arms are designed to process, i.e., cleaning and painting, some blocks. Due to design constraints, only one block at a time can be processed and, to this end, blocks require first to be prepared. Moreover, processing needs resources, water and paint, which are stored into two respective tanks, simultaneously rechargeable by pressing a button.

There are three arms, \( ARM_1, ARM_2 \) and \( ARM_3 \), whose behaviors are modeled by as many available services, respectively \( S_1, S_2 \) and \( S_3 \) depicted in Figures 5.1(b), 5.1(c) and 5.1(d). \( ARM_1 \), a cleaning-disposing arm, is able to clean and dispose blocks, as \( S_1 \) shows. \( ARM_2 \) is capable of preparing, cleaning, and painting blocks and \( ARM_3 \) is a paint arm, which can also prepare blocks for processing. All three arms are able to press the recharge button and, thus, refill tanks. Notice that arm \( S_2 \) behaves nondeterministically when it comes to painting a block. This nondeterminism shows the incomplete information one has about \( ARM_2 \)'s internal logic. Observe also the requirement of \( ARM_1 \) for the environment to be in a state \( (d_0 \) or \( d_1) \) where water is available (see below) so as to be able to perform a clean operation.

A nondeterministic environment, modeled by data box \( DB \) of Figure 5.1(a), provides the general rules of the domain. For instance, blocks can be painted

\(^1\)See [54] for a basic version of this example and [129] for another version of the same scenario as proposed here.
or cleaned only after they have been prepared. Indeed, in this example, we assume that only explicitly reported operations are allowed in each data box state. Consequently, in, e.g., state $d_0$, operation clean is not allowed, whereas recharge and prepare are—in fact, they are the only ones that can be executed in this state. Also, the DB includes some information about the water tank used for block cleaning: in states $d_0$ and $d_1$, the water tank is not empty, whereas in states $d_2$ and $d_3$ it does. It is worth noticing the nondeterminism, from DB’s perspective, of operation paint: since neither the quantity of water contained in the tank nor that needed to perform a clean operation are explicitly modeled, it is not known, with certainty, when the tank becomes empty. As a consequence, executing clean when the DB is in state $d_1$ may either leave DB in the same state (water tank not empty) or yield a transition to state $d_2$ (water tank empty). Observe that in $d_2$ a clean operation is still allowed, this being possible thanks to some method not relying on water.

The desired, non existing, arm $\text{Arm}_t$, to be realized by suitably composing available arms, is modeled by the (deterministic) target service $S_t$ of Figure 5.1(e), whose goal is to repeatedly carrying out a block painting process. In state $t_1$, cleaning is optional because only dirty blocks need to be washed before painting, and such information (clean or dirty) can be observed only by the client. In practice, a (possibly human) client first requests to prepare a block, then evaluates its conditions and, if dirty, decides to clean it before painting. If the block is already clean, it is painted with no additional operations. After painting, independently of its original condition, the block is disposed back in its initial position and, eventually, the recharging button is pressed, so to guarantee that tanks are not empty and a new block can be processed.

5.2 Just-in-Time Orchestrators

In Chapters 3 and 4, we anticipated an important advantage one gets when a simulation-based technique is adopted, that is, the possibility to delay the actual choice of the service that an operation is delegated to. By Theorem 4.3.5, from a high-level perspective, one can think of NCGs (see Definition 4.3.3) as “meta-plans”, or nondeterministic “complete universal plans”, that is, structures which keep all the existing plans at their disposal and decide which one to adopt for the next operation, possibly subject to contingent information. Indeed, while realizing some target trace, the NCG follows the actual execution of the community. Rather than pre-defining some criterion for choosing the service—among those returned by $\omega_g$—that current operation is delegated to, one can decide in a step-by-step fashion, so to make decisions based on current situation and possibly taking into account information not modeled by NCG states—for instance, assigning an operation to some service may yield
CHAPTER 5. COMPOSITION ROBUSTNESS

Figure 5.1: Community and target service of Example 5.1.1.
some additional costs that depends on unpredictable factors one knows only at run-time.

Typically, given an NCG, a \textit{just-in-time} composition is generated as follows:

1. initially, the NCG is in its initial state (and so are all services, included target, and data box);

2. the client issues an operation request, according to target service initial state;

3. based on current NCG state \( s_g \) and client request \( o \), NCG output function \( \omega_g(s_g, o) \) returns the set of available services able to execute \( o \) in current state; at this point, any among the returned available services can be chosen and, in particular, one can base the choice on some additional information (e.g., costs) not explicitly modeled in the framework;

4. after the choice is made, the operation is delegated to the chosen service;

5. the selected available service executes the operation; if some failure arises, we assume to know it before delegation takes place;

6. the NCG moves to next state, according to its transition relation and \textit{after} the actual state of the available service that executed \( o \) has been observed;

7. at the same time, the client moves to next state and issues a new request, that can be served by target service;

8. a new iteration starts from 3.

\textbf{Example 5.2.1} Figure 5.2 depicts a fragment of the NCG for the community and target service of Example 5.1.1, where nodes represent states of the NCG and full edges represent regular NCG transitions. In addition, incoming dashed edges represent the existence of a path from the initial to the destination state, whereas outgoing dashed edges stand for the existence of some outgoing NCG transitions. They are reported to make it clear that this fragment is part of a wider structure (i.e., the NCG). Nodes are labeled by tuples \( (s_t,(s_1,s_2,s_3),d) \), where, according to NCG definition, \( s_t \) is a target service state, \( (s_1,s_2,s_3) \) represents a community state and \( d \) stands for data box state. Transition labels have the form: \( o,i/I \), where (i) \( o \) is an operation allowed by the target service state \( s_t \) appearing in the source state label, (ii) \( i \) represents the index of an available service chosen among those \( \omega_g \) returns, given the source (NCG) state and the labeling operation and (iii) \( I \) is the set returned by \( \omega_g \). Obviously,
CHAPTER 5. COMPOSITION ROBUSTNESS

all edges sharing a same labeling operation \( o \) and outgoing from a same node, also share the same \( I \).

Focus on non final, i.e., non double-lined, states and observe that for each of them, when operation recharge is requested, two available services can be selected: either \( S_1 \) or \( S_3 \). Since no requirement is given to make the decision beforehand, one can wait until a state is actually reached (recall that community evolution is nondeterministic) to evaluate which choice is the best, according to some criterion.

We assume the existence of a choice function \( \text{choose}(\cdot) \) that, for each NCG history \( H_g \) and operation \( o \) requested by the client according to target service capabilities, chooses one element among those returned by \( \omega_g(\text{last}(h_g), o) \). By Theorem 4.3.5, if all such choice functions are considered then all possible compositions are, as well. Observe that such assumption corresponds, in a sense, to be able to switch composition when needed. Just-in-time orchestrators are referred to as \( P_{jit} \) and their distinctive property is that, at each step, the available service selection is based upon the \( \text{choose}(\cdot) \) function.

Figure 5.2: Fragment of the NCG for Example 5.1.1
5.3 Service Failures

There are many situations and domains in which assuming full reliability of components is not adequate. For example, in multi-agent complex and highly dynamic domains, one cannot rely on the total availability nor on the reliability of all the existing modules—that are modeled as available services. Indeed, there are a variety of reasons why modules may stop being available at some point or another: devices may break down, agents may decide to stop cooperating, communication with agents may drop, exogenous events may change the state of the environment, and so on. On the other hand, services may possibly re-appear into the system at a later stage, thus creating new “opportunities” for the overall system.

As mentioned before, services’ and data box’ specifications—intended to model some actual device, robot or the environment they interact with—can be seen as contracts, and failures, as the ones above, as breaches of such contracts. We identify five core ways of breaking contracts, which correspond to the abstraction of actual situations that real devices can be in. It is important highlighting that we assume to be able to distinguishing among the different kinds of failures. In other words, we are reasoning at a logical level of abstraction, where implementation details are not considered. For instance, when saying that a service dies we are assuming that this information is somehow obtained, e.g., as result of an explicit communication where the service informed the orchestrator that it will not participate to any other composition, or the conclusion of a reasoning process undertaken by the orchestrator which, due to repeated communication failures, concludes that the service is no longer active nor will be such in the future. Contract breaches we deal with are as follows:

(a) A service temporarily freezes, that is, it stops responding and remains still, then eventually resumes in the same state it was in. As a result, while frozen, the orchestrator cannot delegate operations to it.

(b) A service unexpectedly and arbitrarily (i.e., without respecting its transition relation) changes its current state. The orchestrator can in principle keep delegating operations to it, but it must take into account the service’s new state.

(c) The data box unexpectedly and arbitrarily (i.e., without respecting its transition relation) changes its current state. The orchestrator has to take into account that this affects both the target and the available services.

(d) A service dies, that is, it becomes permanently unavailable. The orchestrator has to completely stop delegating operations to it.
(e) A service that was assumed dead unexpectedly resumes operation, starting in a certain state. The orchestrator can exploit the new opportunities it creates and start delegating operations to it again.

The composition techniques in [54, 130] do not address the above cases, since they assume that orchestrators always deal with fully reliable services. As a consequence, upon any of the above failures, we are only left with the option of “re-planning” from scratch for a whole new composition.

As discussed above, simulation-based techniques offer a way of solving the composition problem (i.e., synthesizing compositions) that is intrinsically more robust. Roughly speaking, this alternative approach deals with unexpected failures by suitably refining the solution at hand, either on-the-fly (for cases (a), (b), and (c)) or parsimoniously (for cases (d) and (e)), thus avoiding full re-planning.

5.4 Reactive Adaptability

In this Section, we show that Theorem 4.3.5 provides a sound and complete technique for dealing with failure cases (a), (b) and (c) above, without requiring any re-computation. Such ability is referred to as reactive adaptability, so to highlight that no further computation is needed, as the NCG, already available, is sufficient to handle this situations.

Service freezing Just-in-time orchestrators are able to deal with temporary service freezing, corresponding to above failure case (a). Indeed, when this happens, in order to come up with a new correct composition, it is enough for the orchestrator –in fact, for function choose(·)– to avoid selecting the frozen service and simply delegating operations to another available service, if any. For instance, if service $S_1$ of Example 5.2.1 freezes, say, in state $\langle t_4, s_{10}, s_{22}, s_{30}, d_0 \rangle$, operation recharge can be delegated to $S_3$. Clearly, if no other choices are possible, then $P_{jit}$ is required to wait for the service to come back$^2$.

State change of services and/or data box $P_{jit}$ can also address unexpected changes in the internal state of services and/or data box, which correspond to failure cases (b) and (c)$^3$. To understand this, consider a parametric version of target, community and data box transition systems, $S^C(t)$, $S^C(s)$ and $DB(d)$, for $t \in S_t$, $s \in S^C$ and $d \in D$, which are the same as usual $S_t$, $S^C$

$^2$Recall that we assume to know with certainty that a service is temporarily unavailable. Consequently, we also know that the composition will eventually become executable again.

$^3$Although hardly as meaningful as the ones above, unforeseen changes in target state can be accounted for in a similar way.
and $\mathcal{DB}$ except that initial states are replaced by the respective arguments, $t$, $s$ and $d$. Now, assume that (i) an NCG has been computed and, at a given point while running a composition, $\mathcal{S}_c$, $\mathcal{DB}$ and $\mathcal{S}_t$ are, respectively, in states $s_c$, $d$ and $s_t$, obviously such that $\preceq (s_t, s_c, d)$—therefore, NCG is in state $(s_t, s_c, d)$—and that (ii) $\mathcal{S}_c$ and/or $\mathcal{DB}$ states, $s_c$ and $d$, change unexpectedly to states $\hat{s}_c$ and $\hat{d}$.

In principle, in order to keep executing the current composition, a new composition needs to be computed, by applying any technique, considering $\mathcal{S}_t(t)$, $\mathcal{S}_c(\hat{s})$ and $\mathcal{DB}(\hat{d})$ as, respectively, the target, community and data box transition systems. Clearly, such a composition exists if and only if, by Theorem 4.3.5, $\preceq (s_t, \hat{s}_c, \hat{d})$. Observe, though, that an ND-simulation relation is already available (as an NCG was assumed) and that it is, in fact, the largest one, as required by NCG definition. In addition, note that ND-simulation relations are independent of target, community and data box initial states—indeed, in Definition 4.3.1, they are never mentioned—this essentially meaning that given $\mathcal{S}_t(t)$, $\mathcal{S}_c(s)$ and $\mathcal{DB}(d)$, the largest ND-simulation relation $\preceq$ remains the same, no matter which states are actually taken as initial. Consequently, the ND-simulation that was computed to build the NCG (i.e., where $t = s_{t0}$, $s = s_{c0}$ and $d = d_0$) is the same one would re-compute after the states unexpectedly changed. In turn, one would get the same NCG already available, except for the initial state. This implies, ultimately, that such generator (essentially, the same as before) can be used (and the same just-in-time controller $P_{jit}$ as well), with the guarantee that all compositions of the (new) community for the (new) target, if any, are still captured. Put it all together, we only need to check whether $\preceq (t, \hat{s}_c, \hat{d})$, and, if so, continue to use $P_{jit}$ (now starting from the zero-length NCG history: $(t, \hat{s}_c, \hat{d})$).

**Example 5.4.1** Consider Figure 5.2 and suppose that, at a given point, while a conversation between a client and a composed target service is in progress, $\mathcal{S}_t$ is in state $t_4$, $\mathcal{S}_1$ in $s_{t10}$, $\mathcal{S}_3$ in $s_{t30}$ while $\mathcal{S}_2$ and $\mathcal{DB}$ unexpectedly change their state, respectively, from $s_{22}$ to $s_{20}$ and from $d_3$ to $d_0$ (e.g., tanks have been recharged without pressing the recharge button). All that is needed in such case is to check whether $\preceq (t_4, s_{t10}, s_{20}, d_0)$. Since they are (as they appear in an NCG state), the NCG can safely continue the realization of the target service from such (new) configuration.

### 5.5 Parsimonious Refinement

When considering failure cases (d) and (e), a simple reactive approach is not sufficient. In case (d), if some service stops working permanently, the orchestrator can no longer rely on the fact that the service will eventually resume.
Observe how this is different from case (a): indeed, when a service temporarily freezes, one can still keep delegating operations, though this has a cost in terms of execution delay; on the other hand, when service unavailability is permanent, operations can no longer be delegated to the service and it must be considered as if not part of the community. Therefore, all compositions relying on the missing service have to be discarded or, in fact, removed from the NCG. As for case (e), notice that it includes the situation when a new available service is added to the community. Indeed, one can think that any non-community service is, in fact, dead. Clearly, when a new service becomes available, NCG re-computation becomes necessary, as there are some additional possibilities not taken into account.

The challenge we face in this Section is to avoid NCG full re-computation. We will show that there exist smart, incremental, techniques that allow to simply refine the available NCG and reuse all the information about composition, previously computed.

Starting from algorithm NDS (i.e., Algorithm 1 of Chapter 4), we define its parametric version, referred to as NDSP, with two extra input parameters:

- $R_{\text{init}} \subseteq S_t \times S_C \times D$, the starting relation from which the largest ND-simulation is extracted;
- $R_{\text{sure}} \subseteq S_t \times S_C \times D$, a relation containing tuples already known to be in the ND-simulation to be computed.

**Algorithm 2 NDSP($S_t, S_C, DB, R_{\text{init}}, R_{\text{sure}}$)**

1. $R := R_{\text{init}} \setminus R_{\text{sure}}$
2. repeat
3. $R := (R \setminus C)$, where $C$ is the set of tuples $(s_t, s_C, d) \in R$ for which there exists an $o \in O$ such that for each $k \in \{1, \ldots, n\}$ there exist two transitions, $s_t \xrightarrow{g_C,o,k} s'_t$ in $S_t$ and $d \xrightarrow{o} d'$ in $DB$ (for some $g_t \in G_t$, $s'_t \in S_t$ and $d' \in D$), such that one of the following holds:
   1. there is no transition $s_C \xrightarrow{g_C,d} s'_C$ in $S_C$ with $g_C(d) = \top$ (for all $g_C \in G_C$);
   2. there exists a transition $s_C \xrightarrow{g_C,o,k} s'_C$ in $S_C$ with $g_C(d) = \top$ (for some $g_C \in G_C$) but $(s'_t, s'_C, d') \notin R \cup R_{\text{sure}}$.
4. until ($C = \emptyset$)
5. return $R \cup R_{\text{sure}}$

As it turns out, the algorithm is essentially a flexible version of Algorithm 1 with the following features:
the initial set that elements are discarded from can be provided as input \((R_{\text{init}})\);

- a set of elements to be included without any additional check can be provided as input \((R_{\text{sure}})\).

Next result shows that output of algorithm NDSP coincides with that of NDS—
that is, it computes the largest ND-simulation of \(S_t\) by \(S_C\) over \(DB\),—provided the additional parameters are used adequately.

**Lemma 5.5.1** Let \(C\) be a community and \(S_t\) a target behavior. If \(R_{\text{sure}} \subseteq NDS(S_t, S_C, DB) \subseteq R_{\text{init}}, \) then \(NDSP(S_t, S_C, DB, R_{\text{init}}, R_{\text{sure}}) = NDS(S_t, S_C, DB)\).

**Proof:** Let \(R^i\) and \(R^i_p\) be the sets representing \(R\) in algorithms NDS and NDSP, respectively, after \(i\) repeat-loop iterations. Assume that NDS and NDSP require, respectively, \(N\) and \(N_p\) repeat-loop iterations (clearly, \(N_p \leq N\)).

First, we prove, by induction on \(i\), that \(R^i_p \cup R_{\text{sure}} \subseteq R^i\).

For the base step, it is obvious that \(R^0_p \cup R_{\text{sure}} \subseteq R^0\), as \(R_{\text{init}} \subseteq S_t \times S_C \times D\). Suppose now, as induction hypothesis, that \(R^r_p \cup R_{\text{sure}} \subseteq R^r\), with \(r < N_p\). Moreover, assume that \(\tau = (s_t, s_C, d) \in R^r_p \cup R_{\text{sure}}\), but \(\tau \notin R^{r+1}\). Since neither \(R^r\)'s nor \(R^r_p\)'s are ever expanded along iterations, it is the case that \(\tau \notin R_{\text{sure}}\) (otherwise \(\tau \in NDS(S_t, S_C, DB)\) and \(\tau \in R^{r+1}\), as \(NDS(S_t, S_C, DB) \subseteq R^{r+1}\)), and thus \(\tau \in R^r_p\), \(\tau \in R^r_p\), and \(\tau \in R^r\). Since \(\tau \notin R^{r+1}\), \(\tau\) was deleted in \(r\)-th NDS loop iteration. This means that there exists an \(\hat{o} \in O\) such that for each \(k \in \{1, \ldots, n\}\) there exist two transitions, \(s_t \xrightarrow{g, \hat{o}} s'_t\) in \(S_t\) and \(d \xrightarrow{\hat{o}} d'\) in \(DB\) (for some \(g_t \in G_t, s'_t \in S_t\) and \(d' \in D\)), such that either 1. or 2. of NDS’ step 4. holds. In particular, if case 2. applies then for each \(k\) there exists a tuple \(\tau^*_k = (s'_t, s'_C, d')\) such that \(s'_C \xrightarrow{g_C, \hat{\omega}, k} s'_C\) in \(S_C\) and \(d \xrightarrow{\hat{\omega}} d'\) in \(DB\), but \(\tau^*_k \notin R^r\). Therefore, by induction hypothesis, \(\tau^*_k \notin R^r_p \cup R_{\text{sure}}\). Thus, such same operation \(\hat{o}\), together with the corresponding tuple \(\tau^*_k\), does indeed satisfy the NDSP requirement of step 4., so that \(\tau \notin R^{r+1}_p\). Contradiction.

Also if case 1. applies, obviously, \(\tau \notin R^{r+1}_p\). Again, contradiction. Hence, \(R^{r+1}_p \cup R_{\text{sure}} \subseteq R^{r+1}\) and \(NDSP(S_t, S_C, DB, R_{\text{init}}, R_{\text{sure}}) \subseteq NDS(S_t, S_C)\) follows.

Next, we show that \(NDS(S_t, S_C, DB) \subseteq NDS(S_t, S_C, DB, R_{\text{init}}, R_{\text{sure}})\). To this end, we prove, by induction on \(i\), that \(NDS(S_t, S_C, DB) \subseteq R^i_p \cup R_{\text{sure}}\). For the base case, since \(NDS(S_t, S_C, DB) \subseteq R_{\text{init}}\), \(NDS(S_t, S_C, DB) \subseteq R^0_p \cup R_{\text{sure}}\). As induction step, suppose that \(NDS(S_t, S_C, DB) \subseteq R^r_p \cup R_{\text{sure}}\) for some \(r < N_p\) and let \(\tau = (s_t, s_C, d) \in NDS(S_t, S_C, DB)\) but \(\tau \notin R^{r+1}_p \cup R_{\text{sure}}\). By induction hypothesis, \(\tau \in R^r_p \cup R_{\text{sure}}\), so \(\tau\) was removed from \(R_p\) at \(r\)-th NDSP iteration. But then, similarly to above, there exists an \(\hat{o} \in O\) such that
for each \( k \in \{1, \ldots, n\} \) there exist two transitions, \( s_i \xrightarrow{g_i, \hat{o}} s'_i \) in \( S_t \) and \( d \xrightarrow{\hat{o}} d' \) in \( DB \) (for some \( g_i \in G_t, s'_i \in S_t \) and \( d' \in D \)), such that either 1. or 2. of NDSP's step 3. holds. If case 2. applies then for each \( k \) there exists a tuple \( \tau_k' = (s'_i, s'_g, d') \) such that \( s'_g \xrightarrow{g_c, \hat{o}} s'_C \) in \( S_C \) and \( d \xrightarrow{\hat{o}} d' \) in \( DB \), but \( \tau_k' \notin R'_P \cup R_{sure} \). Therefore, by induction hypothesis, \( \tau_k' \notin NDS(S_t, S_C, DB) \) and consequently \( \tau_k' \notin R^N \). However, since \( \tau \in NDS(S_t, S_C, DB) \), also \( \tau \in R^N \). But, then by using the same operation \( \hat{o} \), together with the corresponding \( \tau_k' \notin R^N \) tuples, \( \tau \) is a candidate to be removed from set \( R^N \). Thus, algorithm ND requires more than \( N \). Contradiction. Also if 1. holds, a similar argument applies and, again, a contradiction is obtained. Hence, \( \tau \in R'_P \cup R_{sure} \) and \( NDS(S_t, S_C) \subseteq NDSP(S_t, S_C, DB, R_{init}, R_{sure}) \) follows. \( \square \)

Let us introduce some notational conventions, to shrink and expand communities and ND-simulation relations, that will be useful in the remainder of this chapter.

First, we focus on community expansion, that is we analyze the situation when a set of (supposedly) unavailable services resumes operation. Consider a community \( \mathcal{C} = \langle S_1, \ldots, S_n, DB \rangle \) and a set of available service indexes \( W \subseteq \{1, \ldots, n\} \). We denote by \( \mathcal{C}(W) \) the (sub)community derived from \( \mathcal{C} \) by considering only (i.e., projecting on) all services \( S_i \) such that \( i \in W \). Clearly, \( \mathcal{C} = \mathcal{C}(\{1, \ldots, n\}) \). Given the target service \( S_t \) over \( DB \), we denote by \( \preceq^W \) the largest ND-simulation relation of \( S_t \) by \( S_{\mathcal{C}(W)} \) on \( DB \), where \( S_{\mathcal{C}(W)} \) represents the community transition system obtained from community \( \mathcal{C}(W) \). Let \( U \subseteq \{1, \ldots, n\} \) be such that \( W \cap U = \emptyset \). We denote by \( \preceq^W \otimes U \) the relation obtained from \( \preceq^W \) by (trivially) “putting” all services \( S_i \), such that \( i \in U \), back into the system. Formally, we can define operator \( \otimes \) as follows, assuming, without loss of generality, that \( W = \{1, \ldots, \ell\} \) and \( U = \{\ell + 1, \ldots, m\} \):

**Definition 5.5.1**

\[
\preceq^W \otimes U \triangleq \\{(s_t, s'_d) \mid s' = (s_1, \ldots, s_{\ell}, s_{\ell+1}, \ldots, s_m), \text{where:} \\
\langle s_t, (s_1, \ldots, s_\ell), d \rangle \in \preceq^W, \\
s_i \in S_i, i \in \{\ell + 1, \ldots, m\}, \\
s_i \in S'_i, i \in \{\ell + 1, \ldots, m\}\}.
\]

Intuitively, this operation corresponds to expanding \( \preceq^W \) tuples with further components. In fact, they bring no additional information, with respect to original tuples, apart from “announcing” the existence of additional services. An important property of \( \otimes \) is that it preserves ND-simulation relations. In other words, when a set of services is “put back” into the community through \( \otimes \), an ND-simulation for the obtained community \( S(W \cup U) \), in general not the largest one, is obtained. The following lemma shows this claim.
Lemma 5.5.2 Let \( C = \{ S_1, \ldots, S_n, DB \} \) and \( \preceq_W \) be as above and \( S_t \) a target service over \( DB \). If \( W, U \subseteq \{ 1, \ldots, n \} \) are such that \( W \cap U = \emptyset \) then:

- \( \preceq_W \otimes U \subseteq \preceq_{W \cup U} \);
- \( \preceq_W \otimes U \) is an ND-simulation relation of \( S_t \) by \( S_C(W \cup U) \) over \( DB \).

Proof: Without loss of generality, take \( W = \{ 1, \ldots, \ell \} \), and \( U = \{ \ell + 1, \ldots, m \} \). Suppose that \( \langle s_t, (s_1, \ldots, s_\ell, s_{\ell+1}, \ldots, s_m), d \rangle \in \preceq_W \otimes U \).

Due to operator \( \otimes \) definition (Definition 5.5.1), it is the case that \( \preceq_W (\langle s_t, (s_1, \ldots, s_\ell) \rangle, d) \). Hence:

- if \( s_t \) is final in \( S_t \) then so are all \( s_i \) in \( S_i \), for \( i \in \{ 1, \ldots, \ell \} \) and, of course, for \( i \in \{ \ell + 1, m \} \);
- for each \( o \in O \), there exists an index \( k_o \in W \) as in ND-simulation relation requirement 2. (see Definition 4.3.1) for target service \( S_t \) and community \( C(W) \). Thus, \( \preceq_{W \cup U} (\langle s_t, (s_1, \ldots, s_\ell, s_{\ell+1}, \ldots, s_m), d \rangle) \), as for each \( o \in O \), the same index \( k_o \) would also satisfy the ND-simulation requirements for \( S_t \) and community \( C(W \cup U) \) – i.e., newly introduced services do not reduce others’ capabilities.

This shows that \( \preceq_W \otimes U \) is an ND-simulation relation of \( S_t \) by \( S_C(W \cup U) \) and, hence, \( \preceq_W \otimes U \subseteq \preceq_{W \cup U} \), as \( \preceq_{W \cup U} \) is the largest ND-simulation of \( S_t \) by \( S_C(W \cup U) \) on \( DB \).

Next, we consider community shrinking. Let \( F \subseteq W \) be the indices of some services that, at a certain point, become permanently unavailable. We denote by \( \preceq_W \mid_F \) the relation obtained from \( \preceq_W \) by projecting out all (failed) service components \( s_i \) such that \( i \in F \).

Definition 5.5.2 Let \( W = \{ 1, \ldots, \ell \} \) and assume, without loss of generality that \( F = \{ s_p, \ldots, s_\ell \} \subseteq W \).

\[ \preceq_W \mid_F \triangleq \{ \langle s_t, s', d \rangle \mid s' = \langle s_1, \ldots, s_{p-1} \rangle, \exists s_p, \ldots, s_\ell \text{ s.t. } \langle s_t, (s_1, \ldots, s_\ell), d \rangle \in \preceq_W \}. \]

In this case, the obtained result is no longer guaranteed to be an ND-simulation relation. However, interestingly, we have that the new largest ND-simulation relation of \( S_t \) by \( S_C(W \setminus F) \) on \( DB \), namely \( \preceq_W \mid_F \) is contained in \( \preceq_W \mid_F \). Formally:

Lemma 5.5.3 Let \( C = \{ S_1, \ldots, S_n, DB \} \) and \( \preceq_W \) be as above and \( S_t \) a target service over \( DB \). If \( W, F \subseteq \{ 1, \ldots, n \} \) are such that \( F \subseteq W \) then \( \preceq_W \mid_F \subseteq \preceq_W \mid_F \).
Proof: By Lemma 5.5.2, \( \preceq_{(W \setminus F) \otimes F} \subseteq \preceq_{(W \setminus F) \cup F} \), that is, \( \preceq_{(W \setminus F) \otimes F} \subseteq \preceq_{W \setminus F} \). By projecting \( F \) out of both relations, we obviously get \( \preceq_{(W \setminus F) \otimes F} \subseteq \preceq_{W \setminus F} \). Then, since \( \preceq X \mid X = \preceq \) for any \( \preceq \) and index set \( X \), \( \preceq_{(W \setminus F)} \subseteq \preceq_{W \setminus F} \) follows. \( \square \)

Notice that despite \( \preceq_{W} \) being the largest ND-simulation relation when services in \( W \) are “active”, the projected relation \( \preceq_{W \setminus F} \) is not necessarily even an ND-simulation relation of \( S_t \) by \( S_C(W \setminus F) \) on \( DB \). Indeed, it is not hard to find cases where containment \( (\preceq_{W \setminus F} \subseteq \preceq_{W \setminus F}) \) is proper.

**Permanent unavailability** Obviously, when a service becomes permanently unavailable (i.e., case (d)), waiting for it to resume, if the composition really needs it, is not a feasible solution. Instead, one can either continue executing the composition and just “hope for the best”, i.e., that the failed service will never be necessary, or “refine” the current composition to continue guaranteeing, if possible, the full realization of the target. To adopt the latter approach, one needs to compute a new NCG which takes into account no “failed” services. Since, as we saw, in order to construct the NCG, one needs the largest simulation relation of the target service by the community of available (and, above all, active) services, what we really need is to refining (instead of recomputing) the ND-simulation relation at hand before the failures took place. By Lemma 5.5.3, one can execute the NDSP algorithm providing as input the relation obtained by merely projecting out the failed components. This generally yields a substantial reduction of algorithm iterations. Indeed, as services become unavailable, the effort to obtain the new largest ND-simulation relation is **systematic** and **incremental** in that no tuples that were previously discarded will be considered.

**Theorem 5.5.1** Let \( C = \langle S_1, \ldots, S_n, DB \rangle \) and \( S_t \) be as above. If \( W \subseteq \{1, \ldots, n\} \) is the set of indices of \( C \)'s currently working services and \( F \subseteq \{1, \ldots, n\}, F \subseteq W \), is the set of indices of \( C \)'s permanently unavailable services then, for every \( \beta \) such that \( \beta \subseteq \preceq_{W \setminus F} \):

\[
\preceq_{(W \setminus F)} \subseteq NDSP(S_t, S_C(W \setminus F), DB, \preceq_{W \setminus F}, \beta).
\]

**Proof:** Direct consequence of Lemmas 5.5.1 and 5.5.3. \( \square \)

**Example 5.5.1** Consider orchestrator \( P_1 \) depicted in Figure 5.3(a). Numbers, along with operations, that label transitions identify the available service that the operation is delegated to. For instance, in transition \( c_0 \xrightarrow{\text{prepare},2} c_1 \), operation prepare is delegated to available service \( S_2 \). Labels of transitions outgoing from state \( c_5 \) represent conditions. Indeed, it can be seen that, while executing the orchestrator –which is, in fact, a composition– each time state
c_3 is reached, service S_2 can be either in state s_{20} or s_{22} (whereas S_1 is in s_{10}). Each label reports the state that S_2 must be in, for the orchestrator to trigger the correct delegation. Simply said, when the orchestrator is in state c_3, if S_2 is in s_{20} then operation recharge is delegated to S_1, otherwise, i.e., S_2 is in s_{22}, it is delegated to S_2. Clearly, the reported conditions are mutually exclusive and cover all possible cases.

Now, assume that, while P_1 is in execution, service S_2 unexpectedly breaks down. In particular, assume that this happens right after orchestrator transition \( c_2 \xrightarrow{\text{dispose,1}} c_3 \) takes place and S_2 is in state s_{22} (it can be shown that data box can be either in d_0 or in d_3). Clearly, S_t is no longer realized by P_1. However, a different orchestrator, P_2, depicted in Figure 5.3(b), exists that is able to realize S_t. P_1 is effective provided (i) the data box is either in state d_0 or d_3, (ii) S_1 is in s_{10} and (iii) S_3 is in s_{30}. It can be seen that this conditions are fulfilled each time the orchestrator is in state c_3.

As we said, in order to compute P_2, computing a new simulation from scratch, then the NCG and, finally, a composition, is not required. Rather, one can first project out S_2 from (available) ND-simulation relation \( \preceq_{\{1,2,3\}} \), thus getting \( \preceq_{\{1,2,3\}} \left| \{2\} \right. \) and then, starting from this, compute the ND-simulation relation by exploiting algorithm NDSP. The obtained ND-simulation relation is depicted in Figure 5.4, where transitions, labeled with operation and respective available service index, that are relevant to S_t are also reported. Observe that when S_2 fails, the whole system can be in any state with a transition incoming from “nothing”. As usual, final states are double-line shaped.
Figure 5.4: ND-simulation relation for Example 5.5.1, after $S_2$ failure.
5.5. PARSIMONIOUS REFINEMENT

Resumed behaviors  Consider now the case where while services with indexes in $W$ are currently operating, some services that are supposed to be permanently unavailable, unexpectedly become available again (cf. case (e)). Let the indexes of such services be $U$, with $U \cap W = \emptyset$. Obviously, this could never reduce the capabilities of the whole system, but, possibly, could enhance it with more choices. To exploit such choices, one needs to compute the new largest ND-simulation $\preceq_{(W \cup U)}$. In doing so, one can leverage on the fact that $\preceq_{(W \cup U)}$ contains the relation $\preceq_W \otimes U$ (cf. Lemma 5.5.2) by completely avoiding consideration (for potential filtering) of those tuples in $\preceq_W \otimes U$, that is, we pass those tuples as the “sure set” to the NDSP algorithm. Similarly to previous case –i.e., permanent service unavailability–, this may reduce the computational effort needed for computing the new ND-simulation relation as tuples in $R_{\text{sure}}$ are never checked for filtering.

Theorem 5.5.2 Let $\mathcal{C} = \{S_1, \ldots, S_n, DB\}$ and $S_i$ be as above. If $W \subseteq \{1, \ldots, n\}$ is the set of indices of $\mathcal{C}$’s currently working services and $U \subseteq \{1, \ldots, n\}$, with $W \cap U = \emptyset$, is the set of indices of unexpectedly resumed services then, for every $\alpha$ such that $\preceq_{(W \cup U)} \subseteq \alpha$:

$$\preceq_{(W \cup U)} = \text{NDSP}(S_i, S_{\mathcal{C}(W \cup U)}, \alpha, \preceq_W \otimes U).$$

Proof: Direct consequence of Lemmas 5.5.1 and 5.5.2. □

Observe that even if $U$ included new services not (originally) included in $\{1, \ldots, n\}$ Lemma 5.5.2 would still hold.

Reusing previous computed ND-simulation relation  So far, we investigated over reacting to failures from a “local” point of view. That is, we considered how to react to failures that may take place in a given, present, situation, no matter whether, in the past, there have been other failures, possibly handled, or any services unexpectedly resumed, so that previous computations can be exploited to save computation time. Now, we focus on this latter case, and explore the possibility of recording and reusing ND-simulation relations computed (to handle unexpected situations as above) in the past, with the aim of reducing the computation time of ND-simulation relations needed in new unexpected situations.

Suppose that some sets of services failed or resumed in the past. Each of such events will be associated to a set $W_i$ of active services and to a respective ND-simulation relation $\preceq_{W_i}$, computed with the aim of reacting to the unexpected situation, as seen above. Let $\mathcal{W}$ be the set of all $W_i$ such that $\preceq_{W_i}$ has been computed in the past, assume that $\preceq_{W_i}$ has been recorded for each $W_i \in \mathcal{W}$ and let $W = W_0 = \{1, \ldots, n\} \in \mathcal{W}$.  

Let us define:

$$\bar{\alpha} = \bigcap \{W' \in \mathcal{W} | W' \subseteq W \setminus W\};$$

$$\bar{\beta} = \bigcup \{W' \in \mathcal{W} | W' \supseteq (W \setminus W')\};$$

where $\mathcal{W}$ and $\mathcal{W}$ stand for the set of tightest supersets and subsets, respectively, of $W$ in $\mathcal{W}$, namely:

$$\mathcal{W} = \{W' \in \mathcal{W} | W' \subseteq W \setminus V \subseteq W \setminus W' \setminus W' \supseteq V \setminus W' \setminus V\};$$

$$\mathcal{W} = \{W' \in \mathcal{W} | \forall V \in \mathcal{W}, W' \subseteq W \setminus V \subseteq W \setminus W' \setminus V\}.$$ 

Then, by using the above Theorems 5.5.1 and 5.5.2 we get that:

$$\preceq_W = NDSP(T_t, T_S(W), \bar{\alpha}, \bar{\beta}).$$

Notice that by using $NDSP(T_t, T_S, \bar{\alpha}, \bar{\beta})$ to compute $\preceq_W$, we maximally reuse the computations already done to devise other ND-simulations. Of course, once we have computed $\preceq_W$, we can immediately compute $P_{jit}$ on-the-fly, as before.
Chapter 6

Services with Multiple Operations

So far, we based our work on the assumptions that (i) only one operation at a time can be requested for execution, i.e., target service transitions are labeled by exactly one operation and (ii) when realizing a target transition, only one available service can be delegated to execute the corresponding operation. That is, we considered only *interleaved* execution of concurrent operations and dealt with available services able to execute one single operation at a time.

In this chapter, such restrictions are removed, thus a new variant of the composition problem is obtained and an extension of previous solution techniques becomes necessary. In details, starting from the framework for nondeterministic available services presented in Chapter 4, we extend all services, i.e., available and target, and data box so that their transitions are triggered by sets of operations, instead of one only, and *multiple* delegation is allowed, i.e., target service transitions can be realized by delegating, in general, the target labeling operations to a set of available services. This results in a generalization of the basic framework, in that if we consider only singleton sets of operations, and restrict to single delegation only, we obtain exactly the same framework and results as in Chapter 4.

When an available service executes *multiple* operations, the effects that such execution yields on the service itself and on the data box (i.e., the transitions it triggers) may be very different from those that one would obtain by executing each operation alone. In particular, for instance, an operation might be nondeterministic when executed alone but deterministic when executed in conjunction with other ones. Clearly, such feature allows to express the mutual influence that operations possibly play. As an example, think of two arms, each able to lift the side of a table on top of which an object lies. If
only one arm lifts the table then the object may or may not fall, uncertainty depending, say, on the actual shape of the object or the surface material of the table which are not explicitly modeled, but if both arms are activated at the same time, i.e., both lift operations are executed in parallel, then the object is guaranteed to not fall.

When dealing with multiple operations, a problem that naturally arises concerns concurrency. As we said, so far, we only considered interleaved execution of atomic operations. Now, we switch to a scenario where synchronized execution of atomic operations is also permitted. Inspired by [47], we refer to synchronized operations as nonempty sets of atomic operations performed together by various services. Coherently with our approach, we look at services as high-level abstractions of existing devices/programs and, consequently, we assume that when a service executes a set of operations it also takes care, transparently, from client’s viewpoint, of their correct synchronization. That is, if a service offers the possibility to execute multiple operations, it also guarantees a “correct” execution, so that the obtained effects are those the service and/or data box\footnote{As we will see, execution of multiple operations yields similar consequences on data box.} specify. Such assumption is kept valid also for the target service: we assume that when an orchestrator realizes a target service that allows to execute multiple operations, it “dispatches” all requested operations to the available services and takes care of their synchronization.

### 6.1 The Framework

As usual, we start by introducing our theoretical framework, taken and re-adapted from [130], which is, in turn, an extension of [54]. What follows is the natural extension of all structures presented in Chapter 4, modified to deal with multiple operation executions.

As a notational convention, given the usual operation alphabet $\mathcal{O}$, we refer to set $(2^\mathcal{O} \setminus \emptyset)$ simply as $2^\mathcal{O}$.

**Data box**  Similarly to Definition 4.1.1, the DB is a transition system without final states, where transitions are nondeterministic but are triggered by sets of operations, rather than single operations.

**Definition 6.1.1** Let $\mathcal{O}$ be the operation alphabet of a nondeterministic service community $\mathcal{C}$. A data box with multiple operations for $\mathcal{C}$ is a tuple $\mathcal{DB} = (\mathcal{O}, D, d_0, \rho)$ where:

- $\mathcal{O}$ is the finite set of shared operations, i.e., the whole set of operations services can perform, each of which may or may not affect data box state;
6.1. THE FRAMEWORK

- $D$ is the finite set of data box states;
- $d_0 \in D$ is the initial state;
- $\rho \subseteq D \times 2^O \times D$ is the transition relation among DB states: $\rho(d, O, d')$ holds when the data box may evolve from state $d$ to state $d'$ when all operations included in $O$ are (concurrently) executed.

An important difference between this and the case of single operations is that, here, DB transitions are triggered by sets of operations that can be, as we will see, delegated to different available services, while the (basic) nondeterministic framework of Chapter 4 does not provide this feature. Concerning operation synchronization, as anticipated above, a data box transition labeled with multiple operations represents the existence of a proper synchronization of the labeling operations such that the modeled data box effects take place. In addition, we assume that each time a multiple operation execution, by some service, triggers a data box transition, operations are synchronized as required by the data box for the transition in order to take place.

Available Services Available services are able to execute, in general, many operations under a same transition. This suggests that when a target service transition—which, as it will be seen, requires, in general, multiple operation executions—is to be realized, all the operations it requires can be delegated to many available services. That is, differently from previous frameworks, we have multiple delegations.

Definition 6.1.2 A nondeterministic guarded available service with multiple operations, over data box $DB = (O, D, d_0, \rho)$ is a tuple $S = (O, S, s_0, \rho, G, S_f)$, where:

- $O$ is the same set of operations as in $O$;
- $S$ is the finite set of service states;
- $s_0 \in S$ is the initial state;
- $S_f \subseteq S$ is the set of final states;
- $G$ is a set of boolean functions $g : D \rightarrow \{\top, \bot\}$ called guards;
- $\rho \subseteq S \times G \times 2^O \times S$ is the service transition relation.

We assume that $(s, g, \emptyset, s) \in \rho$ for all $s \in S$ and $g \in G$. 

We adopt the same terminology as in Chapter 4. In particular, when \( \langle s, g, O, s' \rangle \in \rho \), we say that transition \( s \xrightarrow{g,O} s' \) (is) in \( S \). Given a state \( s \in S \), if there exists a transition \( s \xrightarrow{g,O} s' \) in \( S \) (for some \( g \) and \( s' \)) and the data box is in a state \( d \) such that \( g(d) = \top \) then operation set \( O \) is said to be executable in \( s \). A transition \( s \xrightarrow{g,O} s' \) in \( S \) denotes that \( s' \) is a possible successor state of \( s \), when operation set \( O \) is executed in \( s \), provided \( g(d) = \top \), \( d \) being the current data box state. We say that a service \( S \) over a data box \( DB \) is deterministic iff there is no \( DB \) state \( d \in D \) for which there exist, in \( S \), two distinct transitions \( s \xrightarrow{g_1,O} s' \) and \( s \xrightarrow{g_2,O} s'' \) such that \( s' \neq s'' \) and \( g_1(d) = g_2(d) = \top \). As in Chapter 4, given a deterministic service state and a legal operation set in that state, next only state is always known.

As an example of available service with multiple operations, focus on available services (Figures 6.1(a) and 6.1(b)) and observe that an operation which is nondeterministic when executed alone, may become deterministic (or vice versa, if deterministic when executed alone may become nondeterministic) when executed with other ones. This is the case, for instance, of operation \( a \) in \( S_1 \): if executed in conjunction with operation \( b \) it is deterministic while, when executed alone, it is not. As we said above, this allows to take into account the mutual interaction operations may have among themselves. From the perspective of a single (available) service, combinations of single operations give rise to new operations. However, they bring the information about the atomic operations they execute, that can be exploited for compositional purposes.

**Target Service and Community TS** As exactly as in Chapter 4, the target service \( S_t \) is the desired deterministic service whose transitions, triggered by multiple operations, can be guarded. Formally, it has the same form as available services, in the sense above, but is required to be deterministic.

As anticipated, since the target service executes a set of operations under a transition, it might be necessary selecting a set of available services to execute, as a whole, all such operations.

**Definition 6.1.3** We call community with multiple operations a set \( C = \{S_1, \ldots, S_n, DB \} \), where each \( S_i \) is an available service with multiple operations and \( DB \) is a data box with multiple operations.

As for the Community TS, from a high-level perspective, it represents all possible community evolutions when available services execute operations consistently with their specification. However, operation concurrency introduces an important difference with respect to Community TS in Chapter 4, in that, at each step, several, instead of one only, services may change state. Formally, we have the following Definition.
Definition 6.1.4 Let $\mathcal{C} = \{S_1, \ldots, S_n, DB\}$ be a community with multiple operations, where $S_i = \langle O_i, S_{i0}, g_i, G_i, S^f_i \rangle$ $(i = 1, \ldots, n)$ and $DB = \langle O, D, d_0, \rho \rangle$. The nondeterministic guarded community TS of $\mathcal{C}$, with multiple operations, implicitly defined over $DB$, is the tuple $\bar{S}_C = \langle \bar{O}, \bar{G}, S_C, s_{C0}, \gamma_C, S^f_C \rangle$, where:

- $\bar{O} = (2^O)^n$, with $2^O$ being as above, is the set of concurrent operations assigned to available services;
- $S_C = S_1 \times \ldots \times S_n$ is the (finite) set of $S_C$ states; given a state $s_C = \langle s_1, \ldots, s_n \rangle$, we denote, as usual, $s_i$ by $com_i(s_C)$ $(i = 1, \ldots, n)$;
- $s_{C0} \in S_C$, where $com_i(s_{C0}) = s_{i0}$ $(i = 1, \ldots, n)$ is the $S_C$ initial state;
- $\bar{G} = (G_1^\top \times \ldots \times G_n^\top)$ is the set of available service guards, where $G_i^\top = G_i \cup \{\top\}$;
- $\gamma_C \subseteq S_C \times \bar{G} \times \bar{O} \times S_C$ is the $S_C$ transition relation, where $\langle s_C, \bar{g}, \bar{O}, s'_C \rangle \in \gamma_C$, or $s_C \xrightarrow{\bar{g}, \bar{O}} s'_C$ is in $S_C$, for $\bar{g} = \langle g_1, \ldots, g_n \rangle$ and $\bar{O} = \langle O_1, \ldots, O_n \rangle$ if and only if:
  - for all $i$ such that $O_i \neq \emptyset$, $com_i(s_C) \xrightarrow{g_i, O_i} com_i(s'_C)$ in $S_i$;
  - for all $i$ such that $O_i = \emptyset$, $g_i = \top$ and $com_i(s_C) = com_i(s'_C)$;

- $S^f_C \subseteq S_C$ is the set of $S_C$ final states, where $s_C \in S^f_C$ if and only if $com_i(s_C) \in S^f_i$ for all $i = 1, \ldots, n$;

Observe how several services can be active under a same transition, this coming from the fact that multiple delegation is allowed. Indeed, transitions are triggered by tuples of operation sets, where $i$-th tuple refers to operations assigned to $S_i$. Obviously, in order for a transition to be allowed, all guards of those available services that some operation is assigned to need to be satisfied (by current data box state), while this is not required for services that no operation is assigned to (which, therefore, remain still). However, in this latter case, guards are set to $\top$, so to make explicit that their truth value is not relevant for the transition to be allowed.

Finally, we remark that each community TS state represents the new state reached by the community after all delegated operations are executed, provided all respective guards evaluate to $\top$.

Example 6.1.1 An example of available services and community TS with multiple operations is depicted in Figure 6.1. The reported services, as well as their respective operations, have no particular meaning, as the only purpose
CHAPTER 6. SERVICES WITH MULTIPLE OPERATIONS

of the example is to show how a community TS is built when multiple operations are considered. The whole example can be thought of as considering only fragments of available services in a larger community.

Consider the community TS of Figure 6.1(c). As we said, at each step, several operations can be executed by different services and this is reflected by transition labels, where pairs of operation sets represent the operations assigned to each service. In particular, each label has the form \( \langle g_1, g_2 \rangle : \langle O_1, O_2 \rangle \), where \( g_1 \) and \( O_1 \) refer to service \( S_1 \) while \( g_2 \) and \( O_2 \) refer to \( S_2 \). As usual, for each service, only transitions (in general, nondeterministic) whose guards evaluate to \( \top \) (referring to current data box state) can actually take place. In this case, we require this for all guards. So, a transition labeled with \( \langle g_1, g_2 \rangle : \langle O_1, O_2 \rangle \) can take place only if, provided data box is in state \( d \), both \( g_1(d) \) and \( g_2(d) \) evaluate to \( \top \). The effects of the transition are the joint effects of each available service, as the community TS describes.

As expected, when, for a service, an operation is nondeterministic, the community TS is such as well. For instance, consider transitions outgoing from \( \langle s_{10}, s_{20} \rangle \) labeled with \( \langle \top, g_3 \rangle : \langle \{a\}, \{a,d\} \rangle \). Since operation \( a \) is nondeterministic for available service \( S_1 \), with two possible outcomes, while \( S_2 \) executing \( \{a,d\} \) has deterministic effects, we have that two transitions are possible in the community TS. Similarly, this happens each time operation set \( \{a\} \) is executed by available service \( S_1 \).

Orchestrator The orchestrator has some features that are different from the ones presented in previous frameworks but that, as before, naturally reflect the generalization to multiple operations. First, it has the ability to stop, resume and assign operations to several, rather than one, services at each step, and, second, it makes its decisions based on a set of, rather than one, operations provided as input. In the following, we will provide more formal details about orchestrators, after the notions of traces and histories have been introduced.

6.2 Composing Services with Multiple Operations

Now, we are ready to formulate the composition problem for services that allow execution of multiple operations. As usual, we start by introducing some basic concepts, such as traces and histories, that will be necessary for the problem to be formally specified.

**Definition 6.2.1** Given a nondeterministic guarded available service with multiple operations \( S = \langle O, S, s_0, g, G, S^f \rangle \) over data box \( DB = \langle O, D, d_0, \rho \rangle \), a trace for \( S \) on \( DB \) is a possibly infinite sequence \( \tau = \langle s^0, d^0 \rangle \xrightarrow{O_1^1} \langle s^1, d^1 \rangle \xrightarrow{O_2^2} \ldots \), such that:
6.2. COMPOSING SERVICES WITH MULTIPLE OPERATIONS

Figure 6.1: Available service $S_1$ of Example 6.1.1.

(b) Available service $S_2$ of Example 6.1.1.

(c) Community TS of Example 6.1.1

Figure 6.1: Available service and Community TS of Example 6.1.1.
• \( (s^i, d^i) \in S \times D \) for all \( i \geq 0 \);

• \( (s^0, d^0) = (s_0, d_0) \);

• for all \( j > 0 \), there exists some \( g \in G \) such that \( g(d^j) = \top \), \( s^j \xrightarrow{s_0, O^{j+1}} s^{j+1} \) is in \( S \) and \( d^j \xrightarrow{O^{j+1}} d^{j+1} \) is in \( DB \).

A nondeterministic service history is any finite prefix of a trace, ending with a pair \( (s, d) \in S \times D \).

With respect to traces of nondeterministic available services with single operations, the only difference here is, as expected, the presence of multiple operations in transitions, which comes naturally from the definition of available service with multiple operations. On the other hand, a major difference with the frameworks presented in previous chapters is shown by community TS traces and histories when multiple operations are allowed. Indeed, here, several available services can be activated for operation execution at each step. Consequently, transitions appearing in traces or histories are labeled by all operation sets that each available service is request for execution. More formally, we have the following definition.

**Definition 6.2.2** Let \( C = \{S_1, \ldots, S_n, DB\} \) be a community with multiple operations, \( DB \) as above and let \( S_C = (\vec{G}, S_C, s_{C0}, \gamma_C, \vec{G}_C) \) be the respective community TS. A trace for a community with multiple operations \( S_C \) is a possibly infinite sequence \( \tau = (s^0_C, d^0) \xrightarrow{\vec{G}_1} (s^1_C, d^1) \xrightarrow{\vec{G}_2} \ldots \), such that:

• \( (s^i_C, d^i) \in S_C \times D \) for all \( i \geq 0 \);

• \( (s^0_C, d^0) = (s_{C0}, d_0) \);

• for all \( j > 0 \), there exists some \( \vec{g} = (g_1, \ldots, g_n) \in \vec{G} \) such that \( g_i(d^j) = \top \) \( (i = 1, \ldots, n) \), \( s^j_C \xrightarrow{g_i, O^{j+1}} s^{j+1}_C \) is in \( S_C \) and \( d^j \xrightarrow{O^{j+1}} d^{j+1} \) in \( DB \), where \( O^{j+1} = \bigcup_{i=1}^n O^{j+1}_i \), with \( O^{j+1} = (O^{j+1}_1, \ldots, O^{j+1}_n) \).

A nondeterministic community history is any finite prefix of a trace, ending with a pair \( (s_C, d) \in S_C \times D \).

We refer to the set of all and only histories as to \( H \). Moreover, we assign to functions of histories length and last an obvious semantics analogous in the spirit to previous frameworks.

Observe that, so far, we have looked at multiple operations as if they were usual operations, possibly unrelated to their atomic components – in fact, a set of synchronized operations has in general, effects that are not related to
any of its atomic operations. In other words, once the effects of multiple operation executions have been specified by TS’ specification, one can safely forget the component operations of each set $O_i$, rename it as if it were a new operation and simply consider the obtained TS as a new service or data box. However, when the effects of multiple operation execution over the data box are considered, as in above definition, this is no longer valid. Indeed, observe that each generic step $\langle s_C, d \rangle \xrightarrow{O} \langle s'_C, d' \rangle$ of a community TS trace (respectively, history) requires that (i) all operation sets in $\bar{O}$, i.e., $O_i$, are actually executable by the service they are respectively assigned to, i.e., $S_i$, and, importantly, that (ii) by joining all such sets, the obtained (multiple) operation set, $\bar{O} = \bigcup_{i=1}^{n} O_i$, triggers some transitions from $d$ to $d'$ in the data box. Clearly, the latter requirement relies strongly on which operations are included in each set $O_i$. Therefore, it is ultimately necessary recording the information about which operations are included in each $O_i$.

Based on the notions of traces and histories just introduced, we can provide a formal description of orchestrators in the context of multiple operations.

**Definition 6.2.3** Let $C = \{S_1, \ldots, S_n, DB\}$ be a community with multiple operations, $S_C$ the community TS and $H$ the respective set of histories. An orchestrator for $C$ is a possibly partial function $P: H \times 2^O \to (2^O)^n$ such that, if $P(h, O)$ is defined, the following holds, where $\bar{O} = P(h, O)$ and $s_C = \text{last}(h) = \langle s_1, \ldots, s_n, d \rangle$:

- $\bar{O} = \langle Q_1, \ldots, Q_n \rangle$ is such that $O = \bigcup_{i=1}^{n} Q_i$;
- there exists a transition $d \xrightarrow{O} d'$ in $DB$;
- there exists $\bar{g} \in \bar{G}$ and a transition $s_C \xrightarrow{\bar{g}O} s'_C$ in $S_C$ such that $g_i(d) = \top$ for $i = 1, \ldots, n$, where $s'_C = \langle s'_1, \ldots, s'_n, d' \rangle$.

Observe that, differently from Chapter 4, where the orchestrator purpose was to simply select an available service, in this context the orchestrator is more complex. Indeed, in addition to selecting those available services that will be activated to fulfill client’s request –i.e., services whose indices correspond to non-empty output components–, it also dispatches the operations, that is, it actually assigns an operation set to each service. Clearly, all such sets are required to cover the original client request $O$, as prescribed by first requirement.

Now, we can formally define when a target service trace is realized and, then, when an orchestrator realizes a target service with multiple operations.
Definition 6.2.4 Let $\mathcal{C} = \{S_1, \ldots, S_n, \mathcal{DB}\}$ be a community with multiple operations, $\mathcal{DB}$ as above and $S_C$ the respective community TS. Moreover, let $S_t$ be a target service with multiple operations over $\mathcal{DB}$ and $P : \mathcal{H} \times 2^O_0 \rightarrow (2^O_0)^n$ an orchestrator for $\mathcal{C}$. Given a target service trace $\tau = \langle s_C^0, d^0 \rangle \xrightarrow{O^1} \langle s_C^1, d^1 \rangle \xrightarrow{O^2} \ldots$, $P$ realizes $\tau$ if and only if:

- for all community TS histories $h \in \mathcal{H}_{\tau,P}$, $P(h, O^{\text{length}(h)+1})$ is defined, where $\mathcal{H}_{\tau,P} = \bigcup \mathcal{H}_{\tau,P}^j$ is the set of community TS histories induced by $\tau$ and $P$, inductively defined as follows:
  - $\mathcal{H}_{\tau,P}^0 = \{\langle s_C^0, d_0 \rangle\}$;
  - $\mathcal{H}_{\tau,P}^{j+1}$ is the set of $(j+1)$-length community histories of the form $h \xrightarrow{O^{j+1}} \langle s_C^{j+1}, d^{j+1} \rangle$, where:
    - $h \in \mathcal{H}_{\tau,P}^j$, with last($h$) = $\langle s_C^j, d^j \rangle$;
    - $O^{j+1}$ and $d^{j+1}$ are the same operation set and data box state at step $(j+1)$, as in $\tau$;
    - $P(h, O^{j+1}) = \langle Q_1^{j+1}, \ldots, Q_n^{j+1} \rangle = \bar{O}^{j+1}$, that is, after going through community history $h$, the orchestrator assigns operation subset $Q_i^{j+1}$ to available service $S_i$ (observe that, by orchestrator definition, we get that $\bigcup_{i=1}^n Q_i^{j+1} = O^{j+1}$);
    - there exists $\bar{g} = \langle g_1, \ldots, g_n \rangle$ such that, for each $i \in \{1, \ldots, n\}$, $g_i \in G_i$, $g_i(d^j) = \top$ and $s_C^j \xrightarrow{\bar{g} \bar{O}^{j+1}} s_C^{j+1}$ in $S_C$, that is, the community TS $S_C$ can perform the transition prescribed by $P$;

- for all $\langle s_t^\ell, d^\ell \rangle$ in $\tau$ such that $s_t^\ell \in S_t^\ell$ ($\ell \geq 0$), all $\ell$-length histories $h \in \mathcal{H}_{\tau,P}^\ell$ for which last($h$) = $\langle s_C^\ell, d^\ell \rangle$ are such that $s_C^\ell \in S_C^\ell$.

As it comes out, this Definition is very similar in the spirit to its previous counterparts (cf., e.g., Definition 4.2.4) but has some important differences. First, as mentioned above, at each step, all available services are assigned a (possibly empty) set of operations to execute. Therefore, they, in general, execute non-interleaved transitions. A natural requirement when realizing a trace, and hence its component transitions, is to guarantee that the union of all operations assigned to available services at each step (i.e., $Q_1^{j+1} \ldots Q_n^{j+1}$) cover the set of operations the client originally requested (namely set $O^{j+1}$).

Second, observe that a same operation requested by the client can be, in general, assigned to many available services. That is, sets $Q_1^{j+1} \ldots Q_n^{j+1}$ are not disjoint. This, in fact, corresponds to assigning a same task to several devices at the same time. Such feature may or may not reflect a property of the actual community that is being modeled: there might be domains where
disjointness of assigned operation sets is a necessary requirement. In this work, we deal with the less restrictive case. However, if needed, this requirement can be safely added to above definition, and naturally propagated to next ones, when necessary, without affecting correctness of the model and the obtained results, only slight modifications to the framework being required.

Analogously to the “single-operation” case, we now define when an orchestrator is a composition of a target service, with no substantial differences from previous definition, apart from the characterization of the framework components, i.e., community, data box and target service, involved.

**Definition 6.2.5** An orchestrator \( P \) for a community \( C \) with multiple operations is a composition of the target service with multiple operations \( S_t \) by the services in \( C \) over \( DB \) iff it realizes all traces of \( S_t \) on \( DB \).

**The composition problem** Now that all ingredients have been introduced, we can formally state, as usual, the composition problem:

Given a community with multiple operations \( C = \{S_1, \ldots, S_n, DB\} \) and a deterministic target service \( S_t \) over \( DB \), synthesize an orchestrator for \( C \) that is a composition of \( S_t \) by \( S_C \).

### 6.3 Composition Synthesis via MND-Simulation

Following the same approach as previous chapters, we introduce now a solution technique for the service composition problem in presence of multiple, synchronized, operations. Again, the technique is based on a simulation-like relation that extends Definition 4.3.1. As we said, the extension to multiple-operation services starts from [130], where the composition synthesis technique is based on a reduction of the composition problem to a PDL-SAT instance ([64, 90]). Here, we provide a new original contribution by extending the notion of ND-simulation relation to deal with multiple operations and by proposing a new solution technique based on it, thus yielding, as exactly as in the nondeterministic case, all the advantages brought by the flexible solutions one can obtain.

First of all, we introduce the formal notion of MND-simulation relation, that is, the natural extension of ND-simulation relation to deal multiple-operation services.

**Definition 6.3.1** Let \( C = \{S_1, \ldots, S_n, DB\} \) be a community with multiple operations and \( S_t \) a target service with multiple operations over \( DB \). An MND-simulation relation of \( S_t \) by \( S_C \) on \( DB \) is a relation \( R \subseteq S_t \times S_C \times D \) such that \( (s_t, s_C, d) \in R \) implies:
1. if $s_t \in S^f_t$ then $s_C \in S^f_C$;

2. for each transition $s_t \xrightarrow{g,O} s'_t$ in $S_t$ with $g(d) = \top$ if there exists a transition $d \xrightarrow{O} d'$ in $DB$ then there exists an $\tilde{O} = (O_1, \ldots, O_n) \in (2^O)^n$ such that $\bigcup_{i=1}^n O_i = O$ and the following holds:

   (a) there exists a transition $s_C \xrightarrow{\tilde{g},\tilde{O}} s'_C$ in $S_C$ with $\tilde{g} = (g_1, \ldots, g_n)$ and $g_i(d) = \top$ ($i \in \{1, \ldots, n\}$);

   (b) for all transitions $s_C \xrightarrow{\tilde{g},\tilde{O}} s'_C$ in $S_C$ with $g_i(d) = \top$ ($i \in \{1, \ldots, n\}$) and $d \xrightarrow{O} d''$ in $DB$, $(s'_t, s'_C, d'' \in R$.

Roughly speaking, an MND-simulation requires the community TS to be able to mimicking each transition executable by the target service, by assigning to each available service a subset of the operations the target service is able to execute, in a way such that all target service operations are covered.

This is the most general extension of simulation relation we provide in this work. As it can be seen, it naturally extends the notion of ND-simulation to deal with multiple operations. Observe that if (i) only singleton operation sets in target service transitions and (ii) only one nonempty $O_i$ is allowed then we get a definition that is, in the essence, equivalent to Definition 4.3.1 of ND-simulation relation, where the index $k$ corresponds to the index of the nonempty set $O_i$. Importantly, as we already anticipated, one cannot exploit ND-simulation relations to describe MND-simulation relations, for at least two reasons: first, in ND-simulation relations, synchronized service transitions are not allowed, i.e., only one service moves at each step; second, one cannot simply rename operation subsets and consider them as atomic operation, like in previous case, because there would be no way –unless one extends the construction– to capture the fact that target operation set $O$ is covered by the operation sets assigned to available services, namely $O_1, \ldots, O_n$.

**Definition 6.3.2** We say that a state $s_t$ of target service with multiple operations $S_t$ is MND-simulated by a state $s_C$ of a community TS with multiple operations $S_C$ on data box state $d$, denoted $\preceq_M (s_t, s_C, d)$ if and only if there exists an MND-simulation relation $R$ of $S_t$ by $S_C$ on $DB$ such that $(s_t, s_C, d) \in R$. Moreover, if $\preceq_M (s_{t0}, s_{C0}, d)$ then we say that $S_t$ is MND-simulated by $S_C$ on $DB$ or, equivalently, that $S_C$ MND-simulates $S_t$ on $DB$.

Analogously to previous frameworks, relation $\preceq_M$ is, in fact, the largest MND-simulation relation of $S_t$ by $S_C$ over $DB$. 
6.3. COMPOSITION SYNTHESIS VIA MND-SIMULATION

Computational Characterization Consider Algorithm 1 of Chapter 4 and observe that a similar procedure can be defined for computing an MND-simulation relation. In details, it is shown in Algorithm 3. Essentially, it executes the same procedure as Algorithm 1, with some slight modifications needed to deal with multiple operations. It is easy to see that termination and correctness of the algorithm can be shown by relying on arguments similar to those we resorted to for Algorithm 1. Also time complexity can be shown to be polynomial with respect to $|O|, n$ and the size of $S_t$, $S_C$ and $DB$. To see this, it is enough observing that not all $\vec{O} \in (2^O)^n$ need be considered, but only those appearing in $S_C$ transitions. Hence, we get the analogous of

Algorithm 3 $MNDS(S_t, S_C, DB)$ – Computes Largest MND-Simulation of $S_t$ by $S_C$ on $DB$

1: $R := S_t \times S_C \times D$
2: $R := R \setminus \{(s_t, s_C, d) \mid (s_t \in S^f_t \land s_C \notin S^f_C)\}$
3: repeat
4: \[ R := R \setminus D, \text{ where } D \text{ is the set of } (s_t, s_C, d) \in R \text{ for which there exist (i) a transition } d \xrightarrow{O} d' \text{ in } DB \text{ and (ii) a transition } s_t \xrightarrow{g,O} s'_t \text{ in } S_t \text{ with } g(d) = \top \text{ such that for all } \vec{O} = \langle O_1, \ldots, O_n \rangle \in 2^O \text{ with } \bigcup_{i=1}^n O_i = O \text{ one of the following holds:}
\]

1. there is no transition $s_C \xrightarrow{\vec{g},\vec{O}} s'_C$ in $S_C$ with $\vec{g} = (g_1, \ldots, g_n)$ such that $g_i(d) = \top$ for all $i = 1, \ldots, n$;
2. there exists a transition as above but $(s'_t, s'_C, d') \notin R$.
5: until $(D = \emptyset)$
6: return $R$

Theorem 4.3.1, when multiple operations are allowed.

**Theorem 6.3.1** Given a target service $S_t$ and a community $TS S_C$, both with multiple operations, over data box $DB$, computing the largest MND-simulation relation of $S_t$ by $S_C$ on $DB$ is a PTIME problem.

By following the same schema of Chapter 4, we can now prove the following two results.

**Lemma 6.3.1** Let $C = \{S_1, \ldots, S_n, DB\}$ be a community and $S_t$ a target service, both with multiple operations over $DB$. If there exists an orchestrator $P$ for $C$ that is a composition of $S_t$ by $S_C$ on $DB$ then $\preceq_M (s_{t0}, s_{C0}, d_0)$.

The viceversa also holds.
Lemma 6.3.2 Let $\mathcal{C} = \{S_1, \ldots, S_n, DB\}$ be a community and $S_t$ a target service, both with multiple operations, over $DB$. If $\preceq_M (s_{t0}, s_{C0}, d_0)$ then there exists an orchestrator $P$ for $\mathcal{C}$ that is a composition of $S_t$ by $S_C$ on $DB$.

Proofs are omitted as they are just adaptations of proof of lemmas 4.3.1 and 4.3.2, already shown in Chapter 4, modified to take into account multiple operations. Their conjunction, again, leads to the following fundamental result.

Theorem 6.3.2 Let $\mathcal{C} = \{S_1, \ldots, S_n, DB\}$ be a community and $S_t$ a target service, both with multiple operations, over $DB$. An orchestrator $P$ for $\mathcal{C}$ that is a composition of $S_t$ by $S_C$ on $DB$ exists if and only if $\preceq_M (s_{t0}, s_{C0}, d_0)$.

Proof: Direct consequence of Lemmas 6.3.1 and 6.3.2.

This result allows to adopt the same procedure as in the case of single-operation scenarios, to check for the existence of a composition, namely: compute the largest MND-simulation relation $\preceq_M$ of $S_t$ by $S_C$ and then check whether $\langle s_{t0}, s_{C0}, d_0 \rangle \in \preceq_M$. Again, an argument analogous to the one discussed in Chapter 4 leads to conclude that this technique is EXPTIME and, thus, optimal with respect to worst-case time complexity.

It remains to show how a composition can be actually built. In the following, an extension of the NCG in Definition 4.3.3 that is able to deal with multiple operations is presented. Proofs of results are omitted as they have essentially the same schemas as in the single-operation case, with some slight, intuitive, modifications.

First of all, we extend the definition of controller generator to cover also multiple-operation scenarios.

Definition 6.3.3 Let $\mathcal{C} = \langle S_1, \ldots, S_n, DB \rangle$ be a community and $S_t$ a target service over $DB$, both with multiple operations, such that $S_t$ is MND-simulated by $S_C$. The nondeterministic composition generator with multiple operations (MNCG) of $\mathcal{C}$ for $S_t$ is a tuple $\text{MNCG} = \langle \mathcal{O}_0, S_g, s_{g0}, \omega_g, \delta_g \rangle$, where:

1. $\mathcal{O}_0$ is the finite set of community multiple operations;
2. $S_g = S_t \times S_C \times D$ is the (finite) set of MNCG states;
3. $s_{g0} = \langle s_{t0}, s_{C0}, d_0 \rangle$ is the MNCG initial state;
4. $\omega_g : S_g \times \mathcal{O}_0 \rightarrow \mathcal{P}(\mathcal{O}_0)^n$ is the assignment selection function ($\mathcal{P}$ stands for power set). For $s_g = \langle s_t, s_C, d \rangle$, $\omega_g(s_g, O)$ is defined if and only if:
   - $\preceq_M (s_t, s_C, d)$;
6.3. COMPOSITION SYNTHESIS VIA MND-SIMULATION

- there exists a transition \( s_t \xrightarrow{g,O} s'_t \) in \( S_t \), for some guard \( g \) such that \( g(d) = \top \);
- there exists a transition \( d \xrightarrow{O} d' \) in \( DB \),

in such case:

\[
\omega_g(s_g, O) = \{ \tilde{O} = \langle O_1, \ldots, O_n \rangle \mid \bigcup_{i=1}^n O_i = O, \\
s'_c \xrightarrow{\tilde{g},\tilde{O}} s''_c \text{ is in } S_c \text{ and } \preceq_M (s'_t, s'_c, d'), \\
\text{for } \tilde{g} = \langle g_1, \ldots, g_n \rangle \in \tilde{G}, \text{ with } g_i(d) = \top (i = 1, \ldots, n) \}
\]

5. \( \delta_g \subseteq S_g \times (2^O)^n \times S_g \) is the MNCG transition relation, where \( \langle s_g, \tilde{O}, s'_g \rangle \in \delta_g \), or \( s_g \xrightarrow{\tilde{O}} s'_g \) is in MNCG if and only if \( \tilde{O} = \langle O_1, \ldots, O_n \rangle \in \omega_g(s_g, O) \) and, for \( s_g = \langle s_t, s_c, d \rangle \) and \( s'_g = \langle s'_t, s'_c, d' \rangle \):

- there exists a transition \( s_t \xrightarrow{g,O} s'_t \) in \( S_t \) with \( g(d) = \top \), where \( O = \bigcup_{i=1}^n O_i \);
- there exists a transition \( s_c \xrightarrow{\tilde{g},\tilde{O}} s''_c \) in \( S_c \), \( \tilde{g} = \langle g_1, \ldots, g_n \rangle \), with \( g_i(d) = \top (i = 1, \ldots, n) \);
- there exists a transition \( d \xrightarrow{O} d' \) in \( DB \);

Intuitively, the MNCG works the same way as the NCG except that it deals with multiple operations and simultaneous available service activation. Given the current states of the community and the target service, plus the operation set that a client requests for execution, the MNCG outputs, through \( \omega_g \), the set of all assignments to available services such that all the operations requested by the client are executed. When one of such assignments is selected—e.g., an orchestrator performs its choice—the MNCG performs a (possibly nondeterministic) transition, coherently with the chosen assignment, that depends on the actual evolution of the available services.

Once the MNCG is built, it can be exploited to build all those orchestrators that realize the desired target service \( S_t \), i.e., to build compositions. Indeed, the analogous of Theorem 4.3.5, stating that, in the ND scenario, an orchestrator is a composition if and only if it is an ND-generated composition, holds also in this context. First, some preliminary notions are needed.

**Definition 6.3.4** Given an MNCG for \( S_t \) and \( S_c \), a MNCG trace is a possibly infinite sequence \( s^0_g \xrightarrow{g_1} s^1_g \xrightarrow{g_2} \ldots \) such that (i) \( s^0_g = s_{g0} \) and (ii) for each \( i \geq 0 \), \( s^i_g \xrightarrow{g_{i+1}} s^{i+1}_g \) in MNCG. An MNCG history is any finite prefix of a trace, ending with a state. We define \( \mathcal{H}_{mng} \) as the set of all MNCG histories.
By previous Definitions, 6.3.3 and 6.3.4, we obtain the following, straightforward, Lemma.

**Lemma 6.3.3** If $s_g^0 \xrightarrow{O^1} s_g^1 \xrightarrow{O^2} \ldots$ is an MNCG trace then, for all $s_g^i = \langle s_g^i, t_g^i, d_g^i \rangle \ (i \geq 0)$, $\preceq_M (s_g^i, S_g^i, d_g^i)$.

Again, by means of straightforward manipulations of MNCG traces and histories, we are able to derive target and community traces and histories, similarly to what shown in Definition 4.3.5. More importantly, it will be shown that all target (community) traces and histories can be obtained in this way.

**Definition 6.3.5** Given an MNCG trace $\tau_g = s_g^0 \xrightarrow{O^1} s_g^1 \xrightarrow{O^2} \ldots$, where $s_g^i = \langle s_g^i, t_g^i, d_g^i \rangle$,

- the MNCG-derived community trace $\tau_C$ is the sequence $\tau_C = \langle s_C^0, d_C^0 \rangle \xrightarrow{O^1} \langle s_C^1, d_C^1 \rangle \xrightarrow{O^2} \ldots$, obtained by projecting out from each $\tau_g$ state $s_g^i$ the $S_t$ component $s_t^i$;

- the MNCG-derived target service trace $\tau_t$ is the sequence $\tau_t = \langle s_t^0, d_t^0 \rangle \xrightarrow{O^1} \langle s_t^1, d_t^1 \rangle \xrightarrow{O^2} \ldots$, obtained by projecting out from each $\tau_g$ state $s_g^i$ the $S_C$ component $s_C^i$ and replacing each transition label $O^j = \langle O_{t_1}^j, \ldots, O_{t_n}^j \rangle$ with $O^j = \bigcup_{i=1}^{n} O_{t_i}^j$;

- an MNCG-derived community (target service) history is any finite prefix of a MNCG-derived community (target service) trace, ending with a state.

The following results are straightforward generalizations of Lemmas 4.3.4 and 4.3.5.

**Lemma 6.3.4** Each MNCG-derived community (target service, respectively) trace is a community (target service) trace, and analogously for histories.

**Lemma 6.3.5** For each target service trace $\tau_t$ (history $h_t$, respectively) there exists an MNCG trace $\tau_g$ (history $h_g$) such that $\tau_t$ ($h_t$) is MNCG-derived from $\tau_g$ ($h_g$).

In other words, one can obtain exactly the set of target service trace, when multiple operations are considered, by MNCG-deriving all MNCG traces (and respectively for histories).

Now, based on the notions discussed so far, let us show how compositions can be obtained from the MNCG.
Definition 6.3.6 Let $C, DB, S_t$ and $S_C$ be as above. Assume $S_t$ is MND-simulated by $S_C$ on $DB$ and let MNCG be the corresponding nondeterministic composition generator with multiple-operations. An MND generated composition is an orchestrator $P : H \times 2^O \to (2^O)^n$ satisfying the following, inductively defined, requirement:

- let $s^0_g \in \mathcal{H}_P$, where $\mathcal{H}_P \subseteq \mathcal{H}_{mng}$ and $s^0_g = (s_{t0}, s_{C0}, d_0)$;
- if $h \in \mathcal{H}_P$ then for all $O \in 2^O$ such that $\omega_g(\text{last}(h), O)$ is defined:
  - $P(h_C, O) = \vec{O} \in \omega_g(\text{last}(h), O)$, where $h_C$ is the community history MNCG-derived from $h$;
  - for all transitions last($h$) $\xrightarrow{\vec{O}} s_g^{\text{length}(h)+1}$ in MNCG, $h \xrightarrow{\vec{O}} s_g^{\text{length}(h)+1} \in \mathcal{H}_P$.

As it comes out, there is no major difference between MND and ND generated compositions of Chapter 4. Indeed, such differences are already taken into account by function $\omega_g$ which guarantees that all operation sets assigned to available services cover the operation set that the client requested. In other words, given the MNCG, generated compositions are obtained in exactly the same way, in either, single- or multiple-operation, contexts.

Finally, we report the following fundamental theorems which, together, yield the equivalence of MND generated compositions and compositions.

Theorem 6.3.3 Let $C, DB, S_t$ and $S_C$ be as above. Assume $S_t$ is MND-simulated by $S_C$ on $DB$ and consider the corresponding MNCG. If $P$ is an MND generated composition, then $P$ is a composition of $S_t$ by $S_C$ on $DB$.

Theorem 6.3.4 Let $C, DB, S_t$ and $S_C$ be as above. If $P$ is a composition of $S_t$ by $S_C$ on $DB$ then it is an MND generated composition.

And, as anticipated, we get the following.

Theorem 6.3.5 Let $C, DB, S_t$ and $S_C$ be as above. An orchestrator $P$ of $S_C$ for $S_t$ on $DB$ is a composition of $S_t$ by $S_C$ on $DB$ if and only if it is an MND generated composition.
Chapter 7

Distributed Service Composition

In this Chapter, we consider a scenario where available services are distributed and the execution of a centralized orchestrator is not convenient, not possible, or neither. Clearly, in such context, the techniques presented in past chapters fail in addressing the service composition problem, as their output is an orchestrator implicitly assumed to be executed by a central entity.

There are many applicative scenarios where the execution of a central orchestrator is not possible or recommended. For example, a RoboCup soccer team includes multiple players acting on their own, possibly exchanging messages with each other [40]. Robot ecologies [145, 127], the development of autonomous microrobot groups consisting of many heterogeneous members exhibiting collective behavior and intelligence, also offer an excellent example of a domain where coordination and task distribution might be of great benefit to obtain a global desired behavior. In all these cases, the set of available services, representing, as usual, the actual devices involved in the application, cannot be taken as a library accessible from a central system. Indeed, no central entity can be assumed at all and, in fact, the most one could realistically allow under such scenarios is some sort of local control on the existing services, together with some kind of communication.

Here, we address the problem of building a distributed orchestrator, that is, an orchestrator which needs no central entity and can be executed by a set of distributed devices. The essential results we show have been proposed in [130], however, here, we resort to a simulation-based synthesis technique, instead of the PDL-SAT-based one adopted in [130]. With such choice, all orchestrators, rather than one only, can be computed, within the same temporal complexity class as in the case of centralized orchestrator.


7.1 The Framework

In addressing the distributed service composition problem, we not only need to envision a new type of (distributed) solution, but we also need a rich framework that allows to model typical issues of distributed contexts. For instance, it is unrealistic to assume that only one service acts at each step, and hence, many concurrent operations have to be allowed. Here, therefore, we refer to the framework developed in Chapter 6, which is expressive enough for our purposes, once we assume a communication infrastructure services can exploit for (control-oriented) message exchange.

Local Orchestrators  First, let us be more precise about the problem. From an abstract point of view, we are given a set of available services, a data box and a target service, and the objective is to implement an orchestrator that realizes the target service. As we have seen, all centralized solutions proposed so far relied on the possibility of accessing information about all available services. In this scenario, such a feature is no longer present, that is, distribution of available services prevents direct access, by a central entity, to information about available service current states. So, the distributed problem still consists in finding an orchestrator, but it needs to be implemented in some distributed way, as it cannot be centralized. The adopted approach is to provide each available service with a local orchestrator that is able to (i) exchange information with other orchestrators and to (ii) control –i.e., stop, resume and assign operations to– its corresponding available service. Such orchestrator has full observability only on the service it is attached to and, as usual, on the data box, while it has not direct access to other services’ state information.

Thus, the problem becomes the synthesis of all such local orchestrators so that, when executed, as a whole, their evolutions guarantee the realization of the target service. Formal details about local orchestrators and the notion of target service realization will be provided in forthcoming sections.

State Reconstruction and Messages  The main difficulty here is that each local orchestrator cannot access the complete community history, but only the local history of the service it is attached to. That is local orchestrator are not, by their own, able to reconstruct the state of the whole community. Such feature is, clearly, fundamental for local orchestrator in order to make (correct) decisions about which operation(s) have to be assigned to the attached service. To overcome this obstacle, and then make orchestrators able to cooperate in order to realize the target service, a communication infrastructure is assumed that allows local orchestrators to broadcast messages from a set \( \mathcal{M} \) of message types. At this stage, we do not put any limit on the information that local
7.1. THE FRAMEWORK

orchestrators may potentially exchange, though, in the following, it will be shown that, in fact, only a finite number of message types is enough.

Example 7.1.1 Consider a site surveilled by two identical cameras, modeled as available services $S_1$ and $S_2$, represented in Figure 7.1(a). We adopt the same notation as in Chapter 6. Both cameras are able to either track objects or taking photos (operations $track$ and $photo$, respectively) but not both at the same time. Each time a photo is taken, it is stored into a local, i.e., on-camera, buffer which may, or may not, become full. Since the quantity of available memory is not explicitly modeled, such situation is unpredictable –i.e., there is no way to state whether the next photo will or will not fill up the buffer–, photo is modeled as a non-deterministic operation which leads the camera to state $S_2$ each time there is no space available on local buffer. A camera can empty its buffer by performing operation $send$, which sends the buffer content to a remote device (e.g., a hard drive). A $send$ operation can only be executed when some space is available on the remote device, otherwise the pictures currently stored into the local camera buffer would be lost. To prevent this, the camera may ask the remote device for freeing additional space by performing operation $freehdd$. When this happens, the remote storage device applies some policy to provide the camera with additional space.

Data box $DB$ models the remote storage device. Its possible evolutions are depicted in Figure 7.1(b): state $d_0$ models a situation where there is space available on the device, whereas $d_1$ is a state where no space is left. Consider the looping transition in $d_0$ and its labels. To reduce figure size, the following convention is adopted: any of the labeling sets or any union of two or more of them allows the transition. So, for instance, operation set $\{send\}$ allows the looping transition on $d_0$, as well as $\{send, photo\}$. In the latter case, however, also a transition from $d_0$ to $d_1$ is allowed, which is labeled by operation set $\{send, photo\}$. For all other transitions, the usual conventions are adopted, that is, they are allowed only by labeling operation sets.

Target service $S_t$, depicted in Figure 7.1(c), requires the ability to perform operations $send$, $photo$ and $freehdd$ while keeping objects tracked: a task that cannot be performed by a single camera –this is why we resort to service composition. In order to preserve local buffers and/or the remote device from going out of memory, a conservative strategy is adopted: after taking a picture, (i) the remote device is asked for additional space (i.e., $freehdd$ is executed) and (ii) a $send$ operations is performed to empty the local buffer.

1For instance, old or unused data are removed, or additional space is “sold” to the camera by automatically charging an account.
(a) Available services $S_1$ and $S_2$ for Example 7.1.1. (b) Data box $DB$ for Example 7.1.1. (c) Target service $S_t$ for Example 7.1.1.

Figure 7.1: Available and target services of Example 7.1.1.
7.2 Distributed Orchestrators

In this Section, we formally define distributed orchestrators, along with other notions.

First of all, we introduce local service traces with messages, i.e., the analogous of available service traces introduced in Chapter 6, where messages are considered. Local histories are defined, as usual, as finite prefixes of traces.

**Definition 7.2.1** Given a nondeterministic guarded available service with multiple operations $S = \langle O, S, s_0, G, S^f \rangle$ over data box $DB = \langle O, D, d_0, \rho \rangle$ and a set of message types $M$, a local trace for $S$ on $DB$ with $M$ (also called, for short, local trace for $S$, when $DB$ and $M$ are clear from the context) is a possibly infinite sequence $\tau = \langle s_0, d_0, M_0 \rangle \xrightarrow{Q_1} \langle s_1, d_1, M_1 \rangle \xrightarrow{Q_2} \ldots$, where $Q_i \in 2^M$, such that:

- $\langle s_0, d_0 \rangle \xrightarrow{Q_1} \langle s_1, d_1 \rangle \xrightarrow{Q_2} \ldots$ is a trace for $S$ on $DB$, as in Definition 6.2.1;
- $M_0 = \emptyset$ and $M_i \subseteq M$ for $i > 0$.

A local history for $S$ on $DB$ with $M$ is any finite prefix of a local trace for $S$ on $DB$ with $M$, ending with a tuple $\langle s, d, M \rangle \in S \times D \times 2^M$. The set of all local histories for $S$ on $DB$ with $M$ is denoted by $H_{M,S}$.

In this Definition, observe that, except for $M^0$, no constraint is put on the messages that appear along the trace. Indeed, local services have no communication ability—they are as in Definition 6.1.2—and they know nothing at all about messages, though indirectly affected by them. The only module an available service communicates with is its attached local orchestrator, which, by exchanging messages with other local orchestrators, makes decisions and instructs the available service to either remain still or execute some operation set. As one can intuitively expect, we will see that, since each orchestrator broadcasts messages, the message sets $M_i$ included in a local service trace are subject to all local orchestrators attached to community available services.

Each time the client requests execution of an operation set $O$, it simply broadcasts the request to the whole community. Then, all local orchestrators receive such request and, according to their internal specification, delegate an operation subset $Q \subseteq O$ to their respective available service. Notice that in local traces as defined above, the information about original client request is lost. Since it is relevant to our purposes, we define the following extension.

**Definition 7.2.2** Given a nondeterministic guarded available service with multiple operations $S = \langle O, S, s_0, G, S^f \rangle$ over data box $DB = \langle O, D, d_0, \rho \rangle$ and a set of message types $M$, an extended local trace for $S$ on
\( DB \) with \( M \) (also called, for short, extended local trace for \( S \), when \( DB \) and \( M \) are clear from the context) is a possibly infinite sequence \( \tau = \langle s^0, d^0, M^0 \rangle \overset{Q^1}{\longrightarrow} \langle s^1, d^1, M^1 \rangle \overset{Q^2}{\longrightarrow} \ldots \), such that:

- \( Q^1 \subseteq O^1 \in 2^O \);
- \( \langle s^0, d^0, M^0 \rangle \overset{Q^1}{\longrightarrow} \langle s^1, d^1, M^1 \rangle \overset{Q^2}{\longrightarrow} \ldots \) is a local trace as in Definition 7.2.1;

An extended local history for \( S \) on \( DB \) with \( M \) is any finite prefix of an extended local trace for \( S \) on \( DB \) with \( M \), ending with a tuple \( \langle s, d, M \rangle \in S \times D \times 2^M \). The set of all extended local histories for \( S \) on \( DB \) with \( M \) is denoted by \( \mathcal{H}^+_M, S \).

We can now provide a formal definition of local orchestrator. As we said, it is a module externally attached to an available service, able to control its operation. It has the ability of activating/resuming the controlled service by delegating a (possibly empty) set of operations. Also, it is able to broadcasting messages, observing attached service’s current state and accessing, at every step, all messages broadcasted by other local orchestrators and the client, which issues requests for operation set execution. Lastly, the orchestrator has full observability on the data box.

**Definition 7.2.3** Let \( S \) be a nondeterministic guarded available service with multiple operations over data box \( DB \) and let \( M \) be a set of message types. A local orchestrator for service \( S \) is a pair \( PL = \langle P, B \rangle \) where:

- \( P : \mathcal{H}^+_M, S \times 2^O \rightarrow 2^O \)
- \( B : \mathcal{H}^+_M, S \times 2^O \times S \rightarrow 2^M \)

Intuitively, a local orchestrator is a machine that, at each step, makes two decisions:

- based on the extended local history the service executed up to current step and the set \( O \) of operations that the client requested for execution at the same step, function \( P \) states which operation set \( Q \subseteq O \) is to be executed by the service. Observe that extended histories are relevant as local orchestrators may make different decisions based on the history of operations executed up to a given step, as exactly as centralized orchestrators do (cf. Chapter 6):
- under the same circumstances as above, after the service executed operation set \( Q \) and next service state has been observed by the orchestrator, function \( B \) states which messages are to be broadcasted.
In this Chapter, and in general when dealing with distributed services, we assume that each available service $S_i$ in community $C$ has a local orchestrator $P_L^i$ attached. When saying, by slight abuse of terminology, that an available service “sends” or “receives” a message, we actually refer to the attached orchestrator. Moreover, we remark that when a message is broadcasted, it can be read by all active local orchestrators.

We are now ready to formalize the notion of distributed orchestrator.

**Definition 7.2.4** Let $C = \{S_1, \ldots, S_n, DB\}$ be a community of distributed available services with multiple operations. A distributed orchestrator is a set $DP = \{P_L^1, \ldots, P_L^n\}$ of local orchestrators $P_L^i = (P_i, B_i)$ as defined above, one for each available service.

Similarly to previous scenarios, the only orchestrators of interest are those that realize the desired target service. As usual, in order to define target service realization, we first introduce the notion of community trace (with messages) and trace realization.

Let see how community evolutions, i.e., traces, are represented when available services are distributed. The main difference with previous definitions (cf., e.g., Definition 6.2.2) is that broadcasted messages have to be considered, in addition to usual information such as executed operations and the state each available service reaches.

**Definition 7.2.5** Let $C = \{S_1, \ldots, S_n, DB\}$ be a community with multiple operations and distributed available services, and $S_C$ the corresponding community TS. In addition, let $\mathcal{M}$ be a set of message types. A community trace with messages is a sequence of community and data box states, messages and operation sets $\tau = (s_C^0, d_C^0, M^0) \overset{\bar{o}_1}{\rightarrow} (s_C^1, d_C^1, M^1) \overset{\bar{o}_2}{\rightarrow} \ldots$, where:

- $M^0 = \emptyset$ and $M^j \subseteq \mathcal{M}$, $j \geq 0$;
- $\bar{o}_j \in \bar{\mathcal{O}}$, $j \geq 0$, as in Definition 6.2.2;
- $\tau_C = (s_C^0, d_C^0) \overset{\bar{o}_1}{\rightarrow} (s_C^1, d_C^1) \overset{\bar{o}_2}{\rightarrow} \ldots$ is a (usual, as in Definition 6.2.2) community trace for $S_C$ on $DB$.

A community history with messages is any finite prefix of a community trace with messages. The set $\mathcal{H}_\mathcal{M}$ shall denote the set of all community histories with messages.

Based on above definition, we also define projected local histories, which will be useful in the following.
Definition 7.2.6 Let \( h \in \mathcal{H}_M \) be an \( \ell \)-length community history with messages, as above. Then \( h_i \) denotes the \( S_i \) projected extended local history with messages \( h|_i = \langle s^0_C, d^0, M^0 \rangle \xrightarrow{Q^1_i} \ldots \xrightarrow{Q^l_i} \langle s^\ell_C, d^\ell, M^\ell \rangle \), where \( O^j = \bigcup_{u=1}^n Q^j_u \) for \( \tilde{O}^j = \langle Q^j_1, \ldots, Q^j_n \rangle \) as in \( h \).

It is worth noticing that messages are “shared” by the local history of each available service, that is, the same messages appearing in a community history also appear in the projected local history of each available service, in the same order. The same also happens for operation sets \( O^j \) as they are the ones client requests, which are the same received by all orchestrators.

Now, we can formally define when a distributed orchestrator realizes a target service trace.

Definition 7.2.7 Let \( \mathcal{C} = \{S_1, \ldots, S_n, DB\} \) be a community of distributed available services with multiple operations, \( S_i \) a target service and \( \mathcal{M} \) a set of messages. Given a target service trace (without messages, as in Definition 6.2.1) \( \tau = \langle s^0_i, d^0 \rangle \xrightarrow{O^1_i} \langle s^1_i, d^1 \rangle \xrightarrow{O^2} \ldots \), we say that a distributed orchestrator \( DP = \{P_1^L, \ldots, P_n^L\} \), \( P_i^L = \langle P_i, B_i \rangle \), realizes \( \tau \) if and only if:

- for each \( \ell \)-length community TS history with messages \( h \in \mathcal{H}_M, \tau \), all \( P_i(h|_i, O^{\ell+1}) \)'s are defined \( (i = 1, \ldots, n) \) so that

\[
\bigcup_{i=1}^n P_i(h|_i, O^{\ell+1}) = O^{\ell+1},
\]

i.e., the union of all operation sets assigned to each available service by its local orchestrator covers the requested operation set \( O^{\ell+1} \), where \( \mathcal{H}_{M, \tau} = \bigcup_{\ell} \mathcal{H}_{M, \tau}^\ell \) is the set of community histories with messages induced by \( DP \), inductively defined as follows:

- \( \mathcal{H}_{M, \tau}^0 = \{\langle s^0_C, d^0, M^0 \rangle\} \); with \( s^0_C = s^0, d^0 = d^0, M^0 = \emptyset \);
- \( \mathcal{H}_{M, \tau}^{j+1} \) is the set of \( (j + 1) \)-length community histories of the form \( h \xrightarrow{O^{j+1}} \langle s^j_C, d^{j+1}, M^{j+1} \rangle \), where:
  * \( h \in \mathcal{H}_{M, \tau}^{j} \);
  * \( d^{j+1} \) is the same data box state as at \( (j+1) \)-th \( \tau \) step;
  * \( \tilde{O}^{j+1} = \langle P_1(h|_1, O^j), \ldots, P_n(h|_n, O^j) \rangle \). Recall that, by definition, \( \bigcup_{i=1}^n P_i(h|_i, O^j) = O^j \), so the requested operation set is covered by joining all local orchestrator assignments;
  * there exists \( \tilde{g} = \{g_1, \ldots, g_n\} \) such that, for each \( i \in \{1, \ldots, n\} \), \( g_i \in G_i \), \( g_i(d^j) = \top \) and \( s^j_C \xrightarrow{O^{j+1}} s^j_C \) (\( \tilde{O}^{j+1} \) as defined above), that is, the community TS can perform the transition prescribed by \( DP \).
7.3. LOCAL ORCHESTRATOR SYNTHESIS

\[ M^{j+1} = \bigcup_{i=1}^{n} B_i(h_i, O^{j+1}, s^{j+1}_i), \]
where \( s^{j+1}_C = (s^{j+1}_1, \ldots, s^{j+1}_n) \), that is, the set of broadcasted messages is the union of all messages broadcasted by each local orchestrator. Observe that, according to the semantics of function \( B_i \), the returned value depends on the state reached by the service after operation set assigned by \( P^L_i \) has been executed.

- for all \( (s^\ell, d^\ell) \) in \( \tau \) such that \( s^\ell \in S^\ell_f (j \geq 0) \), all \( \ell \)-length histories \( h \in H^\ell_{M, \tau} \) (where \( H^\ell_{M, \tau} \) is as defined above) are such that \( s^\ell_C \in S^\ell_C \), with \( \text{last}(h) = (s^\ell_C, d^\ell, M^\ell) \).

As usual, we intuitively say that an orchestrator realizes a target service trace if it is able to control the community TS evolution, so that the sequence of operation sets appearing in the trace is mimicked by the available services. Clearly, since the community TS is nondeterministic, such control is only partial. In order to do this, each local orchestrator can use the (local) history of its service together with the (global) messages that have been broadcasted so far. In some sense, implicitly through such messages, each local orchestrator receives information on the global community history, i.e., it reconstructs such history, that can be used to make the right decision. Furthermore, at each step, each local orchestrator broadcasts messages, that will be used in the next step by other orchestrators to decide how to proceed.

Based on the above definition, we can now provide the following.

**Definition 7.2.8** Given a community \( C = \{S_1, \ldots, S_n, DB\} \) and a target service \( S_t \), both with multiple operations, a distributed orchestrator \( DP \) for \( C \) realizes \( S_t \) on \( DB \) if and only if it realizes all its traces.

In next sections we will show how a distributed orchestrator can be built so to realize a desired target service.

### 7.3 Local Orchestrator Synthesis

The natural extension of the service composition problem to distributed scenarios is as follows.

**The Service Composition Problem for Distributed Services**

Given a community \( C = \{S_1, \ldots, S_n, DB\} \) of distributed available services with multiple operations, a set of messages \( M \) and a deterministic target service (with multiple operations) over \( DB \), synthesize a distributed orchestrator which realizes the target service by concurrently running the available services under the control of their respective orchestrators.
In order to compute a distributed orchestrator, i.e., a team of local orchestrators, that solves the problem, we start from centralized orchestrators, that is, those defined in the nondeterministic, multiple-operation scenario of Chapter 6. Observe that, given an instance of the distributed problem, one can solve it without taking distribution into account. Indeed, the instance is the same in both the centralized and the distributed scenarios, the actual difference being that a centralized orchestrator is not actually executable in the latter. Assume that a centralized orchestrator has been somehow computed, for instance by using some of the simulation-based techniques shown before. Ideally, we wish to control each available service by using a local orchestrator that is a copy of the centralized one, that knows the whole community history, at each point, and that, according to current history, assigns an operation set to its respective service according to the centralized orchestrator, so to fulfill current client’s request. In the practice, however, a main obstacle arises: in order to know current community history, local orchestrators need the ability to reconstruct current state but this, as we pointed out, is unfortunately not available. Better said, it is not available off-the-shelf: in fact, by suitably exploiting their capability to exchange messages, local orchestrators can cooperate to make each of them work as if it knew the current community state. It is worth reminding that data box state is assumed observable to local orchestrators. The next Theorem shows how, from a finite centralized solution, a distributed one can be obtained.

**Theorem 7.3.1** Let $\mathcal{C} = \{S_1, \ldots, S_n, DB\}$ be a community of distributed available services with multiple operations, $S_t$ a target service and $M$ a set of messages. If there exists a centralized orchestrator $P$ for $S_C$ that realizes $S_t$ on $DB$ then there exists a distributed orchestrator $DP = \{P_L^1, \ldots, P_L^n\}$ that realizes $S_t$ on $DB$ and uses only finitely many messages.

**Proof:** Let $\hat{M}$ be the (finite) set of all and only messages $\hat{M} = \{(i : s) \mid i \in \{1, \ldots, n\}, s \in S_t\}$ that local orchestrators can broadcast.

First of all, we define an auxiliary function $m_C : \mathcal{H}^+_{\hat{M}, S_t} \rightarrow \mathcal{H}_C$ which, given an $S_t$ extended local history with messages from $\hat{M}$:

- if $h^+_{\hat{M}, S_t} = (s_{i0}, d_0, \emptyset)$, returns $m_C(h^+_{\hat{M}, S_t}) = (s_{C0}, d_0)$ for all $i = 1, \ldots, n$;

- otherwise, for $h^+_{\hat{M}, S_t} = (s_{i0}, d_0, \emptyset) \xrightarrow{O^1, Q^1} \ldots \xrightarrow{O^\ell, Q^\ell} (s_{\ell i}, d^\ell, M^\ell)$, where $M^\ell = \{(1 : s^1_1), \ldots, (n : s^1_n)\}$, returns the sequence $m_C(h^+_{\hat{M}, S_t}) = (s_{C0}, d_0) \xrightarrow{Q^1} \ldots \xrightarrow{Q^\ell} (s_{C\ell}^\ell, d^\ell)$, where $s_{C}^\ell = (s^1_1, \ldots, s^\ell_n)$
\(1 \leq j \leq \ell\), \(s_{C0} = (s_{10}, \ldots, s_{n0})\) and \(\vec{O}_j = P((s_{C0}, d_0) \xrightarrow{\vec{O}_1}, \ldots, \xrightarrow{\vec{O}_{j-1}} (s_{C_{j-1}}^{-1}, d^{-1}), O^j))\).

Clearly, \(\vec{O}_j\) definition is recursive and, more importantly, it is not well-defined in general. More precisely, it depends on the sequence of message sets \(M^j\) appearing along \(h^+_{M,S_i}\), which is completely arbitrary, as Definition 7.2.2 puts no constraint on the messages an extended local history may contain. As a consequence, \(P((s_{C0}, d_0) \xrightarrow{\vec{O}_1}, \ldots, \xrightarrow{\vec{O}_{j-1}} (s_{C_{j-1}}^{-1}, d^{-1}), O^j))\) might be not defined and so would \(\vec{O}_j\). However, we will build and use only “safe” extended local histories \(h^+_{M,S_i}\), i.e., those such that \(m_C(h^+_{M,S_i})\) is well-defined and, in particular, represents a usual community history.

Intuitively, \(m_C()\) is a function that exploits all the information locally available to reconstruct the whole community history up to a given point. Of course, the possibility to do this depends strongly on the information that messages contain.

Now, based on \(m_C()\) and the given centralized orchestrator \(P\), for each service \(S_i\), we inductively define a local orchestrator \(P^L_i = (P, B_i)\) as follows:

- let \(\mathcal{H}^0_{M,S_i} = \{\langle s_0^i, d^0, M^0 \rangle\}\) be a set of extended local histories with messages;
- for each \(\ell\)-length extended local history \(h^\ell_{M,S_i} \in \mathcal{H}^\ell_{M,S_i}\), we define:
  
  \[P_i(h^\ell_{M,S_i}, O^{\ell+1}) = Q^{\ell+1}_i, \text{ for all } O^{\ell+1} \in \mathcal{P}_0\]
  such that \(P(m_C(h^\ell_{M,S_i}), O^{\ell+1})\) is defined, where \(Q^{\ell+1}_i, \ldots, Q^{n+1}_i = P(m_C(h^\ell_{M,S_i}), O^{\ell+1})\) is the centralized orchestrator operation assignment;
  
  \[B_i(h^\ell_{M,S_i}, \ell+1, s_{i+1}) = \{\langle i : s_{i+1}^{\ell+1} \rangle\}, \text{ for } O^{\ell+1} \text{ as above and for each } s_{i+1}^{\ell+1} \in S_i \text{ such that } s_{i+1}^{\ell+1} \text{ is in } S_i, \text{ with } Q^{\ell+1}_i \text{ as above, for some } g \in G_i \text{ such that } g(d^\ell) = T, \text{ where } last(h^\ell_{M,S_i}) = (s^\ell, d^\ell, M^\ell); \]
  
  \[h^\ell_{M,S_i}, O^{\ell+1}, Q^{\ell+1}_i, s_{i+1}^{\ell+1} \in \mathcal{H}^{\ell+1}_{M,S_i}, \text{ for all } O^{\ell+1} \text{ and } s_{i+1}^{\ell+1} \text{ such that } B_i(h^\ell_{M,S_i}, O^{\ell+1}, s_{i+1}^{\ell+1}) \text{ is defined as above, } Q^{\ell+1}_i = P_i(h^\ell_{M,S_i}, O^{\ell+1}), \text{ all transitions } d^{\ell+1} \text{ in } DB \text{ and } M^{\ell+1} = \{B_i(h^\ell_{M,S_i}, O^{\ell+1}, s_{i+1}^{\ell+1}), \ldots, B_n(h^\ell_{M,S_i}, O^{\ell+1}, s_{n+1}^{\ell+1})\}, \text{ for all combinations of } s_{j+1}^{\ell+1} (j \neq i) \text{ such that each } B_j(h^\ell_{M,S_j}, O^{\ell+1}, s_{j+1}^{\ell+1}) (j \neq i) \text{ is defined and } s_{i+1}^{\ell+1} \text{ the same as above};\]
The idea behind each local orchestrator is that it replicates the centralized orchestrator behavior by exploiting the received messages to reconstruct the community state and, thus, by delegating to its respective service the same operation sets as the centralized orchestrator, if all available services’ state information were accessible.

We need to show that the so-defined distributed orchestrator \( DP = \{ P^L_1, \ldots, P^L_n \} \) realizes the target service. To this end, consider a generic target service trace (with no messages) \( \tau = (s_0, d^0) \xrightarrow{O_1} (s^1, d^1) \xrightarrow{O_2} \ldots \), and refer to Definition 7.2.7. We show by induction on the length of \( H_{\mathcal{M}, \tau} = \bigcup_i H_{\mathcal{M}, \tau}^i \) histories, defined as in Definition 7.2.7, that \( DP \) realizes \( \tau \).

- For the base case, consider \( H_{\mathcal{M}, \tau}^0 = \{ h_{\mathcal{M}, \tau}^0 \} = \{ (s_0, d_0, \emptyset) \} \). Then, \( h_{\mathcal{M}, \tau}^0 |_i = (s_0, d_0, \emptyset) \) is the projected \( S_i \) extended local history of \( h_{\mathcal{M}, \tau}^0 \) (local histories represent all information about the whole community that a local orchestrator can actually access). The following holds:

  - \( m_C(h_{\mathcal{M}, \tau}^0 |_1) = \ldots = m_C(h_{\mathcal{M}, \tau}^0 |_n) = h^\tau_r \);
  - since \( P \) is a composition, \( P(h_{\mathcal{M}, \tau}^0, O^1) \) is defined and, in particular, \( P(h_{\mathcal{M}, \tau}^0, O^1) = (Q^1_1, \ldots, Q^1_n) \);
  - by definition, for all \( i \in \{1, \ldots, n\} \), \( P_i(h_{\mathcal{M}, \tau}^0 |_i, O^1) = Q^1_i \);
  - by definition of composition \( \bigcup_{i=1}^n P_i(h_{\mathcal{M}, \tau}^0 |_i, O^1) = \bigcup_{i=1}^n Q^1_i = O^1 \);
  - by definition, all \( B_i(h_{\mathcal{M}, \tau}^0 |_i, Q^1_i, s^1_i) \) are defined for all transitions \( s_{i0} \xrightarrow{gO^1} s^1_i \) in \( S_i \) for some \( g \in G_i \) such that \( g(d_0) = \top \);

- for the induction step, assume that for all \( \ell \)-length histories in \( h_{\mathcal{M}, \tau}^\ell \in H_{\mathcal{M}, \tau}^\ell \), the following holds:

  1. \( m_C(h_{\mathcal{M}, \tau}^\ell |_1) = \ldots = m_C(h_{\mathcal{M}, \tau}^\ell |_n) = h^\tau_r \);
  2. \( P(h_{\mathcal{M}, \tau}^\ell, O^{\ell+1}) \) is defined and \( P(h_{\mathcal{M}, \tau}^\ell, O^{\ell+1}) = (Q^{\ell+1}_1, \ldots, Q^{\ell+1}_n) \);
  3. all \( P_i(h_{\mathcal{M}, \tau}^\ell |_i, O^{\ell+1}) \) are defined and \( P_i(h_{\mathcal{M}, \tau}^\ell |_i, O^{\ell+1}) = Q^{\ell+1}_i \);
  4. \( \bigcup_{i=1}^n P_i(h_{\mathcal{M}, \tau}^\ell |_i, O^{\ell+1}) = O^{\ell+1} \);
  5. \( B_i(h_{\mathcal{M}, \tau}^\ell |_i, Q^{\ell+1}_i, s^{\ell+1}_i) \) are defined for all transitions \( s^\ell_i \xrightarrow{gO^{\ell+1}} s^{\ell+1}_i \) in \( S_i \) for some \( g \in G_i \) such that \( g(d^\ell) = \top \), where \( \text{last}(h_{\mathcal{M}, \tau}^\ell |_i) = (s^\ell_i, d^\ell, M^\ell) \).
7.3. LOCAL ORCHESTRATOR SYNTHESIS

Let \( \text{last}(h^\ell_{M,\tau}) = (s^\ell_C, d^\ell, M^\ell) \). By 2., 3. and 5., we have that (i) for each \( O^{\ell+1} \) such that \( P(h^\ell_C, O^{\ell+1}) \) is defined, (ii) for \( O^{\ell+1} = \langle P_1(h^\ell_{M,\tau}, 1), O^{\ell+1}, \ldots, P_n(h^\ell_{M,\tau}, n), O^{\ell+1} \rangle = \langle Q_1^{\ell+1}, \ldots, Q_n^{\ell+1} \rangle = P(h^\ell_{\tau}, O^{\ell+1}) \), (iii) for all \( s^\ell_C = \langle s^\ell_{C1}, \ldots, s^\ell_{Cn} \rangle \) such that, for \( i = 1, \ldots, n \), \( s^\ell_i \xrightarrow{g^{\ell+1}} s^{\ell+1}_i \) for some \( g \in G_i \) such that \( g(d^\ell) = \top \), \( h^{\ell+1}_{M,\tau} \xrightarrow{O^{\ell+1}} \langle s^{\ell+1}_C, d^{\ell+1}, M^{\ell+1} \rangle \in H^{\ell+1}_{M,\tau} \) is a community history with messages, where \( M^{\ell+1} = \bigcup_{i=1}^n B_i(h^{\ell+1}_{M,\tau}, i, O^{\ell+1}, s^{\ell+1}_i) \). Correspondingly, local extended histories have the form \( h^{\ell+1}_C \mid_i = h^{\ell}_{M,\tau} \xrightarrow{O^{\ell+1}} \langle s^{\ell+1}_C, d^{\ell+1}, M^{\ell+1} \rangle \). Therefore, since \( m_C(h^{\ell+1}_{M,\tau} \mid_1) = \ldots = m_C(h^{\ell+1}_{M,\tau} \mid_n) \) and all \( M^{\ell+1} \) are the same for all local histories, we get that \( m_C(h^{\ell+1}_{M,\tau} \mid_1) = \ldots = m_C(h^{\ell+1}_{M,\tau} \mid_n) \) so property 1. is satisfied also for \( (\ell + 1) \)-length histories in \( H_{M,\tau} \).

Now, observe that \( h^{\ell+1}_C = h^{\ell}_{\tau} \xrightarrow{O^{\ell+1}} \langle s^{\ell+1}_C, d^{\ell+1} \rangle \), for all \( O^{\ell+1}, \bar{O}^{\ell+1} \) and \( s^{\ell+1}_C \) as above. But then, since \( P \) is a composition, \( P(h^{\ell+1}_C, O^{\ell+2}) \) is defined. From this, it is easily seen that also properties 2.-5. are satisfied for all \( (\ell + 1) \)-length histories in \( H_{M,\tau} \).

Proving that each time the target service reaches a final state the community also does is straightforwardly done by observing that this is true for \( P \), which is a composition by hypothesis. \( \square \)

The construction above provides an actual procedure for building a distributed orchestrator, from a centralized one. Essentially, it consists in defining a set of local controllers, each behaving in the same way as the centralized one except that it (i) reconstructs the community history based on the received messages and (ii) broadcasts, step by step, a message that allows other orchestrator to reconstruct the history executed by the community. More precisely, at each step:

- based on the local history with messages, a local orchestrator reconstruct the current community history and, considering the operation set currently requested (and broadcasted to all available services, i.e., local orchestrators), delegates to its available service the same operation set that the centralized orchestrator would delegate under the same (reconstructed) history and current request;

- after the attached service has executed the delegated operations, the local orchestrator (which has full observability on service state) broadcasts a message corresponding to the state reached by the service.
Interestingly, we can show that distributed and centralized orchestrators have the same power, that is, a distributed orchestrator realizing a target service exists if and only if there exists a centralized orchestrator that realizes the same target service. It is shown by proving the viceversa of the Theorem 7.3.1.

**Theorem 7.3.2** Let $C = \{S_1, \ldots, S_n, DB\}$ be a community of distributed available services with multiple operations, $S_i$ a target service and $M$ a set of messages. If there exists a distributed orchestrator $DP = \{P^L_1, \ldots, P^L_n\}$ that realizes $S_i$ on $DB$ then there exists a centralized orchestrator $P$ for $S_C$ that realizes $S_i$ on $DB$.

**Proof:** Given an $\ell$-length community history with messages $h^\ell_M = ⟨s_0, d_0, \emptyset⟩ \xrightarrow{\delta^1} \cdots \xrightarrow{\delta^\ell} ⟨s^\ell, d^\ell, M^\ell⟩$, let $h^\ell_M|C = ⟨s_0, d_0⟩ \xrightarrow{\delta^1} \cdots \xrightarrow{\delta^\ell} ⟨s^\ell, d^\ell⟩$ be the result of projecting out all messages from $h^\ell_M$. By definition of community history with messages, $h^\ell_M|C$ is a community history (without messages). We inductively define the centralized orchestrator $P$, as follows.

- Let $H^0_M = \{⟨s^0_C, d^0, \emptyset⟩\}$ be the set of 0-length community histories with messages;
- for all $\ell$-length community histories with messages $h^\ell_M \in H^\ell_M$, we define:
  - $P(h^\ell_M|C, O) = \{P_1(h^\ell_M|1, O), \ldots, P_n(h^\ell_M|n, O)\} = \{Q_1, \ldots, Q_n\} = \overrightarrow{O}$, for all $O \in 2^O$ such that all $P_i(h^\ell_M|i, O)$ are defined ($i = 1, \ldots, n$);
  - $h^{\ell+1}_M = h^\ell_M \xrightarrow{\delta} ⟨s^1_C, d^1, M^{\ell+1}⟩ \in H^{\ell+1}_M$, (i) for all $s^{\ell+1}_C = ⟨s^1_C, \ldots, s^{\ell+1}_C⟩$ such that for all $i = 1, \ldots, n$, $s^\ell_i \xrightarrow{gQ_i} s^{\ell+1}_i$ is in $S_i$ for some $g \in G_i$ such that $g(d^\ell) = \top$, where $\text{last}(h^\ell_M) = ⟨s^\ell_C, d^\ell⟩$ and $s^\ell_C = ⟨s^\ell_1, \ldots, s^\ell_n⟩$, (ii) for all $d^{\ell+1} \in D$ such that $d^\ell \xrightarrow{O} d^{\ell+1}$ in $DB$, (iii) where $M^{\ell+1} = \cup_{i=1}^n B_i(h^\ell_M|i, O, s^{\ell+1}_i)$.

The fact that the so-obtained orchestrator is a composition is an immediate consequence of the fact that $DP$ is a composition. \qed

Theorems 7.3.1 and 7.3.2 can be merged into the following result:

**Theorem 7.3.3** Let $C = \{S_1, \ldots, S_n, DB\}$ be a community of distributed available services with multiple operations, $S_i$ a target service and $M$ a set of messages. A distributed orchestrator $DP = \{P^L_1, \ldots, P^L_n\}$ that realizes $S_i$ on $DB$ exists if and only if a centralized orchestrator $P$ that realizes $S_i$ on $DB$ exists.

Based on this Theorem, all results obtained for centralized orchestrator can be transferred to distributed ones.
7.4 Discussion

So far, we are able to distribute a given, pre-computed orchestrator. Differently from previous scenarios, the orchestration does not rely on a generator that, step by step, provides some choices for operation delegation, thus allowing for building just-in-time compositions. Clearly, this affects the possibility to build robust compositions, in the sense of Chapter 5.

One might think of adopting an approach similar to the one seen so far, and equipping each distributed service with a local composition generator, rather than a local orchestrator, which is able to send and receive messages. Supposing that the obstacle of state reconstruction is overcome by suitably exploiting messages, when doing so, the problem of finding an agreement among all distributed generators arises. For instance, consider a situation where there are two available services $S_1$ and $S_2$ and that, at a given point, a set of operations $O = \{o_1, o_2\}$ (compliant with the target service) is requested. Assume that both $S_1$ and $S_2$ are capable, in the state they are currently in, of executing either $o_1$ or $o_2$ (but not both at the same time). Clearly, there are two possible “successful” delegations: one where $S_1$ executes $o_1$ and $S_2$ executes $o_2$ and another one where operations are switched. How can both services (actually the attached orchestrators) agree on which delegation is to be chosen? Observe that services cannot make their decisions independently of the other while guaranteeing a correct global decision. Moreover, even though they can exchange messages, there is no trivial way to properly synchronize, through the use of messages, the decisions made by each service, unless additional requirements are met, such as the existence of a pre-assigned priority in delegation choice.

Investigating over the techniques for finding an agreement among distributed services is out of the scope of this work. However, it is interesting noticing that if we assign a priority to services, where the one with highest priority is the first to make the decision, and we assume that each decision can be broadcasted to other services through the use of messages, then also generators can be distributed. For instance, in the above example, if service $S_2$ has highest priority and chooses operation $o_1$, then service $S_1$, after receiving $S_2$ message, which brings information about the chosen operation, can only decide to execute operation $o_2$, thus covering the requested operation set $O$. Clearly, this “draft” solution yields new issues. For example, if service $S_2$ stops working, $S_1$ waits indefinitely for its message, before making its decision. Smarter techniques for operation delegation and the possibility to build robust compositions can be object of future work directions.
Chapter 8
Composition and Safety Games

The theoretical frameworks developed in previous chapters require, in order to solve the service composition problem, the computation of a simulation-like relation between the target and the community TS’s. So far, we have shown how the problem can be modeled by providing a behavioral description of the entities involved and, importantly, how from a simulation-like relation the whole set of solutions can be obtained (except for the distributed scenario of Chapter 7). However, no direction has been provided about how one can compute a composition generator. This Chapter focuses on such issue. In particular, we refer to the framework of Chapter 4 and show that the service composition problem can be reduced to the problem of finding a winning strategy for a player in a game structure ([10, 11, 122]). Importantly, this corresponds to computing a maximal ND-simulation relation, thus obtaining the whole set of solutions. Finally, we will show how the presented technique can be easily extended to deal with multiple operations, thus also covering the scenario described in Chapter 6.

8.1 Simulation and Safety Games

In this Section, we show how a service composition problem instance can be encoded into an equivalent game automaton and, correspondingly, how searching for a composition is equivalent to searching for a winning strategy (cf. [10, 11, 122]). The main motivation behind this approach is the availability of software systems, such as TLV [124], Lily [84], Anzu [85] or MOCHA [6], which provide (i) efficient procedures for strategy computation and (ii) convenient languages for representing the problem instance in a modular, intuitive and pretty straightforward way.
8.1.1 Safety-Game structures

Here, we specialize the game structures introduced in [122], used to represent and solve reactive system synthesis problems with invariant properties, to cope with our problem. Even though game structures have been studied also in other work (cf., e.g., [10, 11]), in this context we refer to [122] because the formalism introduced allows for a straightforward understanding of the implementation we will present later on. After introducing the specialization, we show how game structures can be used to describe a service composition problem.

Definition 8.1.1 A safety-game structure (or \(\Box\)-game structure or \(\Box\)-GS, for short) is a tuple \(G = (V, X, Y, \Theta, \rho_e, \rho_s, \Box \varphi)\), where:

- \(V = \{v_1, \ldots, v_n\}\) is a finite set of state variables, ranging over finite domains \(V_1, \ldots, V_n\), respectively. A valuation of all variables in \(V\) is a tuple \(u = \langle u_1, \ldots, u_n \rangle \in V = V_1 \times \ldots \times V_n\), associating value \(u_i\) to variable \(v_i\) \((i = 1, \ldots, n)\).

- \(X \subseteq V\) is the set of environment variables. Without loss of generality, we assume that \(X = \{v_1, \ldots, v_m\}\) \((m \leq n)\). An environment state is a valuation of the variables in \(X\), that is, a tuple \(\vec{x} = \langle x_1, \ldots, x_m \rangle \in X = V_1 \times \ldots \times V_m\), associating value \(x_i\) to environment variable \(v_i\) \((i = 1, \ldots, m)\).

- \(Y = V/X\) is the set of system variables. Without loss of generality, we assume that \(Y = \{v_{m+1}, \ldots, v_n\}\). A system state is a valuation of the variables in \(Y\), that is, a tuple \(\vec{y} = \langle y_{m+1}, \ldots, y_n \rangle \in Y = V_{m+1} \times \ldots \times V_n\), associating value \(y_i\) to system variable \(v_i\) \((i = m+1, \ldots, n)\).

- \(\Theta\) is a formula representing the initial game states, where a game state is a tuple \(\vec{s} = \langle \vec{x}, \vec{y} \rangle \in X \times Y = V\). In details, \(\Theta\) is a boolean formula without negation symbols, whose atoms are expressions of the form \((v_k = u)\), where \(k \in \{1, \ldots, n\}\), \(v_k \in V\) and \(u \in V_k\). Given a state \(\vec{s} = \langle \vec{x}, \vec{y} \rangle \in V\), we write \(\vec{s} \models \Theta\) (\(\vec{s}\) satisfies \(\Theta\)) if and only if \(\Theta\) evaluates to \(\top\), once each of its atoms \((v_k = u)\) is replaced by (i) \(\top\), if \(k\)-th component of \(\vec{s}\) assumes value \(u\), and (ii) \(\bot\), otherwise.

- \(\rho_e \subseteq X \times Y \times X\) is the environment transition relation which relates a current game state to a possible next environment state. In particular, if \(\langle \vec{x}, \vec{y}, \vec{x}' \rangle \in \rho_e\), or, equivalently, \(\rho_e(\vec{x}, \vec{y}, \vec{x}')\), then \(\langle \vec{x}, \vec{y} \rangle\) is referred to as the current game state and \(\vec{x}'\) as the next environment state.

- \(\rho_s \subseteq X \times Y \times X \times Y\) is the system transition relation, which relates a current game state and a (next) environment state to a possible next
system state. If \( \langle \vec{x}, \vec{y}, \vec{x}', \vec{y}' \rangle \in \rho_s \), or, equivalently, \( \rho_s(\vec{x}, \vec{y}, \vec{x}', \vec{y}') \), then \( \langle \vec{x}, \vec{y} \rangle \) is referred to as the current game state, \( \vec{x}' \) as the next environment state and \( \vec{y}' \) as the next system state;

- \( \Box \varphi \) is a formula that represents the invariant property to be guaranteed. In particular, \( \varphi \) has the same form as \( \Theta \) and satisfaction definition is as above.

Intuitively, safety-game structures represent a game played by two adversaries: system and environment. A round is composed of two moves: an environment’s followed by a system’s, the former being a valuation of state variables and the latter being a valuation of system variables. Observe that, since state variables are, in fact, partitioned into system’s and environment’s, at the end of each round a game state is fully defined. Moreover, as state variables range over finite domains, game states are finite. Initially, the game can be in any (initial) state that satisfies \( \Theta \), according to above Definition 8.1.1. Players are not allowed to choose all possible valuations, but, assuming that, when a round starts, the game is in state \( \vec{s} = (\vec{x}, \vec{y}) \), must respect the following constraints:

- environment can choose any valuation \( \vec{x}' \in X \) of environment variables such that \( \rho_e(\vec{x}, \vec{y}, \vec{x}') \) holds;
- system can choose any valuation \( \vec{y}' \in Y \) of system variables such that \( \rho_s(\vec{x}, \vec{y}, \vec{x}', \vec{y}') \) holds, provided \( \vec{x}' \) corresponds to current round environment’s move.

This is formalized by defining when a game state is a successor of another game state.

**Definition 8.1.2** A game state \( \langle \vec{x}', \vec{y}' \rangle \) is a successor of \( \langle \vec{x}, \vec{y} \rangle \), or, equivalently \( \langle \vec{x}, \vec{y} \rangle \rightarrow \langle \vec{x}', \vec{y}' \rangle \), if and only if both \( \rho_e(\vec{x}, \vec{y}, \vec{x}') \) and \( \rho_s(\vec{x}, \vec{y}, \vec{x}', \vec{y}') \) hold.

System goal is to keep the game going while maintaining \( \varphi \) satisfied, whereas environment goal is to lead the game to a state such that \( \varphi \) is not satisfied. Notice that the system can observe environment’s move before performing its move.

Let us rephrase such intuition by means of formal notions.

**Definition 8.1.3** Given a \( \Box \)-GS \( G \) as above, a play of \( G \) is a maximal sequence of \( G \) states \( \eta = \langle \vec{x}_0, \vec{y}_0 \rangle \langle \vec{x}_1, \vec{y}_1 \rangle \cdots \) such that:

- \( \langle \vec{x}_0, \vec{y}_0 \rangle \models \Theta \), and
- \( \langle \vec{x}_j, \vec{y}_j \rangle \rightarrow \langle \vec{x}_{j+1}, \vec{y}_{j+1} \rangle \), for each \( j \geq 0 \).
As it comes out from Definition 8.1.2, a play is, informally, a sequence of game states compliant with the “rules of the game”. We remark that, in general, there are many, though finite, initial states, namely all those that satisfy \( \Theta \).

The notion of play can be generalized to represent sequences of moves that start from any state:

**Definition 8.1.4** Given a \( \square \)-GS \( G \) and a game state \( \langle \vec{x}, \vec{y} \rangle \in V \), a \( \langle \vec{x}, \vec{y} \rangle \)-play of \( G \) is a maximal sequence of \( G \) states
\[
\eta = \langle \vec{x}, \vec{y} \rangle \langle \vec{x}_1, \vec{y}_1 \rangle \cdots
\]
such that
\[
\langle \vec{x}, \vec{y} \rangle \xrightarrow{} \langle \vec{x}_1, \vec{y}_1 \rangle \text{ and } \langle \vec{x}_j, \vec{y}_j \rangle \xrightarrow{} \langle \vec{x}_{j+1}, \vec{y}_{j+1} \rangle, \text{ for } j \geq 1.
\]

Finally, we define when a play is winning for the environment and for the system.

**Definition 8.1.5** Let \( G \) be a \( \square \)-GS as above. A \( \langle \vec{x}, \vec{y} \rangle \)-play \( \eta = \langle \vec{x}, \vec{y} \rangle \langle \vec{x}_1, \vec{y}_1 \rangle \cdots \) is said to be winning for the system if and only if:

- it is infinite, and
- satisfies the winning condition \( \square \varphi \), that is, \( \langle \vec{x}, \vec{y} \rangle \models \varphi \) and \( \langle \vec{x}_j, \vec{y}_j \rangle \models \varphi \), for each \( j \geq 1 \).

Otherwise, it is winning for the environment.

A fundamental question about safety games is whether, given a game structure, the system has a winning strategy, that is, if, no matter how the environment plays, it is always able to keep the game going while satisfying \( \varphi \). The following definitions provide a formalization of this notion.

**Definition 8.1.6** Let \( G \) be a \( \square \)-GS as above. A strategy for the system is a partial function \( f : (X \times Y)^+ \times X \rightarrow Y \) such that \( \lambda = \langle \vec{x}_0, \vec{y}_0 \rangle \cdots \langle \vec{x}_n, \vec{y}_n \rangle \) is a finite sequence of game states then for all \( \vec{x}' \) such that \( \rho_e(\vec{x}_n, \vec{y}_n, \vec{x}') \), \( f(\lambda, \vec{x}') \) is defined and \( \rho_s(\vec{x}_n, \vec{y}_n, \vec{x}', f(\lambda, \vec{x}')). \)

**Definition 8.1.7** Let \( G \) be a \( \square \)-GS and \( f() \) a strategy for the system. A \( \langle \vec{x}_0, \vec{y}_0 \rangle \)-play \( \eta = \langle \vec{x}_0, \vec{y}_0 \rangle \langle \vec{x}_1, \vec{y}_1 \rangle \cdots \) is said to be compliant with \( f() \) if and only if, for all \( i \geq 0 \), \( f(\langle \vec{x}_0, \vec{y}_0 \rangle \cdots \langle \vec{x}_i, \vec{y}_i \rangle, \vec{x}_{i+1}) = \vec{y}_{i+1} \).

**Definition 8.1.8** Given a \( \square \)-GS \( G \) as above, a (system) strategy \( f() \) is winning for the system from state \( \langle \vec{x}, \vec{y} \rangle \), if and only if all \( \langle \vec{x}, \vec{y} \rangle \)-plays compliant with \( f() \) are so.

If, from a state \( \langle \vec{x}, \vec{y} \rangle \), there exists a strategy \( f \) winning for the system, then we expect that, from all possible successors \( \langle \vec{x}', \vec{y}' \rangle \) obtained through \( f() \), a winning strategy for the system exists. This observation suggests the following co-inductive definition:
8.1. SIMULATION AND SAFETY GAMES

Definition 8.1.9 Let $G$ be a $\Box$-GS as above. A set $\tilde{W} \subseteq V$ of game states is said to be winning (for the system) if and only if for each state $\langle \bar{x}, \bar{y} \rangle \in \tilde{W}$:

1. $\langle \bar{x}, \bar{y} \rangle \models \varphi$;

2. for each $\bar{x}' \in X$ such that $\rho_e(\bar{x}, \bar{y}, \bar{x}')$ there exists a $\bar{y}' \in Y$ such that:
   - $\rho_s(\bar{x}, \bar{y}, \bar{x}', \bar{y}')$;
   - $\langle \bar{x}', \bar{y}' \rangle \in \tilde{W}$.

Definition 8.1.10 Given a $\Box$-GS $G$, a game state is winning (for the system) if and only if it is contained in some winning set.

Let $W \subseteq V$ be the set of all winning states for a $\Box$-game $G$. Clearly, $W$ is itself a winning set and, in particular, is the largest winning set of $G$.

Definition 8.1.11 A $\Box$-GS is said to be winning for the system if all initial states are so. Otherwise, it is said to be winning for the environment.

Next, we show how, given a generic $\Box$-GS, the set of all and only states that are winning for the system can be computed. To this end, we introduce the following operator (cf. [10, 122]):

Definition 8.1.12 Let $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box \varphi \rangle$ be a $\Box$-GS as above. Given a set $P \subseteq V$ of game states, the set of $P$'s controllable predecessors is

$$
\pi(P) = \{ \langle \bar{x}, \bar{y} \rangle \in V \mid \forall \bar{x}' \rho_e(\bar{x}, \bar{y}, \bar{x}') \rightarrow \exists \bar{y}' \rho_s(\bar{x}, \bar{y}, \bar{x}', \bar{y}') \wedge \langle \bar{x}', \bar{y}' \rangle \in P \}
$$

Controllable predecessors have the following property.

Lemma 8.1.1 Consider the set of states $V$ of a $\Box$-GS $G$ and two subsets $P \subseteq V$ and $P' \subseteq V$ such that $P' \subseteq P$. Then $\pi(P') \subseteq \pi(P)$.

Proof: Immediate consequence of Definition 8.1.12.

Intuitively, $\pi(P)$ is the set of states from which the system can force the game to reach a state in $P$, no matter how the environment evolves. Clearly, $\pi(\pi(P))$ is the set of states from which the system can force the game to reach a state in $P$ after two moves, and so on for $\pi(\pi(\cdots \pi(P) \cdots))$: by iteratively applying operator $\pi$, one computes the set of game states from which the system can force the game to reach a state in $P$ in a finite set of moves. This procedure is the intuition behind Algorithm 4 which, as Theorem 8.1.2 shows, computes the maximal set of system winning states for the system, in a given $\Box$-GS $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \Box \varphi \rangle$. 

Algorithm 4 \textit{MAX\_WIN} – Computes the maximal set of winning states for the system, in a $\square$-GS

1: $W := \{\langle \vec{x}, \vec{y} \rangle \in V \mid \langle \vec{x}, \vec{y} \rangle \models \varphi \}$
2: \textbf{repeat}
3: \quad $W' := W$
4: \quad $W := W \cap \pi(W)$
5: \textbf{until} ($W' = W$)
6: \textbf{return} $W$

Theorem 8.1.1 Let $G = \langle V, X, Y, \Theta, \rho_e, \rho_s, \square \varphi \rangle$ be a $\square$-GS as above. Algorithm 4 terminates in a finite number of steps and returns the maximal winning set $W$ for $G$.

\textbf{Proof:}

- To see that the algorithm terminates, observe that $W$ is never expanded, as its version at $(i + 1)$-th iteration corresponds to the intersection of its version at $i$-th iteration with another set, namely $\pi(W)$. Consequently, the result is, by definition of intersection, contained in (possibly the same, and in such case the algorithm returns, as) $W$;

- for each $\langle \vec{x}, \vec{y} \rangle \in W$, $\langle \vec{x}, \vec{y} \rangle \models \varphi$ comes from the initialization of $W$ and the fact that $W$ is never expanded;

- by algorithm construction, $W \subseteq \pi(W)$, which corresponds to meet requirement 2 of Definition 8.1.9;

- maximality comes from $W$ initialization, which clearly includes all winning states (but not only), and the use of operator $\pi$, which ensures that requirement 2 of Definition 8.1.9 is locally met by all states kept in $W$. Since only states not satisfying such requirement are discarded, no good solution is lost.

Now, we show two lemmas, that will be useful in showing forthcoming Theorem 8.1.2.

Lemma 8.1.2 Let $W^0$ represent the set $W$ after first initialization of Algorithm 4 and $W^i$ represent $W$ at the end of $i$-th repeat iteration ($i = 1$ for first iteration), i.e., at line 4. Consider a $\square$-GS $G$ as above and assume that, with $G$ as input, Algorithm 4 executes $N$ iterations before returning. For $0 < i \leq N$, $W^i = \pi(W^{i-1}) \cap W^0$. 
8.1. SIMULATION AND SAFETY GAMES

Proof: First, observe that, by Algorithm 4 construction, \( W^i \subseteq W^{i-1} \). We proceed by induction:

- For \( i = 1 \), trivially: \( W^1 = \pi(W^0) \cap W^0 \).
- Let \( i = m \), assume that \( W^m = \pi(W^{m-1}) \cap W^0 \) and consider \((\dagger)\) \( W^{m+1} = \pi(W^m) \cap W^m \). Since, by induction hypothesis, \( W^m = \pi(W^{m-1}) \cap W^0 \), by substituting in \((\dagger)\), we obtain \( W^{m+1} = \pi(W^m) \cap \pi(W^{m-1}) \cap W^0 \). Since \( W^m \subseteq W^{m-1} \), by Lemma 8.1.1, \( \pi(W^m) \subseteq \pi(W^{m-1}) \), hence \( \pi(W^m) \cap \pi(W^{m-1}) = \pi(W^m) \). Therefore:

\[
W^{m+1} = \pi(W^m) \cap \pi(W^{m-1}) \cap W^0 = \pi(W^m) \cap W^0.
\]

\( \square \)

Lemma 8.1.3 Let \( G \), \( W^0 \) and \( W^i \) be as above and assume that, with \( G \) as input, Algorithm 4 executes \( N \) iterations before returning. Consider a game state \( s_0 = \langle x_0, y_0 \rangle \) such that \( s_0 \Vdash \varphi \) and let \( k \in \{1, \ldots, N\} \) be the minimal index such that \( s_0 \notin W^k \). Then, \( \exists x_1 \forall y_1 \ldots \exists x_k \forall y_k \) such that, if sequence \( s_0 \ldots s_k \), where \( s_i = \langle x_i, y_i \rangle \) \( (i = 0, \ldots, k) \), is such that \( s_{i-1} \rightarrow s_i \) and \( s_{i-1} \Vdash \varphi \) for \( i = 1, \ldots, k \), then \( s_k \not\Vdash \varphi \).

Proof: Assume that \( s_0 \Vdash \varphi \), \( s_0 \notin W^k \), for \( k \) minimal, and that, by contradiction, \((\dagger)\) \( \forall x_1 \exists y_1 \ldots \forall x_k \exists y_k \) such that sequence \( s_0 \ldots s_k \) is such that \( s_{i-1} \rightarrow s_i \), \( s_{i-1} \Vdash \varphi \) and \( s_k \Vdash \varphi \). Consider such a sequence \( s_0 \ldots s_k \). We show, by induction, that \( s_{k-i} \in W^i \) for \( i = 0, \ldots, k \), hence \( s_0 \in W^k \), i.e., a contradiction:

- for \( i = 0 \), since \( s_k \Vdash \varphi \), \( s_k \in W^0 \);
- for \( i = m < k \), assume that \( s_{k-m} \in W^m \) and consider state \( s_{k-m-1} \). We have that \( s_{k-m-1} \in \pi(W^m) \), as, by \((\dagger)\), \( \forall x_{k-m} \exists y_{k-m} \) such that \( s_{k-m} \) is a successor of \( s_{k-m-1} \) which, by induction hypothesis, is in \( W^m \). Moreover, since \( s_{k-m-1} \Vdash \varphi \), also \( s_{k-m-1} \in W^0 \) holds. But then \( s_{k-m-1} \in W^0 \) and \( s_{k-m-1} \in \pi(W^m) \), so, by Lemma 8.1.2, \( s_{k-m-1} \in W^{m+1} \).

\( \square \)

Now, we can show the following fundamental result.

Theorem 8.1.2 Let \( G = (V, X, Y, \Theta, \rho_e, \rho_s, \Box \varphi) \) be a \( \Box -GS \) as above and \( W \) obtained as in Algorithm 4. Given a state \( \langle x_0, y_0 \rangle \in V \), a winning strategy \( f() \) for the system, starting from \( \langle x_0, y_0 \rangle \), exists if and only if \( \langle x_0, y_0 \rangle \in W \).

Proof: (If Part.) By Theorem 8.1.1, \( W \subseteq \pi(W) \). So, by Definition 8.1.12: \((\dagger)\) for each \( \langle x, y \rangle \in W \) and for each \( x' \in X \) such that \( \rho_e(x, y, x') \), there exists a \( y' \in Y \)
such that $\rho_s(\vec{x}, \vec{y}, \vec{x}', \vec{y}')$ and $\langle \vec{x}', \vec{y}' \rangle \in W$. Moreover, by construction of $W$ in Algorithm 4: $(\dagger)$ for each $\langle \vec{x}, \vec{y} \rangle \in W$, $\langle \vec{x}, \vec{y} \rangle \models \varphi$. Now, let $\Lambda^i \subseteq (X \times Y)^{i+1}$, for $i \geq 0$. We define, by induction, a function $f : (X \times Y)^+ \times X \rightarrow Y$ and sets $\Lambda^i$ as follows:

- $\Lambda^0 = \{\langle \vec{x}_0, \vec{y}_0 \rangle\}$. By hypothesis, $\langle \vec{x}_0, \vec{y}_0 \rangle \in W$;
- for each $\lambda \in \Lambda^\ell$, $\lambda = \langle x_0, y_0 \rangle \cdots \langle x_\ell, y_\ell \rangle$ ($\ell \geq 0$), assume that $\langle x_\ell, y_\ell \rangle \in W$, then, for each $\vec{x}_{\ell+1} \in X$ such that $\rho_e(\vec{x}_\ell, \vec{y}_\ell, \vec{x}_{\ell+1})$, choose one $\vec{y}'$ among those as in $(\ddagger)$ and let:
  - $\vec{y}_{\ell+1} = f(\lambda, \vec{x}_{\ell+1}) = \vec{y}'$;
  - $\lambda' = \langle x_0, y_0 \rangle \cdots \langle x_\ell, y_\ell \rangle \langle x_{\ell+1}, y_{\ell+1} \rangle \in \Lambda^{\ell+1}$.

Since, by $(\ddagger)$, $\langle x_{\ell+1}, y_{\ell+1} \rangle \in W$, function $f()$ is well founded.

Due to this definition, if a $\langle \vec{x}_0, \vec{y}_0 \rangle$-play $\eta = \langle \vec{x}_0, \vec{y}_0 \rangle \langle \vec{x}_1, \vec{y}_1 \rangle \cdots$ is compliant with $f()$ then it is infinite and $\langle \vec{x}_i, \vec{y}_i \rangle \in W$ for all $i \geq 0$, so, by $(\ddagger)$, $\langle \vec{x}_i, \vec{y}_i \rangle \models \varphi$. Therefore $\eta$ is winning for the system.

(Only-If Part.) Let $f()$ be a winning strategy for the system from $\langle \vec{x}_0, \vec{y}_0 \rangle$ and assume, by contradiction, that $\langle \vec{x}_0, \vec{y}_0 \rangle \notin W$. By definition of winning strategy, each play $\eta = \langle \vec{x}_0, \vec{y}_0 \rangle \langle \vec{x}_1, \vec{y}_1 \rangle \cdots$ compliant with $f()$ is winning for the system. In particular, it is infinite and for all $i \geq 0$, $\langle \vec{x}_i, \vec{y}_i \rangle \models \varphi$. Then, also $\langle \vec{x}_0, \vec{y}_0 \rangle \models \varphi$ so, when $W$ is initialized in Algorithm 4, $\langle \vec{x}_0, \vec{y}_0 \rangle \in W$. Therefore, $\langle \vec{x}_0, \vec{y}_0 \rangle$ must have been removed at some iteration, say $k$-th. But, then, Lemma 8.1.3 applies, thus preventing the existence of a winning strategy, as, for every strategy $f()$, it is always possible to find a non-winning play compliant with $f()$.

Observe that the If Part of above proof provides an actual procedure for building a strategy from a state in $W$. By above theorem, we get the following result:

**Theorem 8.1.3** Let $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \boxdot \varphi \rangle$ be a $\square$-GS as above and $W$ obtained as result of Algorithm 4. $G$ is winning for the system if and only if all of its initial states belong to $W$.

**Proof:** Direct consequence of Theorem 8.1.2.
8.1. SIMULATION AND SAFETY GAMES

- check whether (†) all initial states are in W;
- if (†) holds then compute a winning strategy for the system by following the construction provided in If Part of Theorem 8.1.2;
- otherwise, notify that G is winning for the environment.

8.1.2 From Composition Problem to Safety Games

We introduce our approach for reducing a composition problem instance to a $\Box$-GS from a high-level perspective. Recall that the service composition problem consists in finding an orchestrator able to coordinate a community so that the obtained system can always satisfy the requests of a client compliant with a given deterministic target service.

In order to encode the composition problem as a safety-game structure, we need first to individuate which place each component, e.g., target, available services, data box, will occupy in the game representation. To this end, some remarks are worth making:

- given the target service, all and only legal client evolutions result in all possible target service executions;
- the target service is a virtual entity whose operations are to be actually executed by one available service, subject to its current state and capabilities. Therefore, when composing services, both the (virtual) target service and the (actual) community can be soundly thought as evolving synchronously, the latter executing exactly what the former is supposed to;
- our framework prescribes community services to be mutually asynchronous, i.e., exactly one moves at each step, yet synchronous with the data box;
- among all the entities involved in the setting, the only one that needs to be synthesized is the orchestrator: it is an automaton which, synchronously with both the target service and the community, outputs, depending on the current community state and the target service evolution that is being composed, an identifier used to delegate the requested operation to one available service.

Conceptually, our goal is to refine an unconstrained orchestrator –that is, an automaton capable of selecting, at each step, one among all the available
services— in a way such that the community is always able to satisfy target service requests\(^1\). This suggests to identify, in the game structure:

- the orchestrator with the *system*, which is the game player we want to synthesize a winning strategy for and, consequently,
- community services, target service and data box, properly combined, with the *environment*, of course guaranteeing that all synchronization requirements are met.

Finally, once the game players are described, a winning condition needs to be encoded. Recall that system’s goal amounts to guaranteeing that the game goes on while keeping the winning condition verified. In our composition framework, we require that an orchestrator delegates operations, requested by the target service, so that

- if the target service is in a final state, all community services do, as well;
- the service selected by the orchestrator is able to perform the operation currently requested by the target service.

The winning condition will be formally encoded as conjunction of the above high-level properties, in addition to a third one needed to deal with a single (artificial) initial state, rather than many.

Now, we can show how to derive, given a composition problem instance, a □-GS.

**Formal Reduction to Safety-Game Structure**

Before providing technical details, let us address an important issue about TS modeling. As we saw, a game is winning for a system if all plays starting from an initial state are winning. To be such, in particular, a play needs to be infinite. In the reduction we propose next, the composition problem is encoded as a safety game, so that each winning strategy for the system corresponds to a composition and viceversa. To do so, we require that the target service has only infinite runs, otherwise finite plays are allowed and thus, as it will be clear soon, there might be loss of solutions. So, given a service composition problem instance, we preliminarily apply the following transformations, aimed at *extending* all finite runs that are possibly finite:

- if all target service states have at least one successor state (according to transition function), then nothing changes;

\(^1\)Here and in the following, we blur the notions of client and target service as, like we said before, all client evolutions yield all target executions, so one can see a target execution as an alternative representation of some client’s evolution.
• otherwise:
  – a special operation, say \(\text{nop}\), not already present in \(\mathcal{O}\), is added to \(\mathcal{O}\);
  – for each state \(s\) with no successor, a (looping) transition \(s \xrightarrow{\top,\text{nop}} s\) is added to the (functional) transition relation \(\mathcal{q}_t\);
  – for all available services and the data box, a loop as above is added for each of their states.

Intuitively, we require that, after a finite run has been traversed, the target service can only decide to remain still, that each available service can be delegated, at any time, to execute such operation, and that no available service or the data box is affected by \(\text{nop}\). In other words, we are explicitly representing the fact that the whole system remains still over time. From now on, we assume to deal only with infinite-run target services, possibly obtained by applying the above manipulation. Therefore, each community and target service state admit always a successor state (possibly itself).

Now, let \(\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{DB}\}\) be a community and \(\mathcal{S}_t\) a target service over \(\mathcal{DB}\), where \(\mathcal{S}_i = (\mathcal{O}, \mathcal{S}_i, s_{i0}, \mathcal{q}_i, \mathcal{G}_i, \mathcal{S}^f_i)\) \((i = t, 1, \ldots, n)\) and \(\mathcal{DB} = (\mathcal{O}, \mathcal{D}, d_0, \rho)\). From \(\mathcal{C}\) and \(\mathcal{S}_t\), we derive a \(\Box\)-GS \(G = (\mathcal{V}, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_c, \Box \varphi)\), as follows:

\[\begin{align*}
\mathcal{V} &= \{s_t, s_1, \ldots, s_n, d, o, \text{ind}\}, \\
\mathcal{X} &= \{s_t, s_1, \ldots, s_n, d, o\}, \\
\mathcal{Y} &= \{\text{ind}\}, \\
\Theta &= (\bigwedge_{i=t,0,\ldots,n}(s_i = \text{init}) \land (d = \text{init}) \land (o = \text{init}) \land (\text{ind} = \text{init}),
\end{align*}\]

with an intuitive semantics: each complete valuation of \(\mathcal{V}\) represents \((i)\) the current state of community (variables \(s_1, \ldots, s_n\)), data box (variable \(d\)) and target service (variable \(s_t\)), \((ii)\) the operation to be performed next (variable \(o\)) and \((iii)\) the available service selected to perform it (variable \(\text{ind}\)). Special value \(\text{init}\) has been introduced for convenience, so to have a single initial state:

\[\begin{align*}
\mathcal{V} &= \{s_t, s_1, \ldots, s_n, d, o, \text{ind}\}, \\
\mathcal{X} &= \{s_t, s_1, \ldots, s_n, d, o\} \text{ is the set of environment variables;} \\
\mathcal{Y} &= \{\text{ind}\} \text{ is the (singleton) set of system variables;} \\
\Theta &= (\bigwedge_{i=t,0,\ldots,n}(s_i = \text{init}) \land (d = \text{init}) \land (o = \text{init}) \land (\text{ind} = \text{init}),
\end{align*}\]
• Let $S_G = \hat{S}_1 \times \hat{S}_1 \times \ldots \times \hat{S}_n$ be the set of $G$’s environment states. $\rho_e \subseteq S_G \times \{1, \ldots, n, \text{init}\} \times S_G$ is defined as follows:
  
  - $\langle (\text{init, \ldots, init, init, } s_1, s_1, \ldots, s_n, d, o) \rangle \in \rho_e$ if and only if:
    1. $s_i = s_{i0}$, for $i = t, 1, \ldots, n$, and $d = d_0$;
    2. there exists a transition $s_{i0} \xrightarrow{g, o} s'_i$ in $S_t$, for some $g \in G_t$ and $s_i' \in S_t$, such that $g(d_0) = \top$;
    3. there exists a transition $d_0 \xrightarrow{o} d'$ in $DB$, for some $d' \in D$;
  
  - if $s_i \neq \text{init}$, for $i = t, 1, \ldots, n$, $d \neq \text{init}$, $o \neq \text{init}$ and $\text{ind} \neq \text{init}$ then $\langle (s_t, s_1, \ldots, s_n, d, o), \text{ind}, \langle s'_1, s'_1, \ldots, s'_n, d', o' \rangle \rangle \in \rho_e$ if and only if the following holds:
    1. there exists a transition $s_t \xrightarrow{g, o} s'_i$ in $S_t$, for some $g \in G_t$, such that $g(d) = \top$;
    2. there exists a transition $d \xrightarrow{o} d'$ in $DB$;
    3. there exists a transition $s_{\text{ind}} \xrightarrow{g, o} s'_{\text{ind}}$ in $S_{\text{ind}}$, for some $g \in G_{\text{ind}}$, such that $g(d) = \top$;
    4. $s_i' = s_i$, for all $i = 1, \ldots, n$ such that $i \neq \text{ind}$;
    5. there exists a transition $s_i' \xrightarrow{g', o'} s''_i$ in $S_t$, for some $g' \in G_t$ and $s''_i \in S_t$, such that $g'(d') = \top$;
    6. there exists a transition $d' \xrightarrow{o'} d''$ in $DB$;

• $\rho_s \subseteq S_G \times \{1, \ldots, n, \text{init}\} \times S_G \times \{1, \ldots, n, \text{init}\}$ is defined as follows:
  
  - $\rho_s(\langle s_t, s_1, \ldots, s_n, d, o \rangle, \text{ind}, \langle s'_1, s'_1, \ldots, s'_n, d', o' \rangle, \text{ind}') \in \rho_s$ if and only if
    - $\rho_e(\langle s_t, s_1, \ldots, s_n, d, o \rangle, \text{ind}, \langle s'_1, s'_1, \ldots, s'_n, d' o' \rangle)$, and
    - $\text{ind}' \in \{1, \ldots, n\}$;

• Formula $\varphi$ is defined depending on current service states $s_t, s_1, \ldots, s_n$, operation $o$ and service selection $\text{ind}$, as follows (for succinctness, variables that each term depends on are omitted, e.g., we write $\text{fail}_i$ instead of $\text{fail}_i(s_t, d, o, \text{ind})$):

  $$\varphi \doteq \Theta \lor \left( \bigwedge_{i=1}^{n} \neg \text{fail}_i \right) \land (\text{final}_t \rightarrow \bigwedge_{i=1}^{n} \text{final}_i),$$

  where:

  - $\Theta$ is defined as above;
  
  - $\text{fail}_i \doteq (\text{ind} = i) \land (\bigwedge_{(s, g, o, s')} \in E_{\Theta}(g(d) = \bot \lor s_i \neq s \lor o \neq o))$ encodes the fact that service $i$ has been selected but, in its current state, no transition can take place which executes the requested operation;
8.1. SIMULATION AND SAFETY GAMES

- final \( i \) encodes the fact that service \( i = t, 1, \ldots, n \) is currently in one of its final states.

Observe how, since game structures do not allow for transition labeling, the operation requested by a client appears as a state variable, in the game representation. This transformation is similar, in the spirit, to the procedure usually adopted to transform a Mealy into a Moore machine. As for the winning condition, it is an invariant property that needs to be always satisfied, in order for the system to win or, rephrased from service composition viewpoint, in order for an orchestrator to be a composition. It should be intuitively clear that each game play essentially reproduces a synchronous execution of the target service and the community, when sequences of operations compliant with target service specification are executed. Analogously, the converse holds. Such intuitions will be formally addressed later on in this chapter.

The basic idea behind the reduction to a safety-game structure is that, once the service composition problem is encoded as a 2-GS, from the maximal winning set, computed as in Algorithm 4, one can extract a NCG (see 4.3.3), thus a composition which solves the original problem.

Technical Results

We can now show how the obtained game structure allows for computing a composition generator. Recall that, in order to define the NCG, one needs to build an ND-simulation (see Definition 4.3.3). The following Theorem shows that this can be equivalently done by computing the maximal set of winning states for the system.

**Theorem 8.1.4** Let \( C = \{S_1, \ldots, S_n, DB\} \) be a community and \( S_t \) a target service over \( DB \) where, as usual, \( S_i = \langle O, S_i, s_{i0}, q_i, G_i, S_f^i \rangle \) \( (i = 1, \ldots, n, t) \) and \( DB = \langle O, D, d_0, \rho \rangle \). From \( C \) and \( S_t \) derive a \( \Box \)-GS \( G = \langle V, X, Y, \Theta, \rho_c, \rho_s, \Box \varphi \rangle \), as shown above, and let \( W \subseteq V \) be the maximal set of winning states for the system. Then, \( \langle \text{init}, \ldots, \text{init} \rangle \in W \) if and only if \( \subseteq (s_{i0}, s_{C0}, d_0) \).

**Proof:** (If Part.) Consider an ND-simulation relation \( R \) such that \( (s_{i0}, s_{C0}, d_0) \in R \), where \( s_{C0} = \langle s_{10}, \ldots, s_{n0} \rangle \). From this, we build a new relation \( \bar{R} \subseteq X \times Y \), as follows: for each \( r = (s_i, s_C, d) \in R \), where \( s_C = \langle s_1, \ldots, s_n \rangle \), for each \( S_t \) transition \( s_t \xrightarrow{o} s'_t \) as in requirement 2 of Definition 4.3.1 such that there exists \( d \xrightarrow{o} d' \) in \( DB \), and for each \( k \) as in requirement 2 of Definition 4.3.1, let \( \bar{r} = ((s_t, s_1, \ldots, s_n, d, o), k) \in \bar{R} \). For each of such \( \bar{r} \), we have that \( \bar{r} \vdash \varphi \) and \( \bar{r} \in \pi(\bar{R}) \). For the former, consider formulae \( \text{fail}_i \) and \( \text{final}_i \), defined above and observe that, for \( \bar{r} = ((s_t, s_1, \ldots, s_n, d, o), k) \), by construction of \( \bar{R} \) and by definition of ND-simulation:
• there exists a transition $s_k \xrightarrow{g_o} s_k'$ in $S_k$, for some $s_k' \in S_k$ and $g \in G_k$, such that $g_k(d) = \top$, hence, $\text{fail}_i$ is never satisfied, for $i = 1, \ldots, n$;

• if $s_i \in S_i^f$ then $s_i \in S_i^f$ for all $i = 1, \ldots, n$, hence, $\text{final}_i \rightarrow \wedge_{i=1}^n \text{final}_i$ is always satisfied.

The fact that $\hat{r} \in \pi(\hat{R})$ comes from: (i) requirement 2 of Definition 4.3.1, and (ii) $\hat{R}$ construction which is, essentially, a certified representation of $R$, where all operations executable by $S_t$ and respective service selections $\text{ind}$ are explicit. Due to these, for each environment successor $\langle s'_t, s_1, \ldots, s_n, d', o' \rangle$, $\hat{r}$ has a successor $\hat{r}' = \langle \langle s'_t, s_1, \ldots, s_n, d', o' \rangle, \text{ind}' \rangle$ in $G$, i.e., such that $\rho_e(\langle s_t, s_1, \ldots, s_n, d, o \rangle, \text{ind}, \langle s'_t, s_1, \ldots, s_n, d', o' \rangle)$ and $\rho_s(\langle s_t, s_1, \ldots, s_n, d, o \rangle, \text{ind}, \langle s'_t, s_1, \ldots, s_n, d', o' \rangle, \text{ind}')$.

Clearly, $\hat{r}' \in R$, as $\langle s'_t, s_1, \ldots, s_n, d' \rangle \in R$ and, by $G$ construction, (i) $s'_t$ has at least one successor under $o'$ and (ii) $\text{ind}'$ is properly chosen so that $S_{ind'}$ is able to execute $o'$ in its current state with respect to $\hat{r}'$. Hence, for each $\hat{r} \in \hat{R}$, $\hat{r} \models \varphi$ and $\hat{r} \in \pi(\hat{R})$. Therefore, by Theorem 8.1.1, due to maximality of $W$, $\hat{R} \subseteq W$. But, then, also $\langle \text{init}, \ldots, \text{init} \rangle \in W$. Indeed, (i) $\langle \text{init}, \ldots, \text{init} \rangle \models \varphi$ as, by definition, $\langle \text{init}, \ldots, \text{init} \rangle \models \Theta$ and (ii) $\langle \text{init}, \ldots, \text{init} \rangle \in \pi(\hat{R})$ as, by $G$ and $\hat{R}$ constructions, all $\langle \text{init}, \ldots, \text{init} \rangle$ successors are in $\hat{R}$ (in particular, they are of the form $\hat{r}_0 = \langle \langle s_{t_0}, s_{t_0}, \ldots, s_{t_0}, d_0, o \rangle, \text{ind} \rangle$).

(Only If Part.) Let $R \subseteq S_t \times (S_1 \times \cdots \times S_n) \times D$ be a relation inductively defined as follows:

• for each game successor of $w_{\text{init}} = \langle \langle \text{init}, \ldots, \text{init} \rangle, \text{init} \rangle \in W$, say $w_0 = \langle \langle s_{t_0}, s_{t_0}, \ldots, s_{t_0}, d_0, o \rangle, \text{ind} \rangle$, let $r_0 = \langle s_{t_0}, s_{t_0}, \ldots, s_{t_0}, d_0 \rangle \in R$ and $w_0 \in W^0$.

• for each element $w = \langle \langle s_t, s_1, \ldots, s_n, d, o \rangle, \text{ind} \rangle \in W^{t-1}$ and for each of its game successors $w' = \langle \langle s'_t, s'_1, \ldots, s'_n, d', o' \rangle, \text{ind}' \rangle$ such that $w' \in W$, let $\langle s'_t, s'_1, \ldots, s'_n \rangle, d' \rangle \in R$ and $w' \in W^1$.

We show that $R$ is an ND-simulation relation. First, it is straightforward seeing that $w \models \varphi$, for all $w \in W$, yields that for all $\langle s_t, (s_1, \ldots, s_n), d \rangle \in R$, if $s_t \in S_t^f$, then $\langle s_1, \ldots, s_n \rangle \in S_C^f$. Now, focus on requirement 2 of Definition 4.3.1. We proceed by induction.

• (Base case) For each game successor of $w_{\text{init}}$, consider $w_{\text{init}}^0$ such that $\rho_e(\langle \langle \text{init}, \ldots, \text{init} \rangle, \text{init}, w_{\text{init}}^0 \rangle)$, where $w_{\text{init}}^0 = \langle s_{t_0}, s_{t_0}, \ldots, s_{t_0}, d_0, o \rangle$. For each of such $w_{\text{init}}^0$, there exists an $\text{ind} \in \{1, \ldots, n\}$ such that $\langle w_{\text{init}}^0, \text{ind} \rangle \in W$, as $w_{\text{init}} \in W \subseteq \pi(W)$. Considering $R$ and $\rho_e$ definitions, this yields that
for each \( r_0 = (s_{t_0}, \langle s_{t_0}, \ldots, s_{n_0} \rangle, d_0) \in R \) and (\( \dagger \)) for each \( o \in \mathcal{O} \) such that, for some \( s'_t \in S_t \), \( d' \in D \) and \( g \in G_t \) such that \( g(d) = \top \), there exist \( s_{t_0} \xrightarrow{g,o} s'_t \) in \( S_t \) and \( d_0 \xrightarrow{o} d' \) in \( DB \) (in the reminder, we refer to (\( \dagger \)) by writing that \( o \in \mathcal{O} \) is executable in \( s_{t_0} \), or, in general, in \( s_t \)), there exists an \( ind \in \{1, \ldots, n\} \) such that \( w_0 = (w_e, ind) \in W \).

- (Induction step) Consider a generic \( r = (s_t, \langle s_1, \ldots, s_n \rangle, d) \in R \) and assume that for each \( o \in \mathcal{O} \) executable in \( s_t \), with \( s_t \xrightarrow{g,o} s'_t \) and \( d \xrightarrow{\bullet} d' \), there exists \( ind \in \{1, \ldots, n\} \) such that \( w = (w_e, ind) \in W \), where \( w_e = (s_t, s_1, \ldots, s_n, d, o) \). Since \( w \in W \subseteq \pi(W) \) and by \( \rho_e \) definition, for each \( o' \in \mathcal{O} \) executable in \( s'_t \), there exists \( ind' \in \{1, \ldots, n\} \) such that \( w' = (s'_t, s'_1, \ldots, s'_n, d', o') \in W \). So, by \( R \) definition above, \( r' = (s'_t, s'_1, \ldots, s'_n, d') \in R \).

Based upon this result, the following theorem provides an actual procedure to build an ND composition generator, as in Definition 4.3.3, and, from this, all possible compositions, as shown by Theorem 4.3.5.

**Theorem 8.1.5** Let \( C = \{S_1, \ldots, S_n, DB\} \), \( S_t, DB \) and \( G = \langle V, \mathcal{X}, \mathcal{Y}, \mathcal{G}, \mathcal{O}, \rho_e, \rho_s, \square \mathcal{F} \rangle \) be as in Theorem 8.1.4 above. In addition, let \( W \) be the winning set for the system. If \( (\text{init}, \ldots, \text{init}) \in W \), then the ND composition generator \( NCG = \langle \mathcal{O}, \{1, \ldots, n\}, S_g, s_{y_0}, \omega_g, \delta_g \rangle \) of \( C \) for \( \mathcal{S}_t \) can be built from \( W \), as follows:

- \( \mathcal{O} \) is the usual set of operations and \( \{1, \ldots, n\} \) the set of available services’ indexes;
- \( S_g \subseteq S_t \times S_{\mathcal{O}} \times D \) is such that \( \langle s_t, \langle s_1, \ldots, s_n \rangle, d \rangle \in S_g \) if and only if there exists a game state \( \langle s_t, s_1, \ldots, s_n, d, o, \text{ind} \rangle \in W \), for some \( o \in \mathcal{O} \) and \( \text{ind} \in \{1, \ldots, n\} \);
- \( s_{y_0} = (s_0, (s_{t_0}, \ldots, s_{n_0}), d_0) \) is \( NCG \)’s initial state;
- \( \omega_g : S_g \times \mathcal{O} \rightarrow 2^{\{1, \ldots, n\}} \) is defined as \( \omega_g(\langle s_t, \langle s_1, \ldots, s_n \rangle, d \rangle, o) = \{i \in \{1, \ldots, n\} \mid \langle s_t, s_1, \ldots, s_n, d, o, i \rangle \in W \} \);
- \( \delta_g : S_g \times \mathcal{O} \times \{1, \ldots, n\} \rightarrow S_g \) such that \( \langle \langle s_t, \langle s_1, \ldots, s_n \rangle, d \rangle, o, k \rangle, \langle s'_t, \langle s'_1, \ldots, s'_n \rangle, d' \rangle \rangle \in \delta_g \) if and only if \( \langle s_t, s_1, \ldots, s_n, d, o, k \rangle \in W \) and there exist \( o' \in \mathcal{O} \) and \( k' \in \{1, \ldots, n\} \) such that \( \langle s'_t, s'_1, \ldots, s'_n, d', o', k' \rangle \in \rho_s \).

**Proof:** Comes from (If Part) of Theorem 8.1.4, the observation that \( W \) is, in fact, an ND-simulation relation **enriched** with all executable operations and all respective indices for selecting community TS transitions, and from Definition 4.3.3. \( \square \)
Above theorems show how one can exploit tools from system synthesis for computing all compositions of a given target service. In details, starting from $\mathcal{C} = \{S_1, \ldots, S_n, DB\}$ and $S_t$, one can build the corresponding game structure $G$ as shown above, then compute the set $W$ through the use of some available tool and, if it contains $G$’s initial state, use such set to generate the NCG. 

In fact, this last step is not really needed. Indeed, it is not hard to convince oneself that given a current state $\langle s_t, s_1, \ldots, s_n, d \rangle$ and an operation to be executed $o \in \mathcal{O}$ a service selection ind is ”good” (i.e, the selected service can actually execute the operation and the whole community can still simulate the target service) if and only if $W$ contains a tuple $\langle s_t, s_1, \ldots, s_n, d, o, ind \rangle$, for some $\text{ind} \in \{1, \ldots, n\}$. Consequently, at each step, on the basis of the current state $s_t$ of the target service, the states $s_1, \ldots, s_n$ of available services, the state $d$ of data box, and the operation $o$ requested, one can select a tuple from $W$, extract the $\text{ind}$ component, and use this for next service selection.

Finally, observe that time complexity of Algorithm 4 is polynomial in $|V|$, that is the size of input $\square$-GS’ state space. Since in our encoding $|V|$ is polynomial in $|S_1|, \ldots, |S_n|, |S_t|, |D|$ and exponential in $n$, we get the following result.

**Theorem 8.1.6** Let $\mathcal{C} = \{S_1, \ldots, S_n, DB\}$ be a community and $S_t$ a target service over $DB$. Checking the existence of compositions by reduction to safety games can be done in polynomial time with respect to $|S_1|, \ldots, |S_n|, |S_t|, |D|$ and exponential time with respect to $n$.

That is, the technique is actually optimal wrt worst-case time complexity, the composition problem being EXPTIME-hard [108].

### 8.2 Using TLV for computing compositions

So far, we have seen that the synthesis of a composition can be reduced to searching for a set of winning states for a system, and to the synthesis of a respective strategy. Currently, there are many implemented systems, such as Anzu ([85]), Lily ([84]) or TLV ([124]), just to mention some, able to synthesize a strategy for various kinds of game. Here, we focus in particular on TLV, which serves as a sample tool, so to make actual the theoretical notions introduced. All basic concepts remain essentially valid for all others.

TLV is a software for verification and synthesis of general LTL specifications, which is based on symbolic manipulation of states made possible by the use of Binary Decision Diagrams (BDDs). It takes two inputs: (i) a procedure, encoded in TLV-BASIC, and (ii) an LTL specification, encoded in SMV [133]. The LTL specification is provided as input to the procedure which is executed and
8.2. USING TLV FOR COMPUTING COMPOSITIONS

whose output is, then, returned. In particular, we exploit a given \(^2\) TLV-BASIC procedure for safety games, which takes as input an LTL specification supposed to encode a \(\Box\)-game and returns, if any, the system’s maximal winning set, enriched with all the transitions among the contained game states.

The purpose of this section is to show a methodology for producing, starting from an abstract description of a composition problem, the SMV code that the procedure can deal with, so to automatically generate, if any, the system’s winning set. Our approach consists in deriving, starting from the problem specification, i.e., the community and the target service, the SMV encoding of the \(\Box\)-GS it generates, as shown in subsection 8.1.2. We will proceed bottom up, by first building those basic blocks, i.e. SMV modules, that are representative of available and target services and data box, and then interconnecting them so to create the environment, the system and, finally, the whole game structure. Here, no detail about the language SMV are given. For a deeper introduction, we refer the reader to [133]. However, its syntax and semantics are pretty intuitive and the reader should find no major problems in understanding it, as the encoding we propose is essentially a rewriting of the abstract structures introduced so far.

8.2.1 The SMV encoding

Here, we detail the encoding of a service composition problem. Let us start by introducing an example, which will be used in the following to provide an actual instance of the encoding.

Example 8.2.1 Figures 8.1, 8.2 and 8.3 depict an instance of a service composition problem.

Figure 8.1 shows the behavior of a media-library system which provides, among others, two operations: pick and store. The former randomly chooses a media content, which can be either audio or video, and put it into a buffer accessible to clients, while the latter allows a client to access the buffer, copying the content locally, and emptying it. States \(d_0\), \(d_1\) and \(d_2\) model, respectively: emptiness of the buffer, presence of an audio content and presence of a video content. Since pick is a nondeterministic operation, its execution may lead the data box to either state \(d_1\) or \(d_2\), depending on whether the selected item is audio \((d_1)\) or video \((d_2)\). Observe that operation store can be performed only after pick, as it guarantees that the buffer is not empty. The pick operation, on the other hand, is always allowed: when the buffer is not empty, it is overwritten with the next content picked up. The picture is simplified by implicitly...
CHAPTER 8. COMPOSITION AND SAFETY GAMES

Figure 8.1: Graphical representation of a “media-library” data box.

assuming that all non-mentioned operations are always executable and yield no effect on the data box -graphically, they would be self-loops-, though this is not the case in general (the assumption is valid only for Figure 8.1).

The services represented in Figure 8.2 constitute the community. As usual, concentric circles represent final states and an edge from s to s' is labeled with “g : o” if and only if s \(\xrightarrow{g,o} s'\) is in S. When g is identically true, prefix “g :” is omitted. For example, consider service \(S_1\) in Figure 8.2(a). It is able to perform operations: pick (ask data box for picking up a media item), store (download item from data box’ buffer and empty it), and free_mem (compress local items to free memory space), subject to current state and, possibly, guard evaluations. In detail, when \(S_1\) is in state \(s_{10}\), operation store can be performed, provided \(DB\) is in a state where an audio or video item has been selected. As already observed, data box’ state \(d_1\) (respectively \(d_2\)) represents the selection of an audio (video) item, that is, evaluation of guard video (audio) on state \(d_1\) (\(d_2\)) is video\((d_1) = \top\) (audio\((d_2) = \top\)). Operation store, besides being guarded, is nondeterministic. In fact, if performed when video is evaluated to \(\top\), two outcomes are possible (it is not the case if audio is true): the service can move either to \(s_{11}\) or \(s_{12}\). The former state models successful completion of the operation, whereas the latter represents the fact that \(S_1\) has gone out of memory (video items are assumed to be larger than audio). In such case, before performing any other operation, \(S_1\) must free some disk space by performing a free_mem. Observe that free_mem can be performed also in \(s_{12}\), without yielding any state change. Finally, from state \(s_{11}\), \(S_1\) may instruct the data box to pick up a new item, by performing the unconditioned operation pick. Semantics of other services in Figure 8.2 is straightforward.

Finally, a sample target service defined over \(DB\) is shown in Figure 8.3.

In the following, we will see how such example can be encoded in the language smv to represent a safety game structure. First, we need to introduce SMV modules.
8.2. USING TLV FOR COMPUTING COMPOSITIONS

Figure 8.2: Available services interacting with the media-library data box.

Figure 8.3: Target service for the media-library example.
**Modules** An SMV module is a syntactic structure starting with the keyword **MODULE**, possibly followed by some parameters, and ending where a following module starts or the SMV code ends. Essentially, it represents a transition system, possibly obtained as composition of other modules, with optional input-output parameters. Among the many (possibly optional) sections it may contain, we will consider only those four which are needed for our purposes:

1. the variable section, enclosed within the keyword **VAR** and the following section/module, where state variables are declared, along with their respective ranges;

2. the initialization section, enclosed within the keyword **INIT** and the following section/module, intended to hold a boolean expression characterizing the initial states (one only, in our case);

3. the transition section, enclosed within the keyword **TRANS** and the following section/module, providing a definition of the transition relation;

4. the expression-definition section, enclosed within the keyword **DEFINE** and the following section/module, intended to contain definitions of boolean expressions over module parameters, internal variables or other defined expressions.

A sample module, representing the SMV encoding of the target service in Figure 8.3, is provided in Figure 8.4. The keyword **MODULE** is followed by the name of the module, i.e., Target, possibly followed, in turn, by a list of parameters between parentheses. Let call such first line the *signature* of the module. In this case, the module has two parameters: op and db_state. As the comment (from -- to the end of line) says, the former is an output parameter, while the latter is an input one. Actually, SMV provides no explicit construct that defines a module parameter as input or output: parameter reading or writing is always allowed. However, we say that a parameter is input if the module never modifies it, and output otherwise. Module parameters are used within the body of the module and, in fact, provide a mechanism for inter-module communication: (i) transitions between module states can be subject to the value of some input parameter, usually set by another module and, on the other hand, (ii) an output parameter can be set by a module to constrain another module’s behavior.

The **VAR** section tells us that module Target contains a state variable called state, which ranges over the value set \{start_st,t0,t1,t2,t3\}. Observe that, differently from state variables, parameters have no domain defined. Actually, such definition appears elsewhere, so we can still assume that each parameter has a (finite) domain.
A module containing only the signature and a \texttt{VAR} section is perfectly conceivable. In particular, it represents a (extended) transition system that can start in any initial state and where a transition is allowed between any pair of states, where module states are joint evaluation of state variables and parameters, over their respective domains (although parameter domains are defined externally, they are always taken into account). However, such a module would be not very useful. In order to specify more constrained modules, we need an \texttt{INIT} and a \texttt{TRANS} sections.

The \texttt{INIT} section of \texttt{Target} constrains the initial state to assign value \texttt{start_st} to variable \texttt{state} and value \texttt{start_op} to parameter \texttt{op} (that will be, presumably, read by other modules). As for parameter \texttt{db_state}, no initialization is explicitly defined within the module, so it can range over all of its values (unless initialized elsewhere).

The \texttt{TRANS} section specifies the transition relation of the module. In this particular case, it is expected to be functional since, as we said, target services are deterministic. The \texttt{case} statement specifies, based on the current value of state variables and input parameters (i.e., the \texttt{condition} before “:" in each case entry), the next values of state variables and output parameters. A target service, in order to select the operation to execute next (parameter \texttt{op}), needs to evaluate the guard of the respective transition. That is, before operation selection, the target service needs to access first the data box state. Unfortunately, \texttt{SMV} provides no mechanism to explicitly doing so, therefore an alternative solution is adopted. Focus on third entry of \texttt{case} statement. Such entry says that if \texttt{state=t1} and \texttt{op=play} then, at next step: (i) parameter \texttt{op} can take any value from domain \{\texttt{pick, store}\}; (ii) variable \texttt{state} assumes value \texttt{t2}; (iii) value \texttt{pick} can be assigned to \texttt{op} only if, at next step, \texttt{db_state} assumes value \texttt{d2}; (iv) \texttt{store} can be assigned to \texttt{op} only if, at next step, \texttt{db_state} assumes value \texttt{d1}. This way, operation selection at next step is constrained by next data box state—which is not a-priori known, as it is, in general, nondeterministic—, thus requiring first to evaluate the latter and then to choose the former. Observe that, in its representation as an \texttt{SMV} module, the target service introduces nondeterminism over operations. It is the case, for instance, of state \texttt{t1}, when operation \texttt{play} is executed. Indeed, after operation execution, the module can be in a state \texttt{t2} and parameter \texttt{op} can assume either value \texttt{pick} or \texttt{store}. Such nondeterminism reflects the uncertainty about the next operation that the client requests.

Finally, in the \texttt{DEFINE} section two expressions are defined, namely \texttt{initial} and \texttt{final}, which identify, respectively, initial and final states.

After this high-level, intuitive description, let us provide a formal approach to module encoding, that allows for representing a service composition problem instance as a $\square$-GS. It is worth noticing that the encoding schema we provide
MODULE Target(op, db_state) -- op is an output parameter, db_state is input
VAR
  state : {start_st, t0, t1, t2, t3};
INIT
  state = start_st & op = start_op
TRANS
  case
    state = start_st & op = start_op : next(state) = t0 & next(op) in {pick};
    state = t0 & op = pick : next(state) = t1 & next(op) = play;
    state = t1 & op = play : next(op) in {pick, store} & next(state) = t2 &
     (next(op) = pick -> next(db_state) = d2) &
     (next(op) = store -> next(db_state) = d1);
    state = t2 & op = pick : next(state) = t1 & next(op) in {play};
    state = t2 & op = store: next(state) = t3 & next (op) in {free_mem};
    state = t3 & op = free_mem: next(state) = t0 & next(op) in {pick};
  esac
DEFINE
  initial := state=start_st & op=start_op;
  final := state in {t1}; -- final state(s)

Figure 8.4: SMV encoding of target service of Example 8.2.1

is not optimal, neither with respect to the encoding size nor to the solution
time. In particular, for the former, more compact descriptions can be provided,
while for the latter, one needs to get into details of the TLV solution engine,
i.e., the algorithm for winning set computation. Even though this topic is of
great interest, in this work we only focused on the problem of finding a feasible
encoding. Observe that once an encoding is

**Data box.** Given a data box $DB = \langle O, D, d_0, \rho \rangle$, the SMV encoding is as
follows:

1. we create a module named `Databox`, containing an input parameter
   `operation`, intended to store the operation (externally chosen by a
   client) to be performed next;

2. in the `VAR` section, we add a state variable `state` ranging over the set
   $\{d_0, \ldots, d_{|D|-1}\} \cup \{\text{start_st}\}$, where we assume that $D = \{d_0, \ldots, d_{|D|-1}\}$
   contains exactly the same number of symbols as $D$ and there exists a
   mapping from $D$ to $D$ such that names are preserved, e.g., $xyz \in D$
   maps to $xyz \in D$, so that notations can be soundly interchanged. Moreover,
   the special state `start_st` is used to represent the `init` state used in the
   $\square$-game representation\(^3\).

\(^3\)String `init` is not allowed as it is an SMV keyword.
8.2. USING TLV FOR COMPUTING COMPOSITIONS

MODULE Databox(operation)
VAR
state:{start_st,d0,d1,d2};
INIT
state=start_st
TRANS
case
  state = start_st & operation = start_op : next(state) = d0;
  state = d0 & operation = pick : next(state) in {d1,d2};
  state = d1 & operation = pick : next(state) in {d1,d2};
  state = d2 & operation = pick : next(state) in {d1,d2};
  state = d1 & operation = store : next(state) in {d0};
  state = d2 & operation = store : next(state) in {d0};
  TRUE : next(state) = state;
esac

Figure 8.5: Encoding of data box of Example 8.2.1

3. in the INIT section, we add the expression state = start_st;

4. the TRANS section consists of a case statement containing one condition for each transition in \( \rho \), plus an initialization condition stating that, after first step, the module moves to a state representing the actual initial state (as appearing in the original abstract description) of DB, provided special operation start_op is executed. More in details, let \( \rho \) be nondeterministic, that is, for a same pair of state \( d_i \) and operation \( o_i \) (\( i = 1, \ldots, p \)) there exist many transitions \( \langle d_i \xrightarrow{o_i^j} d_i^j \rangle \in \rho \), where \( j = 1, \ldots, q(i) \). Then, the case statement assumes the following general form, where \( d_0 \) is the state that the DB’s (only) initial state maps to:

\[
\begin{align*}
\text{case} \\
state = start_st & \text{ operation } = \text{ start_op : next(state) } = s_0; \\
state = d_i & \text{ operation } = o_i^j : next(state) \in \{d_i^{q(i)}, \ldots, d_i^{q(i)}\}; \\
state = d_i & \text{ operation } = o_i : next(state) \in \{d_i^{q(i)}, \ldots, d_i^{q(i)}\}; \\
\ldots \\
state = d_p & \text{ operation } = o_p : next(state) \in \{d_p^{q(p)}, \ldots, d_p^{q(p)}\}; \\
esac
\]

Note that, according to the interpretation of input parameter, no value is assigned to operation, by this module.

For instance, in Figure 8.5, the SMV encoding of the data box introduced in Example 8.2.1 is shown. As we said, more compact representations may exist. Last condition of case statement says that if none of the above conditions hold, then the requested operations is executable and yields a self-loop on the
current state. This reflects the same picture simplification we adopted in the example.

**Available services.** The available service encoding schema is an extension of data box’ one. Essentially, the extension is required to deal with the more complex structure one obtains when guards and service selections are introduced. We refer to guards as boolean disjunctions intended to test whether current data box state belongs to a given set. They have the following form: 

\[(\text{db state} = s^1 | \ldots | \text{db state} = s^r),\]

where operator | stands for logical disjunction, in SMV. Concerning service selection, we require available services to loop in their current state when not selected. This corresponds to an asynchronous execution schema, where only the selected service moves, while others remain still.

In details, let \( S_i = (\mathcal{O}, S_i, s_{i0}, q_i, G_i, S_i^f) \) be the generic available service \((i = 1 \ldots, n)\) over data box \( DB = (\mathcal{O}, D, d_0, \rho) \). The SMV encoding is obtained as follows:

1. we build a new module Service\(_i\) with input parameters index, operation and db\_state, the first representing current service selection, the second current requested operation and the third current data box state;

2. VAR and INIT sections are encoded the same way as in data box. In particular, within VAR, a variable state is declared that ranges over the states in \( S_i \) or, more precisely, in \( S_i \) (see data box encoding for naming conventions), whereas section INIT contains exactly the same expression as in data box encoding (see steps 2 and 3);

3. the TRANS section contains a case statement that, in addition to current service state and requested operation, takes into account service selection and guards. Recall that guards play the role of preconditions for transition execution. In order for an operation \( o \) to be correctly performed in a state \( s \), the following requirements must be satisfied:

   - service \( S_i \) is selected, i.e., input parameter index has value \( i \), otherwise the service loops in its current state (i.e, remains still);
   - among all transitions \( s \xrightarrow{g,o} s' \) in \( S_i \), there exists at least one whose guard evaluates to \( \top \) in current \( DB \) state (represented by input parameter db\_state);
   - if there are many transitions as above, whose guards evaluate to \( \top \), then all of them are allowed to take place. In other words, if data box is in state \( d \) and there are many transitions \( s \xrightarrow{g,o} s' \) such
that \( g(d) = \top \) then all \( s' \) must be considered as possible successors of \( s \), when operation \( o \), which is nondeterministic, in general, is executed.

Now, consider relation \( R \subseteq S_1 \times \mathcal{O} \) such that \( (s, o) \in R \) if and only if there exist \( g \in G \) and \( s' \in S_1 \) such that \( g(s, g, o, s') \). Intuitively, \( R \) contains a pair \( (s, o) \) if and only if \( o \) is executable in \( s \), subject to \( DB \) state. From \( R \), we define the following:

- refer to \( R \) element by means of superscripts, \( R = \{(s^1, o^1) \ldots (s^p, o^p)\} \), and let \( (s^l, o') \) be the generic element;
- in general, for each \( R \) element, there exist many, say \( w(l) \geq 1 \), guards \( g \) such that \( g_l(s, g, o, s') \). Refer to them as to \( g^l v \), where \( v = 1, \ldots, w(l) \);
- for each \( l \), define the set \( D^l \subseteq D \) of \( DB \) states satisfying at least one guard \( g^l v \), i.e., \( D^l = \{d \in D \mid \exists g_v d^l = \top, 1 \leq v \leq w(l)\} \). \( D^l \) is the set of \( DB \) states that allow \( S_l \) to execute \( o' \) from state \( s^l \). Refer to its members as to \( d^{lr} \), where \( r = 1, \ldots, s(l) = |D^l| \);
- finally, for each pair \( lr \), define the set \( S^{lr} \) of states reachable from \( s^l \), when \( o' \) is executed and \( DB \) is in state \( d^{lr} \), \( S^{lr} = \{s \in S_1 \mid \exists s^l g^{lr} d' \rightarrow s in S_1, \ for \ some \ v \in \{1, \ldots, w(l)\}\} \), and refer to its elements as to \( s^{lrm} \), where \( m = 1, \ldots, q(l, r) = |S^{lr}| \).

These sets are useful in describing the following case statement (\( \text{index} = 0 \) corresponds to \( ind = \text{init} \) in \( \square - GS \)), which encodes the transition relation of the \( smv \) module corresponding to \( S_l \):

```plaintext
case
  state = start & operation = start & p & index = 0 : next(state) = s_0;
  index != i : next(state) = state;
  state = s^l & operation = o^l : db_state in \{d^{l1}, \ldots, d^{lw(l)}\} &
    (db_state = d^{l1} \rightarrow next(state) in \{s^{l11}, \ldots, s^{lw(l,1)}\}) &
    \ldots &
    (db_state = d^{lw(l)} \rightarrow next(state) in \{s^{l1w(l)}, \ldots, s^{lw(l,1)w(l)}\});
  \ldots
  state = s^p & operation = o^p : db_state in \{d^{p1}, \ldots, d^{pw(p)}\} &
    (db_state = d^{p1} \rightarrow next(state) in \{s^{p11}, \ldots, s^{pw(p,1)}\}) &
    \ldots &
    (db_state = d^{pw(p)} \rightarrow next(state) in \{s^{p1w(p)}, \ldots, s^{pw(p,w(p))}\});
  esac
```

First line of this statement encodes the same idea as in \( DB \), that is: from (artificial) module initial state, the module can only move to a state that represents the initial state in the original (abstract) description. The second line constrains the service to remain still, i.e., looping
in its current state, if not explicitly selected. The rest of the statement implements the same ideas already discussed for $DB$, extended to cope with guards. Intuitively, they state that, fixed the state $s$ (as result of previous transition) and the operation $o$ (externally set by the target service), (i) the data box must be in a state such that a transition can take place and (ii) the module can move to any state reachable by some transition currently allowed (by $DB$ state). When \( \{d^{11}, \ldots, d^{1s(1)}\} = D \), expression $db\_state \in \{d^{11}, \ldots, d^{1s(1)}\}$ can be safely omitted as, by default, variable $db\_state$ assumes one of such values. Similarly, antecedents in implications can be omitted if possible successor states are not dependent on current $DB$ state.

4. A `DEFINE` section is introduced, to define $i$-th service contributions to goal formula statement. It consists of three expressions, each of them being true if and only if module is in an initial, failure and final state, respectively. Referring to the direct correspondence with the $\square$-GS formulation, they will be later combined to form the whole game goal formula. Assume the same notation used above to define module transitions and let $S^f_i = \{s^f_1, \ldots, s^f_r\} \subseteq S_i$. The `DEFINE` section has the following form:

```
initial := (state = start_st & operation = start_op & index = 0);
failure := index = 1 &
\[ ((\text{state} = s^1 & \text{operation} = o^1 & \text{db}\_state \in \{d^{11}, \ldots, d^{1s(1)}\})) | \]
\[ \ldots | \]
\[ (\text{state} = s^p & \text{operation} = o^p & \text{db}\_state \in \{d^{p1}, \ldots, d^{ps(p)}\}) \];
final := (state in \{s^f_1, \ldots, s^f_r\});
```

Observe that the `final` expression essentially encodes those situations where the service has been selected but the requested operation cannot be executed, due either to current service state or current $DB$ state. An actual example of available service encoding can be found in Figure 8.6.

**Target Service.** To encode the target service, one must keep in mind that its traces correspond to those of a potential client which interacts with the community. This means it selects the operation to be executed next, which, as exactly as for available services, may have preconditions. The obtained encoding is very similar to available services, nonetheless, there are some differences, due to next operation selection. Given a target service $S_t = (O, S_t, s_{t0}, \varrho_t, G_t, S^f_t)$ defined over $DB = (O, D, d_0, \rho)$, we get the following encoding:

1. we create a module named `Target` with an input and an output parameters, namely $db\_state$ and $op$, which represent current $DB$ state and selected operation, respectively;
8.2. USING TLV FOR COMPUTING COMPOSITIONS

MODULE Service1(index,operation,db_state)
VAR
  state : {start_st,s10,s11,s12};
INIT
  state=start_st
TRANS
  case
    state=start_st & operation=start_op & index=0: next(state)=s10;
    (index != 1) : next(state) = state;
    (state=s10 & operation = store) : db_state in {d1,d2} &
      (db_state = d1 -> next(state) in {s11,s12}) &
      (db_state = d2 -> next(state) in {s11});
    (state=s11 & operation=pick) : next(state) in {s10};
    (state=s11 & operation=free_mem) : next(state) in {s11};
    (state=s12 & operation=free_mem) : next(state) in {s11};
  esac
DEFINE
  initial := state=start_st & operation=start_op & index = 0;
  failure :=
    index = 1 &
    !((state = s10 & operation=store & db_state in {d1,d2})|
    (state= s11 & operation=pick) |
    (state= s11 & operation = free_mem)|
    (state=s12 & operation=free_mem));
final := state in {s10,s11};

Figure 8.6: Encoding of Available service $S_1$ of Example 8.2.1
2. the VAR section declares, as usual, the variable state ranging over \( S_0 \cup \{\text{start, st}\} \);

3. the INIT section contains the expression 
   \[
   \text{state} = \text{start, st} & \text{op} = \text{start, op};
   \]

4. the TRANS section consists of a case statement very similar to available services’ but adapted to deal with determinism and operation selection. Assume there are \( p \) state-operation pairs \((s^l, o^l)\) \((l = 1, \ldots, p)\) such that there exists a transition \( s^l \xrightarrow{o^l,g^l} s'^l \) (note that if such transition exists it is unique, as the target service is deterministic). In particular, assume that \( s^l = s^0 \) for the first \( q \) pairs, i.e., for \( l = 1, \ldots, q \leq p \). Now, in dependence on \( l \) (and \( s'^l \)), define the set of transitions outgoing from \( s'^l \) (they are as many as operations allowed in \( s'^l \)): \( \rho^l = \{ s \xrightarrow{o,g} t' \in \rho \mid s = s'^l \} \) and rename its elements as \( s'^l \xrightarrow{o^{lr},g^{lr}} s'^{lr} \), for \( r = 1, \ldots, s(l) \). We have:

   \[
   \begin{align*}
   \text{case} & \quad \text{state} = \text{start, st} & \text{op} = \text{start, op} : \\
   & \text{next(state)} = s^0 & \text{next(op)} \in \{ o^1, \ldots, o^q \} & \text{&} \\
   & \text{(next(op)} = o^1 \xrightarrow{g^1}) & \text{&} \\
   & \ldots & \text{&} \\
   & \text{(next(op)} = o^q \xrightarrow{g^q}); \\
   \text{state} = s^1 & \text{& op} = o^1 : \\
   & \text{next(state)} = s^1 & \text{next(op)} \in \{ o^{11}, \ldots, o^{1s(1)} \} & \text{&} \\
   & \text{(next(op)} = o^{11} \xrightarrow{g^{11}} & \text{&} \\
   & \ldots & \text{&} \\
   & \text{(next(op)} = o^{1s(1)} \xrightarrow{g^{1s(1)}}); \\
   \ldots & \\
   \text{state} = s^p & \text{& op} = o^p : \\
   & \text{next(state)} = s^p & \text{next(op)} \in \{ o^{p1}, \ldots, o^{ps(p)} \} & \text{&} \\
   & \text{(next(op)} = o^{p1} \xrightarrow{g^{p1}} & \text{&} \\
   & \ldots & \text{&} \\
   & \text{(next(op)} = o^{ps(p)} \xrightarrow{g^{ps(p)}}); \\
   \text{esac}
   \end{align*}
   \]

Note that simplification can be performed as done in available services, in particular, when guards are identically true implications can be omitted. Moreover, observe how the SMV encoding happens to be nondeterministic, from the operation selection point of view. Actually, this should be unsurprising, as it reflects the uncertainty one has about which operation the client chooses next, at each step. The fact that target encoding becomes nondeterministic is just a technical consequence of moving operations into states, but it brings the same information as in the original TS representation;

5. we add a DEFINE section, where initial and final are defined as exactly as in available services.
8.2. USING TLV FOR COMPUTING COMPOSITIONS

module Target(op, db_state) -- op is an output parameter, db_state is input
VAR
  state : {start_st, t0, t1, t2, t3};
INIT
  state = start_st & op = start_op
TRANS
  case
    state = start_st & op = start_op : next(state) = t0 & next(op) in {pick};
    state = t0 & op = pick : next(state) = t1 & next(op) = play;
    state = t1 & op = play : next(op) in {pick, store} & next(state) = t2 &
      (next(op) = pick -> next(db_state) = d2) &
      (next(op) = store -> next(db_state) = d1);
    state = t2 & op = pick : next(state) = t1 & next(op) in {play};
    state = t2 & op = store: next(state) = t3 & next (op) in {free_mem};
    state = t3 & op = free_mem: next(state) = t0 & next(op) in {pick};
  esac
DEFINE
  initial := state=start_st & op=start_op;
  final := state in {t1}; -- final state(s)

Figure 8.7: Encoding of target service $S_t$ of Example 8.2.1

For an instance of target service encoding, see Figure 8.7.

System. As already discussed in Subsection 8.1.2, we model the system as an unconstrained controller which, except for the initial state, outputs, at each step, an available service index. So, if the community contains $n$ services, the system’s encoding is, trivially, as follows:

1. we create a module named Sys with no parameters;
2. the VAR section declares the variable index ranging over $\{0...n\}$;
3. the INIT section contains the expression ind = 0, where value 0 stands for init in □-game structure;
4. the TRANS section contains, as usual, a case statement, which forces the system to output index 0 at first step and non-zero indices at subsequent steps:

   case
     index = 0 : next(index! = 0);
     index! = 0 : next(index! = 0);
   esac

No DEFINE section is defined, as there is no relevant expression that needs to be explicitly encoded. Refer to Figure 8.8 for an instance with $n = 3$. 
MODULE Sys
VAR
    index : 0..3; -- num of services, 0 used for init
INIT
    index = 0
TRANS
    case
        index=0 : next(index)!=0;
        index!=0 : next(index)!=0;
    esac

Figure 8.8: Encoding of target service $S_t$ of Example 8.2.1

**Environment.** This module represents the abstract environment. As we have seen, when a composition problem is reformulated as a $\square$-game, the environment state is composed of (i) the state of each available service, (ii) the state of the target service and (iii) the operation to be performed next. State transitions are subject to the following rules:

- target service’s component moves to the (only) successor obtained by performing the requested operation in current state, as prescribed by $S_t$’s transition function;
- all but one, i.e., the selected one, available service’s components *move* by looping in their current state (without failing, as they are not selected);
- selected service’s component may either generate a failure or correctly change, according to its respective service transition relation;
- finally, the requested operation changes to one of those the target service can perform in its successor state.

Summing up, in the game formulation, *all components synchronously* move at each step, and all but one loop in their current state. This fact is important to be remarked as, in our encoding, the environment is built upon several modules that, by SMV semantics, run in parallel. The aim of module Env is to combine all modules defined so far and to define when the so-obtained player environment is in an initial state or fails to meet the game rules (while the synthesis goal is to build up a controller such that failures never happen). Given all previous modules, Env assumes the following form:

MODULE Env(index)
VAR
    operation : start_op, $o_1, \ldots, o_{|O|}$;
    db : Databox(operation);
    target : Target(operation, db.state);
    s1 : Service1(index, operation, db.state);
    …
8.2. USING TLV FOR COMPUTING COMPOSITIONS

As anticipated above, the state of this module is composed of (i) the state of variable operation and (ii) the state of all variables declared within each contained module. For implementation reasons, db’s variable state is not exported through a parameter but is referred to as db.state. Note that index is an input parameter, according to environment’s definition in □-game structure, where index is not an environment state variable. A sample instance of Env for the media-library example introduced in Section 4.1 can be found in Figure 8.9.

Figure 8.9: Encoding of environment module combining target and available services of Example 8.2.1

\textbf{Main.} Finally, in main, modules Env, Sys and expression good are put together to build the game. The synthesis algorithm interprets (i) the first declared module as environment, (ii) the second one as system and (iii) the expression good as the invariant property to be guaranteed by the synthesized controller. In particular, it requires the game to be either in its initial state or in a state not yielding an environment failure, that is a state such that the service selected by the system is able to execute the currently requested operation. Module main’s general form, which is instance independent, is reported in Figure 8.10. Note that, variable index is referred to as sys.index and then passed to env as input parameter.
MODULE main
VAR
    env: system Env(sys.index);  
    sys: system Sys;
DEFINE
    good := (sys.initial & env.initial) | !(env.failure);
Chapter 9

Implementation

In this Chapter, we describe a prototypical system, WSCE (Web Service Composition Engine), for automated service composition, which has been actually realized. Its implementation is based on the results obtained in Chapters 8 and preceding ones, and is briefly reported here to show the effectiveness of our approach. Indeed, we show that actual technologies can be exploited to realize a system for automated composition and deployment of (composed) web services, which can be abstracted as in Chapter 4.

9.1 General Architecture and Typical Scenario

Before going into details related to the actual implementation, let us introduce some practical generalities about client-service interaction.

The typical web-service scenario involves a client and a set of services published over the Internet. The client searches for a service that meets its needs, e.g., an online car-rental service, over a set of service providers or service brokers. If a service that meets the requirements is found, then the client starts interacting with it, in order to achieve its goal. A picture of the general architecture for this scenario is shown in Figure 9.1. We point out the

![Figure 9.1: General Service Oriented Architecture](image-url)
difference between service providers and brokers: the former are entities, such as companies, which provide the actual implementation of a set of services, whereas the latter categorize and publish provider’s services and provide search features (for discovery). In principle, brokers are not strictly needed, as a client might search for a service by directly contacting a provider (once its location is known). However, from client’s viewpoint, they represent known centralized repositories where services from several providers are published, thus providing single points of access, which are clearly more convenient than several, possibly non-uniform, providers. Once a service that meets client’s request is found, the broker redirects the client to the service, so that the interaction can start. Of course, such step is not needed if the client directly contacts the service provider.

As for the technologies used in the practice, nowadays service oriented architectures rely on four basic ingredients, that can be considered de-facto standard:

- **XML**, the well-known general-purpose markup language upon which almost all other languages for web service communications are based;

- **SOAP**, which stands for Simple Object Access Protocol, is a lightweight XML-based protocol used to wrap the information exchanged between Web services and clients. SOAP messages can be transferred over several transport protocols, the most used being HTTP;

- **WSDL**, the Web Service Description Language, i.e., an XML-based language suitable for non-behavioral services’ description;

- **UDDI** (Universal Description Discovery and Integration), is an XML-based distributed directory where providers are allowed to register and publish their services, that can be accessed by clients over the internet. A UDDI registry is composed of many nodes, or servers, each compliant with the UDDI specification.

In a nutshell, XML is the basic language used to tag the data, SOAP is the protocol used for message transferring, WSDL is used for service descriptions (stored in UDDI registers) and UDDI is used to list the services available over the Internet. We anticipate that, since services’ behavior is taken into account, WSDL is not suitable for our composition purposes and a new language has been devised.

### 9.2 WSCE

As we said, service composition arises when a requested service is not present in a repository (being it a broker or a provider). When this happens, the
request can be elaborated to check whether services in the repository can be properly combined so to fulfill client’s needs. In order to automatize such a task, a composition engine is required. In the following, we present WSCE (Web Service Composition Engine): a tool that (i) exploits the system TLV to build a composition generator and (ii) actually coordinates services’ execution so to realize the requested (target) service. Currently, WSCE is able to deal with nondeterministic, fully-observable, services in the absence of a data box –hence guards are not present– and without explicit data manipulation (the abstract model is as in Chapter 4). Preliminary experiments, however, show that dealing with a finite-state data box has a minor impact on the system, so it will be likely introduced in the future.

9.2.1 System Overview

Figure 9.2 shows a high-level description of WSCE’s architecture and its interaction with clients. It is composed of two main components:

- Composer;
- Orchestrator Executor.

First, the system takes in input (i) a behavioral description of the target service, i.e., the Target Schema, provided by a client in an appropriate language (see below) and (ii) a representation of the community services, specified in the same language as target services. Then, module Composer translates input services into transition systems and builds a simulation relation, if any. From that, an orchestrator generator is obtained and, finally, executed by module Orchestrator Executor, which interacts with the client and community services in order to realize, in a step-by-step fashion, the target service. Importantly, this last interaction is completely transparent from clients’ viewpoint, i.e., the client acts as exactly as if communicating with a usual community service. Thick arrows in Figure 9.2 represent the information flow related to the synthesis step, whereas interactions among components, at service execution time, are identified by dashed arrows. Observe that, at this stage, we are working at a logical level, i.e., we are not addressing typical issues related to web-services such as deployment or discovery. Besides composing, WSCE is able to deploy the obtained (composed) web services, by making use of standard technology such as Axis. However, this point will be briefly discussed in the following. Now, let us introduce service representation.

9.2.2 Service Representation

An actual language for service description is needed to describe available and target services. WSCE accepts input files –i.e., community and target services–
written in a combination of (i) the ad-hoc language WS-TSL (Web Service Transition System Language), specifically developed to provide a behavioral description of services as transition systems, and (ii) standard WSDL, used to describe service’s interface. WS-TSL is an XML-based language with a straightforward semantics, that allows for describing service states and transitions. Rather than its full specification, we report a sample encoding containing all basic language constructs and covering all relevant cases, i.e.: initial, final and transient states; deterministic and nondeterministic transitions. In Figure 9.3 the transition system associated with a web service is depicted and, in Figure 9.4 the respective WS-TSL encoding is reported. Essentially, a WS-TSL file lists all the states of a transition system along with their \textit{typology} attribute, \textit{initial}, \textit{final} or \textit{transient}, with obvious semantics, and, for each of them, reports all outgoing transitions and respective target states.

Such a language is powerful enough to represent nondeterministic transition systems, indeed it is sufficient, for a given transition, specifying a set of target states, rather than one only. For instance, from state $s_0$, when operation \textbf{search} is executed, service in Figure 9.3 can either remain still in state $s_0$ or move to state $s_1$. This is specified in first \texttt{STATE} section, where a transition under operation (identified by the keyword \texttt{action}, in WS-TSL) \textbf{search} is specified, which contains, in turn, two target states, identified by keyword \texttt{TARGET}.

![Figure 9.2: WSCE abstract architecture](image)
9.2. WSCE

Figure 9.3: A web service transition system.

```xml
<TS
    xmlns='http://www.dis.uniroma1.it/WS-TSL'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://www.dis.uniroma1.it/WS-TSL'
    service="Service1"
>
    <STATE name="S0" typology="initial-final">
        <TRANSITION action="search">
            <TARGET state="S1"/>
            <TARGET state="S0"/>
        </TRANSITION>
    </STATE>
    <STATE name="S1" typology="transient">
        <TRANSITION action="return">
            <TARGET state="S0"/>
        </TRANSITION>
        <TRANSITION action="display">
            <TARGET state="S0"/>
        </TRANSITION>
    </STATE>
</TS>
```

Figure 9.4: WS-TSL encoding of web service TS of Figure 9.3.
9.2.3 System Details

From architecture overview shown in Figure 9.2, it should be clear that module Composer is the system’s core. As anticipated, the synthesis procedure relies on system Tlv, already introduced in Chapter 8, as a tool for safety game’s strategy computation. WSCE is the result of integrating Tlv with WS-TSL. It provides users with (i) a convenient language for service specification and (ii) a tool that automatically solves a composition problem by adopting the procedure shown in Chapter 8. In addition, (iii) WSCE provides an actual runtime environment to deploy and execute the obtained solution, i.e., an orchestrator generator.

Figure 9.5 shows module Composer’s details. It consists of two submodules: Translator and Tlv. The former, given a WS-TSL description of community and target services, i.e., their WS-TSL description as transition systems, automatically builds an SMV game encoding of the composition problem, according to the approach of Chapter 8. The so obtained encoding is then given in input to the Tlv system which returns, if any, a solution –i.e., a strategy or an orchestrator generator– as a textual document. Such output is, essentially, a relational representation of the orchestrator generator (see Chapter 8 for a sample Tlv output), interpreted and executed by the Orchestrator Executor module, which (i) accepts client’s requests and (ii) coordinates community services for actual execution.

More in details, once Tlv has returned the orchestrator generator in a textual format, its output is parsed by the Orchestrator Executor, which builds an internal representation of generator’s structure. Then, target service execution starts, i.e., the Orchestrator Executor interprets the obtained orchestrator generator, which is an FSM, by starting from its initial state:

1. the executor proposes to the client all operations which label some transition outgoing from target service current state;
2. the client, based on its needs, chooses one of such operations;
3. according to the orchestrator generator, the executor selects an available service for operation delegation;
4. the selected service executes the operation and, consequently, changes state;
5. the executor observes the current state of the selected available service (recall that available services are nondeterministic, in general) and performs a state transition (which “follows” target service transition);
6. a new iteration starts from step 1.
In other words, the Orchestrator Executor can be seen as an \textit{interpreter} of orchestrator generators (that can be seen as programs). We remark that available services’ full observability is the fundamental property which allows the executor to track community evolution. When such property lacks, uncertainty arises, which avoids the possibility to reconstruct current community state and, therefore, selecting those services that are actually able to execute the selected operation.

In addition to the core engine of the system, i.e., program modules which implement the abstract logic of the composition task, WSCE implements a subsystem for actual network deployment, which makes both the available and the composed services actually accessible for clients. The implementation of this subsystem relies on the programming language Java and the SOAP implementation Axis, and is actually composed of two modules: a deployer and the Axis SOAP server. A schema is depicted in Figure 9.6. Its main purpose is to publish a composed web service as if it were a usual web service and, upon request, redirect the client to the orchestrator executor i.e., the actual back-end of the published (composed) web service. It works as follows:

- when a client requests a web service, the Axis repository, containing both composed and usual available services, is searched;
- if a service is found which meets client’s requirements, the client is redirected to it. If it is a composed one, the client is redirected to the
CHAPTER 9. IMPLEMENTATION

 CHAPTER 9. IMPLEMENTATION

orchestrator executor, to which, in turn, the orchestrator generator is given in input;

- if no such service is found, the requested service specification is sent to the composer module, which outputs, if any, an orchestrator generator which, in turn, is passed in input to the deployer module;

- the deployer module produces a WSDL plus a WS-TSL description of the generator and puts it in the Axis repository for future access, then the client is redirected to the orchestrator executor;

Since we deal with stateful services, during client-service interactions current state needs to be stored. From a practical standpoint, for such task there are no off-the-shelf solutions, as all standard technologies (including Axis) provide no support for stateful services. This is due to the fact that SOAP is built on top of HTTP, which is a stateless protocol. Thus, after the service fulfills a request sent by the client, the connection is closed and all information about current state are lost. To address this issue, an ad-hoc solution, based on the use of sessions, essentially permanent data structures, has been developed so to associate the orchestrator executor, community services and target service states to a client. Before a connection is closed, the current state of all involved parties is saved. Then, on next client request, the system reconstructs the state of all services at the end of previous interaction with the same client, becoming, thus, able to make its decisions.

Finally, Figure 9.7 shows the screenshot of a running client interacting with a Media-Store service, realized as composition of available services. Notice that some data are present in listed items, so one might argue that services are able to deal with them. In fact, no explicit data manipulation is allowed, in the sense that actual data cannot affect either operations or service behavior. More precisely, operations are allowed to provide data, in response to some
9.3. CURRENT EXTENSIONS

Currently, WSCE is being extended to provide capabilities for distributed orchestration. As discussed in Chapter 7, in distributed scenarios, we are only able to execute an orchestrator, rather than an orchestrator generator, and this is actually what this extension focuses on. As for the underlying technology, the extension is based on ServiceMix: an Enterprise Service Bus (ESB) for Apache, i.e., a software infrastructure, a.k.a. middleware, which provides basic services for complex architectures through an event-driven and standard-based messaging engine. The essential purpose of ESB’s is to provide a communi-
cation layer that allows different software applications to be integrated. As it comes out from Chapter 7, distributed orchestrators, in order to cooperate, need to exchange messages. When dealing with service oriented architectures, a natural solution to communication requirements is represented by middleware infrastructures. So, being ServiceMix one of such infrastructures and an open software framework, it has been selected as the communication layer for our extension.

In a typical client-service session, after client request has been issued, if no available service is found which meets client’s specification, the extended system works as follows:

- a new orchestrator generator is built, e.g., by using the composer described above;
- an orchestrator is generated, possibly taking into account information available at generation time;
- a set of distributed orchestrators is produced, one for each distributed service involved in the composition;
- the distributed orchestration is executed, by relying on the underlying framework for message exchange.

This is a faithful implementation of the procedure presented in Chapter 7. Of course, a natural further extension may be distributing the whole generator.
Chapter 10

Conclusions and Future Work

10.1 Conclusions

This thesis focused on the service composition problem, i.e., the problem of suitably scheduling a set of available services so to realize a desired target service. We considered services that export a behavioral description or, more precisely, a description of the conversations they support when interacting with clients, and took finite-state transition systems as underlying abstract descriptions, following the same approach behind the so-called “Roman Model” ([25, 24]). The goal of this thesis was building solution techniques able to overcome obstacles met by previous PDL-based technique. In particular, we adopted an approach based on the formal notion of simulation relation and, in doing so, introduced a set of significant novelties.

First, though our focus was not on modeling issues, we built a model able to capture a great variety of scenarios. Namely: we introduced nondeterminism in available services and, inspired by [54], included a finite shared memory structure that services can access and modify; moreover, we added multiple operations, i.e., provided services with the capability of executing several operations on a same transition. Such extensions allowed for capturing interesting scenarios such as distributed ones.

Second, as each new model is associated with a respective composition problem, we devised a solution technique for each obtained problem. Such techniques are based on the formal notion of simulation relation which brings two important advantages: it allows for building the whole set of solutions (including infinite ones) and, as a consequence, provides the possibility to generate just-in-time solutions, (or orchestrators), i.e., which are able to react to (some classes of) unexpected situations and take advantage of runtime information. This represents a significant improvement with respect to existing PDL-based solution techniques (cf. [22]), previously proposed, where a single,
pre-compiled, orchestrator, unable to adapt its behavior to exceptional events, was returned. Importantly, the obtained benefits yield no additional computational cost: indeed, the addressed problems are proven EXPTIME-complete and our technique requires exponential time, in the worst case. These achievements constitute the core of our contributions.

Third, we focused on implemented tools for effective problem resolution. As previous work ([22]) witnessed, PDL-based techniques, in addition to above drawbacks, suffer from a lack of actual tools for effective orchestrator computation. In particular, DL reasoners, though effective in checking for the existence of a solution, are not always useful when it comes to synthesizing solutions. We addressed this issue by resorting to actual systems from Verification & Synthesis. Namely, we adopted the tlv (Temporal Logic Verifier) system as a composition engine. In order to do so, a reduction from the service composition problem to the problem of searching for a winning strategy in a safety-game has been built, thus creating a link between the Service Oriented Computing and Verification & Synthesis research fields.

Below, we summarize our main research contributions, which have been published in international journals, proceedings of international conferences and workshops:

- we proposed an approach to behavioral description able to deal a variety of scenarios;
- we developed simulation-based techniques for computing robust and flexible service composition problem’s solutions. Importantly, such techniques bring significant advantages with respect to previous, PDL-based, one, without requiring any additional computational effort (in the worst case);
- we refined the complexity bound obtained in previous work [25, 24];
- an application to a distributed scenario has been proposed;
- a reduction from service composition problem to the problem of finding a winning strategy in a safety-game has been built, which made possible the development of an effective tool, Web Service Composition Engine (WSCE), based on the well-known synthesis system tlv, used as a composition engine.

We remark that, importantly, our approach and techniques are general enough to apply to several areas, such as Web services [29], Behavioral Agents [130], Intelligent Devices [129] or Robot Ecologies [127].
10.2 Future Work

We are currently focusing on observability issues. So far, we assumed that orchestrators have always full observability on community service states, even in distributed contexts, local orchestrators can observe the state of the service they are attached to. However, this assumption may fail in some scenarios. For instance, if an orchestrator infers—rather than observing—current services’ state from, say, their outputs, and different states exist which give raise to a same output message, then the orchestrator may have uncertainty about current community’s state. In such case, we say that states are partially observable, in the sense that only some of them can be known with certainty. When full observability lacks, simulation-based techniques seen so far fail, as orchestrators need to know which operations can be executed by the community at each step—in fact, orchestrators base their decisions on whole histories. Unfortunately, under partial observability, when an observation takes place, an orchestrator knows only a set of possible states, rather than a single certain one, that the community can be in and, consequently, is unable to know which service can be successfully selected for operation delegation. The approach under investigation consists in checking whether an orchestrator that relies only on the certain knowledge about community state can be built. For instance, if, at a certain point, a service can be in two possible states, but both allow a same operation to be executed, then the service can be still selected to execute that operation, no matter which state it is actually in. The abstract procedure to do so amounts to (i) building the asynchronous product of community services, (ii) grouping states that are undistinguishable under a given observation and considering only the common operations they allow and, finally, (iii) executing the usual technique on the so obtained asynchronous product. Preliminary experiments, where problem instances have been encoded in a particular way so to allow partial observability, show that simulation-based techniques can be effective also in this case.

Another interesting research direction deals with planning in AI. All simulation relation variants proposed in this thesis have a common characteristic: at orchestrator’s execution time, they require that each target service transition is actually executed by the service community and, vice versa, that each time the community executes a transition, this corresponds to a target service’s. In other words, the orchestrator cannot delegate operations if not explicitly requested by the client. This means that, from client’s viewpoint, all operations are observable, that is, no internal, silent (a.k.a. \( \tau \)-), actions are ever executed by the community. Consequently, the possibility for orchestrators of autonomously controlling community services so to fulfill client’s requests, is precluded. To see usefulness of such ability, consider Figure 10.1, where service
Figure 10.1: Two behavioral services: 10.1(a) Service $S_a$; 10.1(b) Service $S_b$.

$S_a$ (Subfigure 10.1(a)) represents a target service and $S_b$ (Subfigure 10.1(b)) a community transition system. It is easily seen that $S_a$ is not simulated, in a classical sense, by $S_b$. Indeed, starting with both services in their initial state, when $S_a$ executes $a$, $S_b$ realizes it by executing $a$ from state $s3$. Then, $S_a$ moves to state $s2$ and $S_b$ to state $s4$. Now, from $s4$, $S_b$ is not able to execute neither $b$ nor $c$, which shows that $S_a$ cannot be simulated by $S_b$. However, with a slight intervention on its evolution, $S_b$ can still become able to mimic $S_a$ behavior. Indeed, if the orchestrator would autonomously send $S_b$ a request for operation $a$, the service would execute it and then move to $s5$, where both $b$ and $c$ are executable and, consequently, $S_a$ would still be mimicked. Observe that, from client’s perspective, the target service would be perfectly realized, as silent operation executions do not involve clients at all, so this would result in a significant enhancement of service community potentialities.

As it comes out, synthesis of orchestrators able to autonomously control available services is not trivial. On one side, this is a classical planning problem for reachability [63], where orchestrator’s goal is to execute a plan so to lead the community, from current state, to a desired state which simulates current target’s one. On the other side, computing a simulation relation is still required, as the new state, reached through orchestrator’s plan, is required to simulate target service current state. This ability to “plan for simulation” dramatically improves service community capabilities, by increasing the set of target services actually realizable, as shown by above example.

We also mention the possibility to exploit Alternating-Time Temporal Logics (ATL, ATL*) [5] as an alternative formalism to encode orchestrator synthesis problems. Such logics are interpreted over concurrent game structures, i.e., games where several players cooperate against an environment, which resolves game’s devilish nondeterminism, in order to reach a common goal, that is, maintaining game evolution within a set of desired paths. Informally, the problem can be encoded by identifying players with available services, and the environment with the target service (which also abstracts all possible behaviors of a client). Analogously to our safety-game approach, a solution is
10.2. FUTURE WORK

A strategy for players, which makes services (i.e., players) mimicking target’s behavior. Using ATL logics would open the possibility to exploit further available tools, such as MCMAS\(^1\) as composition engines, so to create an alternative implementation that can be used for, e.g., performance comparison.

Further, a representation issue needs to be addressed. Throughout this work, we assumed an *explicit*, or extensional, state space representation, as the one proposed in Chapter 8, where all operations actually executable from a state and all possible successor states are listed. Clearly, such a representation is very uncomfortable for a human designer, as it does not allow for an intentional description of states and transitions, and, in practice, is not feasible when the number of states grows. Considering that classical model checking is tractable for systems with up to \(10^{20}\) states\(^2\), it comes out that an efficient representation is needed, in order to fully exploit the available tools for verification and synthesis. An appealing alternative would be to represent services based on propositional properties, as customary in AI, where languages such as PDDL (see e.g., [65]) are used to compactly encode a representation of planning domains. This allows to identify executable operations, based on *preconditions*, and their respective successor states based on *postconditions*, that are conditions over modeled properties, rather than explicitly over states. Although, in the practice, such approach allows for a very succinct description of services and dynamic domains, thus facilitating human designers in their task, the new representation might affect resolution performance. Hence, a formal analysis of such impact becomes desirable.

This thesis shows that finite-state systems are a very powerful behavioral model for services, in that they are able to capture a variety of non-trivial composition problems, while guaranteeing the existence of complete solution approaches. However, this is no longer true, in general, when behavioral features are mixed with data-management capabilities. Indeed, on the one hand, when data are introduced without restrictions, one gets undecidability of the service composition problem, whereas, on the other hand, when restrictions are tight enough to yield decidability, the obtained model might be not expressive as originally expected. As usual, we have a trade-off between expressive power and computability. When dealing with data, the challenge is adjusting such trade-off, by finding the right amount of representable data properties, while guaranteeing problem’s decidability. With respect to this issue, a remarkable effort is represented by [23], that can be integrated, as we are currently doing,

\(^1\)http://sse.cs.ucl.ac.uk/projects/mcmas/

\(^2\)The TLV system, that we adopt to implement our composition engine, is based on model checking technology, i.e., essentially, on Ordered Binary Decision Diagrams (OBDD) as a mean for internal service representation, that allows to limit the exponential state space explosion typical of this applications.
Currently, we are carrying on a work inspired by [23], which starts from a model similar to COLOMBO, where: (i) services described as transition systems are able to execute operations with input/output parameters, (ii) operation execution affects (through postconditions) an underlying database and, in turn, (iii) the database itself affects the set of executable operations (through preconditions). On top of such model, we have defined a notion of data-aware simulation relation, i.e., a relation similar to the usual simulation, that takes into account both input/output parameters and the underlying database, with the same intuition as when data are not present: a set of data-aware service community simulates a target service if the former support, as a whole, at least the same conversations as the latter. The difference, however, is what we mean by conversations: when data are not present, they correspond to sequences of atomic operations, whereas, in a data-aware context, they also contain a representation of the input/output behavior. Not only. As for the database, it plays the same role as data box in Chapter 4, but with the fundamental difference of being infinite-state, due to general data infiniteness, which, substantially, precludes the possibility to use the (simulation-based) techniques devised so far, and calls for new approaches for problem resolution.

In order to make the simulation relation actually computable, we adopted a process of abstraction over data similar to the one presented in [23], where equivalence data classes are represented by symbols, so to represent the infinite set of states by means of finitely many equivalence classes, and hence reason over a symbolic finite structure, rather than an infinite actual one. To do so, a set of constraints over the whole system are required. In particular, we assume database’s active domain to have a fixed bound on the size. Observe that such a restriction is not enough to get a finite number of database instances, if the interpretation domain is infinite. Indeed, for a fixed cardinality, an infinite domain admits infinite subsets of that cardinality. So, the same happens for fixed-cardinality active domains that take values from an infinite domain, thus making such case still interesting.
Appendix A

Automatic Verification of Data-Centric Business Processes

In this Appendix, we present a work ([55]) concerning the service composition problem indirectly. Some preliminary results on reasoning about properties of data-structures, namely artifact systems, that evolve over time and interact with a database, are shown. In particular, they are useful for verification of LTL-FO properties, i.e., linear temporal properties of dynamic systems with an underlying relational structure, whose content changes over time in response to the interaction with external services.

The architecture of an artifact system includes a set of atomic services able to interact with a set of business artifacts, or artifacts for short, that is, essentially, data records, with an internal relational structure. In addition, a relational database is present which stores information that are relevant for the system. At each step, based on preconditions over current artifacts’ state, some services can be executable. Upon execution, services may affect the current state of artifacts. They can (i) execute queries over the database to retrieve information, (ii) access, through queries, the internal state of artifacts and (iii) update artifacts, i.e., execute queries that update their state.

The results presented here focus on the verification task and are inspired by an earlier work, namely [56], which concerns the formal verification of Web applications’ temporal properties. We believe that, although developed in an area different from service composition, such results are very relevant to our work as, ultimately, providing an abstraction approach to deal with relational data and infinite-state systems. Also, there are some similarities between this and our approach. First, an infinite ordered set is assumed as the domain which provides new values introduced by the client. Second, at each step a
set of (atomic) services can be executed to manipulate the artifacts, i.e., data structures. Such services can be thought of as operations proposed to clients, similarly to the Roman Model approach. Third, the system as a whole is infinite-state. Finally, data have a relational structure. The major difference is in the possibility of accessing data through arbitrary queries, as opposed to key-based accesses, making the verification task undecidable in the general case, and calling for a set of restrictions over systems’ structure that yield decidability.

A.1 Framework

We assume fixed an infinite, countable domain $D$ equipped with a total dense order $\leq$ with no endpoints. As usual, a database schema $\mathcal{D}$ consists of a finite set of relation symbols with specified arities. The arity of relation $R$ is denoted $a(R)$. An instance, or interpretation, over a database schema, is a mapping associating to each relation symbol $R$ of the schema a finite relation over $D$, of arity $a(R)$.

Given a schema $\mathcal{D}$, $\mathcal{L}_D$ denotes the set of FO formulas over $D \cup \{=, \leq\}$ ($=$ and $\leq$ are built-in relations over $D$). In addition to relations, FO formulas may use a finite set of constants, consisting of elements of $D$. As customary in relational calculus, constants are always interpreted as themselves (this differs from constants in classical logic). If $\varphi(\bar{x})$ is an FO formula with free variables $\bar{x}$, and $\bar{u}$ is a tuple over $D$ of the same arity as $\bar{x}$, we denote by $\varphi(\bar{u})$ the sentence obtained by substituting $\bar{u}$ for $\bar{x}$ in $\varphi(\bar{x})$. Note that, since $D$ is infinite, an FO formula $\varphi(\bar{x})$ may be satisfied by infinitely many tuples $\bar{u}$ over $D$ (so may define an infinite relation). Finiteness and effective evaluation can be guaranteed by using the active domain semantics, in which the domain is restricted to the set of elements occurring in the given instance (sometimes augmented with a specified finite set of constants in $D$, by default empty). For an instance $I$, we denote its active domain by $\text{adom}(I)$. We assume unrestricted semantics unless otherwise specified.

The artifact model uses a specific notion of class, schema and instance, defined next.

**Definition A.1.1** An artifact class is a pair $C = (R, S)$ where $R$ and $S$ are two relation symbols. An instance of $C$ is a pair $I = (R, S)$, where (i) $R$, called attribute relation, is an interpretation of $R$ containing exactly one tuple over $D$, and (ii) $S$, called state relation, is a finite interpretation of $S$ over $D$.

We also refer to an artifact instance of class $C$ as artifact instance, or simply artifact when the class is clear from the context or irrelevant.
**Definition A.1.2** An artifact schema is a tuple

\[ A = \langle C_1, \ldots, C_n, DB \rangle \]

where each \( C_i = \langle R_i, S_i \rangle \) is an artifact class, \( DB \) is a relational schema, and \( C_i, C_j, \) and \( DB \) have no relation symbols in common for \( i \neq j \).

By slight abuse, we sometimes identify an artifact schema \( A \) as above with the relational schema

\[ DB_A = DB \cup \{ R_i, S_i \mid 1 \leq i \leq n \}. \]

An instance of an artifact schema is a tuple of class instances, each corresponding to an artifact class, plus a database instance:

**Definition A.1.3** An instance of an artifact schema

\[ A = \langle C_1, \ldots, C_n, DB \rangle \]

is a tuple \( A = \langle C_1, \ldots, C_n, DB \rangle \), where \( C_i \) is an instance of \( C_i \) and \( DB \) is an instance of \( DB \) over \( D \).

Again by slight abuse, we identify each instance

\[ A = \langle C_1, \ldots, C_n, DB \rangle \]

of \( A \) with the relational instance \( DB \cup \{ R_i, S_i \mid 1 \leq i \leq n \} \) over schema \( DB_A \). Let \( A \) be an artifact schema and \( DB_A \) its relational schema. Given an artifact instance over \( A \), the semantics of formulas in \( L_A \) is the standard semantics on the associated relational instance over \( DB_A \).

**Example A.1.1** We illustrate the expressive power of the artifact model by specifying a scenario where a manufacturer fills customer purchase orders, negotiating the price of each line item on a case-by-case basis. We focus on two artifacts manipulated by the negotiation process, ORDER and QUOTE.

During the workflow, the customer repeatedly adds new line items into (or updates existing ones in) the purchase order modeled by the ORDER artifact. Each line item specifies a product and its quantity. Every tentative line item spawns a negotiation process, in which manufacturer and customer complete rounds of declaring ask and bid prices, until agreement is reached or the negotiation fails. The prices at every round are stored in the QUOTE artifact, which also holds the manufacturer’s initially desired price, the lowest bid he is willing to entertain, and the final negotiated price. Once the negotiation on a tentative line item succeeds, its outcome is scrutinized by a human executive working for the manufacturer. Upon the executive’s approval, the line item is
included into the purchase order. During the negotiation, the manufacturer
consults an underlying database, which lists information about available pro-
ducts (e.g. manufacturing cost) and about customers (e.g. credit rating and
status).

The corresponding artifact system $\Gamma_{ex} = (A, \Sigma)$ is partially described here
and in Example A.1.2 (see Section A.5 for the full specification). For conve-
nience, we allow an artifact class to have several state relations. This can be
easily simulated with a single state relation.

The artifact schema is $A = (ORDER, QUOTE, DB)$, detailed as follows.
$DB = (PRODUCT, CUSTOMER)$ is the database schema, where:

- $PRODUCT(prod_id, manufacturing\_cost, \text{min}_order\_qty)$
  lists product manufacturing cost and minimum order quantity, and

- $CUSTOMER(customer\_id, status, credit\_rating)$ lists customer status
  and credit rating.

$ORDER = (R_O, line\_items, in\_process, done)$ is the artifact class containing the information about a customer’s order.

- $R_O(order\#, customer\_id, need\_by, li\_prod, li\_qty)$
  is the attribute relation holding the order number, the identifier of the
  customer who placed the order, the day it is needed by. The role of
  attributes $li\_prod$ and $li\_qty$ is described later.

- $line\_items(prod\_id, qty)$
  is a state relation that acts as a “shopping cart” holding the collection
  of line items requested so far.

- $in\_process$ and $done$
  are nullary state relations (Boolean flags) keeping track of the stage the
  artifact is in. \footnote{Artifact class ORDER illustrates an extension of Definition A.1.1 that allows several state relations. This extension is for convenience only: it is easy to show a reduction from multiple-state artifacts to single-state artifacts that preserves our decidability result.}

The intention is that, in stage $in\_process$, the customer repeatedly updates
the shopping cart by filling an individual, tentative line item into attributes
$li\_prod$ and $li\_qty$. Subsequently, this line item is inserted into $line\_items$
provided the price negotiation succeeds. When the customer completes the
purchase order, the ORDER artifact transitions to stage $done$. 
A.1. FRAMEWORK

QUOTE = \langle R_Q, \text{l}_i\text{quotes}, \text{idle}, \text{desired\_price\_calc}, \text{negotiation}, \text{approval\_pending}, \text{archive} \rangle

is the artifact class modeling quotes, with:

- $R_Q(\text{order\#}, \text{desired\_price}, \text{lowest\_acceptable\_price},$
  $\text{ask}, \text{bid}, \text{final\_price}, \text{approved}, \text{li\_prod}, \text{li\_qty},$
  $\text{manufacturing\_cost})$
  is the attribute relation.

- $\text{l}_i\text{quotes}(\text{prod\_id}, \text{qty}, \text{price})$
  is a state relation storing the line item with the final negotiated price quotes.

- $\text{idle}, \text{desired\_price\_calc}, \text{negotiation}, \text{approval\_pending}, \text{and archive}$
  are nullary state relations keeping track of the stage the artifact is in.

When inactive, the QUOTE artifact is in state $\text{idle}$, but moves to $\text{desired\_price\_calc}$ as soon as the customer fills in the product id and quantity of a line item. In this stage, desired\_price attribute is set (from the manufacturer's point of view), possibly taking into account the need\_by\_date attribute in the corresponding ORDER artifact and the manufacturing cost listed in the PRODUCT database. During the ensuing negotiation stage, the ask and bid prices are repeatedly set (in attribute ask by the manufacturer, respectively bid by the customer) until a final price is established and recorded in attribute final\_price, or the negotiation fails. Final prices may require approval by a human executive who works for the manufacturer. While approval is awaited, the QUOTE artifact is in stage $\text{approval\_pending}$. Approval is granted by setting Boolean attribute approved. Approved final prices are then archived in state relation $\text{l}_i\text{quotes}$ (while the QUOTE artifact is in stage $\text{archive}$).

We now define the syntax of services. It will be useful to associate to each attribute relation $R$ of an artifact schema $\mathcal{A}$ a fixed sequence $\vec{x}_R$ of distinct variables of length $a(R)$.

**Definition A.1.4** A service $\sigma$ over an artifact schema $\mathcal{A}$ is a tuple $\sigma = \langle \pi, \psi, S \rangle$ where:

- $\pi$, called pre-condition, is a sentence in $\mathcal{L}_A$;

- $\psi$, called post-condition, is a formula in $\mathcal{L}_A$, with free variables $\{\vec{x}_R \mid R$ is an attribute relation of a class in $\mathcal{A}\}$;

- $S$ is a set of state rules containing, for each state relation $S$ of $\mathcal{A}$, one, both or none of the following rules:
APPENDIX A. AUTOMATIC VERIFICATION OF DATA-CENTRIC
BUSINESS PROCESSES

\[
\begin{align*}
-S(\bar{x}) & \leftarrow \phi^+_S(\bar{x}); \\
-\neg S(\bar{x}) & \leftarrow \phi^-_S(\bar{x});
\end{align*}
\]

where \(\phi^+_S(\bar{x})\) and \(\phi^-_S(\bar{x})\) are \(L_A\)-formulas with free variables \(\bar{x}\) s.t. \(|\bar{x}| = a(S)\).

**Definition A.1.5** An artifact system is a pair \(\Gamma = \langle A, \Sigma \rangle\), where \(A\) is an artifact schema and \(\Sigma\) is a non-empty set of services over \(A\).

We next define the semantics of services. We begin with the notion of possible successor of a given artifact instance with respect to a service.

**Definition A.1.6** Let \(\sigma = \langle \pi, \psi, S \rangle\) be a service over artifact schema \(A\). Let \(A\) and \(A'\) be instances of \(A\). We say that \(A'\) is a possible successor of \(A\) with respect to \(\sigma\) (denoted \(A \xrightarrow{\sigma} A'\)) if the following hold:

1. \(A|\pi = \pi\);
2. \(A'|DB = A|DB\);
3. if \(\bar{u}_R\) is the content of the attribute relation \(R\) of \(A\) in \(A'\), then \(A\) satisfies the post-condition \(\psi\) where \(\bar{x}_R\) is replaced by \(\bar{u}_R\) for each \(R\);
4. for each state relation \(S\) of \(A\) and tuple \(\bar{u}\) over \(\text{adom}(A)\) of arity \(a(S)\), \(A' \models S(\bar{u})\) iff

\[
A \models (\phi^+_S(\bar{u}) \land \neg \phi^-_S(\bar{u})) \lor (S(\bar{u}) \land \phi^+_S(\bar{u}) \land \phi^-_S(\bar{u}))
\]

\[
\lor (S(\bar{u}) \land \neg \phi^+_S(\bar{u}) \land \neg \phi^-_S(\bar{u}))
\]

where \(\phi^+_S(\bar{u})\) and \(\phi^-_S(\bar{u})\) are interpreted under active domain semantics, and are taken to be false if the respective rule is not provided.

Note that, according to (2) in Definition A.1.6, services do not update the database contents (thus, the database contents is fixed throughout each run, although it may of course be different across runs). Instead, the data that is updatable throughout a run is carried by the artifacts themselves, as attribute and state relations. This distinction between the static and updatable portions of the data is convenient for technical reasons, as it is used in formulating the restrictions needed for verification (see Section A.2). Note that, if so desired, one can make the entire database updatable by turning it into a state. Also observe that the distinction between state and database is only conceptual, and does not preclude implementing all relations within the same DBMS.

We next define the notion of run of an artifact system \(\Gamma = \langle A, \Sigma \rangle\). An initial instance of \(\Gamma\) is an artifact instance over \(A\) whose states are empty.
Definition A.1.7 A run of an artifact system $\Gamma = \langle A, \Sigma \rangle$ is an infinite sequence $\rho = \{\rho_i\}_{i \geq 0}$ of artifact instances over $A$ (also called configurations) such that:

- $\rho_0$ is an initial instance of $\Gamma$;
- for each $i \geq 0$, $\rho_i \xrightarrow{\sigma} \rho_{i+1}$ for some $\sigma \in \Sigma$.

A pre-run is a finite sequence $\{\rho_i\}_{0 \leq i \leq n}$ satisfying the same conditions as above for $i < n$. We say that a pre-run is blocking if its last configuration has no possible successor.

Example A.1.2 Continuing Example A.1.1, we show how to model the operations allowed on artifacts by the set $\Sigma$ of available services. For ease of readability, we relegate most of the specification of $\Sigma$ to Section A.5, focusing here on the service that models the negotiation process. To illustrate the artifact model’s natural ability to specify processes at different levels of abstraction, we describe the negotiation process at two levels. In a first, coarser cut, the process is abstracted as service $\text{abstract\_negotiation} = \langle \pi_{\text{an}}, \psi_{\text{an}}, S_{\text{an}} \rangle$ about which we only know that the final price is reached when the ask and bid prices coincide, and that it is guaranteed to lie between the allowed margins stored in attributes $\text{desired\_price}$ and $\text{lowest\_acceptable\_price}$ of artifact QUOTE. The specification of this service is relatively simple and given in Section A.5. Alternatively, we show below service $\text{refined\_negotiation} = \langle \pi_{\text{rn}}, \psi_{\text{rn}}, S_{\text{rn}} \rangle$ which refines the negotiation process all the way to the level of individual negotiation rounds, each of which sets the current ask and bid prices.

Conventions We adopt the following conventions:

(i) We model uninitialized attributes by setting them to the reserved constant $\omega$.

(ii) We model Boolean states by nullary state relations, and drop the parentheses from atoms using them: $S()$ becomes $S$. We assume the usual encoding of true as the singleton nullary relation, and false as the empty nullary relation. In particular, all Boolean states are initially false (since all state relations are initially empty).

(iii) For convenience, we use the following syntactic sugar for post-conditions: we write post-conditions as non-Horn rules $h(\bar{x}) := b(\bar{y})$ where the head $h$ is a conjunction of atoms over attribute relations in $A$, with variables $\bar{x}$, and the body $b$ is a formula in $\mathcal{L}_A$ with free variables $\bar{y}$, where $\bar{y} \subseteq \bar{x}$. The semantics is that whenever $A \xrightarrow{\sigma} A'$ holds, $A' \models h(\bar{u})$ for some tuple $\bar{u}$, and $A \models b(\bar{u} | \bar{y})$. Moreover, artifact relations not mentioned in
h remain unchanged. Clearly, this syntactic sugar can be simulated by
the official post-conditions, and conversely.

We now describe service refined negotiations = (π^rn, ψ^rn, S^rn):

The pre-condition

\[ \pi^rn = \text{negotiation} \]

ensures that the service applies only as long as the Boolean state flag negotiation is set in the QUOTE artifact.

Post-condition ψ^rn is given as

\[ R_Q(o,d,l,a,b,f,app,p,q,m) := \\
(\exists a', b' R_Q(o,d,l,a',b', ω, app, p, q, m) \land a' \neq b' \land \\
l \leq a' \land b' \leq b \land f = ω) \\
\lor \\
(\exists a, b, ω, app, p, q, m) \land a = b = f \).

According to the first disjunct, the negotiation is well-formed, i.e. the bid never exceeds the ask price, and in each round, asking prices never increase while bids b never decrease. Moreover, as long as ask and bid price differ, the final price remains undefined (equal to ω). Notice that the values of a and b are otherwise unconstrained, being simply drawn from the infinite domain. This reflects the fact that they are external input from the manufacturer, respectively customer. The second disjunct states that once ask price a and bid price b coincide, the final price f is automatically set to the common value.

S^rn contains rules that, upon detecting successful negotiation, switch the QUOTE artifact to stage approval_pending if the customer does not enjoy preferred status with excellent credit. If he does, then the approval is short-circuited and the QUOTE goes directly to stage archive. The negotiation is successful when the ask and bid prices agree.

approval_pending ←
(\exists o,d,l,a,f,app,p,q,m R_Q(o,d,l,a,a,f,app,p,q,m) \land \\
\exists c, n R_Q(o,c,n,p,q) \land \\
\neg \text{CUSTOMER}(c,"preferred","excellent"))

archive ←
(\exists o,d,l,a,f,app,p,q,m R_Q(o,d,l,a,a,f,app,p,q,m) \land \\
\exists c, n R_Q(o,c,n,p,q) \land \\
\text{CUSTOMER}(c,"preferred","excellent"))

negotiation ← (\exists o,d,l,a,f,app,p,q,m \\
R_Q(o,d,l,a,a,f,app,p,q,m))
Note that state flag negotiation must be set before the state update rules execute (since pre-condition \( \pi^m \) is satisfied). If neither of the state rule bodies is satisfied, then according to the possible successor semantics, the negotiation flag remains set, enabling another negotiation round.

One of the points illustrated by Example A.1.2 is that the artifact model is particularly well-suited for expressing a wide spectrum of abstraction levels desired in specification. This is shown by the two specifications of the negotiation process, one refining it down to individual rounds, the other abstracting it to an atomic sub-task with a post-condition on its outcome. In practice, the motivation for abstraction ranges from lack of information about an external process provided by an autonomous third party as a black box with pre- and post-execution guarantees, to modeling non-deterministic processes governed by chance or human agents rather than by program. There are also technical reasons, such as the undecidability of verification in the presence of arithmetic (as is the case in many settings, including ours). In all these cases, abstracted sub-processes can be naturally modeled as services, leveraging the non-determinism in their post-conditions.

In order to specify temporal properties of runs, we use an extension of linear-time temporal logic (LTL). Recall that LTL is propositional logic augmented with temporal operators such as \( X \) (next), \( U \) (until), \( G \) (always) and \( F \) (eventually). Essentially, the extension we use, denoted LTL-FO, is obtained from LTL by replacing propositions by FO statements about individual artifact instances in the run. The different statements may share variables that are universally quantified at the end. Similar extensions have previously been used in various contexts [61, 2, 137, 56, 57].

**Definition A.1.8** The language \( \text{ltl} \) (first-order linear-time temporal logic) is obtained by closing FO under negation, disjunction, and the following formula formation rule: If \( \varphi \) and \( \psi \) are formulas, then \( X\varphi \) and \( \varphi U \psi \) are formulas. Free and bound variables are defined in the obvious way. The universal closure of an LTL-FO formula \( \varphi(\bar{x}) \) with free variables \( \bar{x} \) is the formula \( \forall \bar{x} \varphi(\bar{x}) \). An LTL-FO sentence is the universal closure of an LTL-FO formula.

Let \( A \) be an artifact schema. An \( \text{ltl} \) sentence over \( A \) is one where each FO component is over \( DB_A \). The semantics of LTL-FO formulas is standard, and we describe it informally. Let \( \Gamma = (A, \Sigma) \) be an artifact system, and \( \forall \bar{x} \varphi(\bar{x}) \) an LTL-FO sentence over \( A \). The artifact system \( \Gamma \) satisfies \( \forall \bar{x} \varphi(\bar{x}) \) iff every run of \( \Gamma \) satisfies it. Let \( \rho = \{\rho_i\}_{i \geq 0} \) be a run of \( \Gamma \), and let \( \rho_{j+} \) denote \( \{\rho_i\}_{i \geq j} \), for \( j \geq 0 \). Note that \( \rho = \rho_{\geq 0} \). The run \( \rho \) satisfies \( \forall \bar{x} \varphi(\bar{x}) \) iff for each valuation \( \nu \) of \( \bar{x} \) in \( D \), \( \rho_{\geq 0} \) satisfies \( \varphi(\nu(\bar{x})) \). The latter is defined by structural induction on
the formula. Satisfaction of an FO sentence $\psi$ by $\rho_i$ is defined in the obvious way. The semantics of Boolean operators is standard. The meaning of the temporal operators $X$, $U$ is the following (where $|$ denotes satisfaction and $j \geq 0$):

- $\rho \geq j | X \varphi$ iff $\rho \geq j + 1 | \varphi$,
- $\rho \geq j | \varphi U \psi$ iff $\exists k \geq j$ such that $\rho \geq k | \psi$ and $\rho \geq l | \varphi$ for $j \leq l < k$.

Observe that the above temporal operators can simulate all commonly used operators, including $F$ (eventually), $G$ (always), and $B$ (before, which requires its first argument to hold before its second argument fails). Indeed, $F \varphi \equiv true U \varphi$, $G \varphi \equiv \neg F(\neg \varphi)$, and $\varphi B \psi \equiv \neg (\neg \varphi U \neg \psi)$. We use the above operators as shorthand in LTL-FO formulas whenever convenient.

Note that, as customary in verification, LTL-FO properties of artifact systems concern exclusively their infinite runs. Thus, blocking finite pre-runs are ignored. In particular, if an artifact system has only blocking pre-runs (so no proper run) then it vacuously satisfies all LTL-FO formulas. For this and other reasons, one may wish to know if, for a given artifact system (i) all of its pre-runs are blocking, or (ii) there exists a blocking pre-run. We consider decidability of these questions at the end of Section A.2 (Corollary A.2.2) and Section A.3 (Corollary A.3.3).

**Example A.1.3** We illustrate desirable properties for the artifact system $\Gamma_{ex}$ in Example A.1.2. These properties pertain to the global evolution of $\Gamma_{ex}$, as well as to the consistency of its specification. One such consistency property requires the state flags $\text{in\_process}$ and $\text{done}$ in class $\text{ORDER}$ to always be mutually exclusive:

$$G(\neg (\text{in\_process} \land \text{done})).$$

A more data-dependent consistency property requires line item quotes archived in state $\text{li\_quotes}$ of the $\text{QUOTE}$ artifact to pertain only to tentative line items previously input by the customer (into attributes $\text{li\_prod}$ and $\text{li\_qty}$ of the $\text{ORDER}$ artifact), and which underwent successful negotiation and approval. Successful negotiation occurs when $\text{ask}$, $\text{bid}$ and $\text{final\_price}$ coincide and the $\text{QUOTE}$ artifact is in state $\text{archive}$:

$$\forall \text{pid, qty, prc}$$

$$G \left( (\exists o, c, n R_O(o, c, n, \text{pid, qty}) \land \right.$$

$$\exists d, l, m R_Q(o, d, l, \text{prc, prc, prc, "yes"}, \text{pid, qty, m}) \land$$

$$\text{archive} \right)$$

$$\text{B}$$

$$\neg \text{li\_quotes(pid,qty,prc)}$$
A.2 DECIDABLE VERIFICATION

Notice the use of the before operator $B$ (requiring its first argument to hold before its second argument fails).

The following property is more semantic in nature, capturing part of the manufacturer’s business model. It requires that if the customer’s status is not ”preferred” and the credit rating is worse than ”good”, then before archiving a line item with final negotiated price lower than the manufacturer’s desired price, explicit approval from a human executive must have been requested. We assume the following ordering on the constants indicating the credit rating: ”poor” < ”fair” < ”good” < ”excellent”.

\[\varphi_3:\]
\[
\forall o,c,n,p,q,d,l,f,m,s,r \quad G ((R_O(o,c,n,p,q) \land \text{in\_process} \land \text{negotiation} \land R_Q(o,d,l,f,f,f,\omega,p,q,m) \land f < d \land \text{CUSTOMER}(c,s,r) \land s \neq ”preferred” \land r < ”good”) \rightarrow (\text{approval\_pending} \land \neg \text{archive} \land \text{li\_quotes}(p,q,f)))
\]

Note that $\varphi_3$ involves both artifacts and the underlying database. If the negotiation process is described by service refined\_negotiation, then the property happens to be satisfied: indeed, recall from its state rules that this service requests approval whenever the customer’s status is not preferred and his credit rating is not excellent. In particular, this applies to customers whose rating is worse than good, according to the above ordering of credit ratings.

A.2 Decidable Verification

In this section we establish the main decidability result on verification of artifact systems.

It is easily seen that satisfaction of an LTL-FO formula by an artifact system is generally undecidable, using Trakhtenbrot’s theorem. To obtain decidability, we introduce a restricted class of artifact systems and LTL-FO properties, called guarded. This is the analog to artifact systems of the input-boundedness restriction, first introduced by Spielmann in the context of ASM transducers [137], and subsequently used for Web service verification [56]. The guarded restriction mainly requires a form of bounded quantification in formulas used in state update rules and LTL-FO properties, together with some additional restrictions. The guarded restriction is formulated as follows.
Definition A.2.1 Let $\Gamma = \langle A, \Sigma \rangle$ be an artifact system. The set of guarded FO formulas over $A$ is obtained by replacing in the definition of FO the quantification formation rule by the following:

- if $\varphi$ is a formula, $\alpha$ is an atom using an attribute relation of some artifact of $A$, $\bar{x} \subseteq \text{free}(\alpha)$, and $\bar{x} \cap \text{free}(\beta) = \emptyset$ for every state atom $\beta$ in $\varphi$, then $\exists \bar{x}(\alpha \land \varphi)$ and $\forall \bar{x}(\alpha \to \varphi)$ are formulas.

An artifact system is guarded iff all formulas used in the state rules of its services are guarded, and all pre-and-post conditions are $\exists^* \text{FO}$ formulas in which all state atoms are ground (i.e. contain only constants). An LTL-FO sentence over $A$ is guarded iff all of its FO components are guarded.

Note that, in addition to the usual bounded quantification conditions, Definition A.2.1 places the restriction $\bar{x} \cap \text{free}(\beta) = \emptyset$ for every state atom $\beta$ occurring in the scope of a quantification of $x$ in a guarded formula. This says that state atoms can only contain constants or free variables in guarded formulas. Together with the fact that state atoms must appear ground in pre-and-post conditions, this places strong restrictions that considerably limit the use of state information. Unfortunately, both restrictions are needed for decidability of verification. On the positive side, as illustrated by our running example, guarded artifact systems appear to remain powerful enough to model significant applications.

Example A.2.1 The artifact system $\Gamma_{ex}$ in our running example is guarded. This includes the complete specification in Section A.5, which shows that the guardedness restriction still offers significant expressive power. For instance, notice that post-condition $\psi_{rn}$ is an $\exists^* \text{FO}$ formula with no non-ground state atoms (trivially so, as it mentions no state at all). In addition, in all state rules the quantified variables appear guarded by atoms using attribute relations $R_Q$ or $R_Q$. No quantified variables appear in any state atom because no such atoms are mentioned. See the state rules of service include line item in Section A.5 for a less trivial example of guarded state rules. There, state atoms do occur in the rule body, but only with non-quantified variables. All properties listed in Example A.1.3 are guarded.

For an example of an unguarded state rule, consider a relaxation of the insertion rule in $S_{rn}$ demanding that executive approval be short-circuited and the archive flag be set for all preferred customers with better than fair rating:

archive ←

$\exists o, d, l, a, f, app, p, q, m, r$

Note that these formulas do not have to obey the restricted quantification formation rule.
The problem here is that quantified variable $r$ does not appear in any attribute atom (indeed it cannot, since neither $R_O$ nor $R_Q$ have any rating attribute). While our undecidability results in Section A.3 imply that not every unguarded rule or property can be equivalently rewritten into a guarded one, it is often possible to do so by slightly modifying the specification, such that it preserves the intended business process semantics, at the cost of widening the attribute relation. This happens to apply here. One can simply extend the attribute schema of $R_O$ to include the customer’s rating, in addition to the originally included customer id. The rating attribute would be set at the same time as the customer id attribute. The latter is set by service initiate_order in Section A.5, using a guarded post-condition that would remain guarded after the proposed extension.

The main result on decidability of verification for artifact systems is the following.

**Theorem A.2.1** It is decidable, given a guarded artifact system $\Gamma$ and a guarded LTL-FO formula $\varphi$, whether every run of $\Gamma$ satisfies $\varphi$. Furthermore, the complexity of the decision problem is PSPACE-complete for fixed arity schemas, and EXPSPACE otherwise.

The main challenge in establishing the above result is that artifact systems are infinite-state systems, due to the presence of unbounded data. To deal with this, the key idea is to develop a concise, symbolic representation of equivalence classes of runs of $\Gamma$, called pseudoruns, that retain just the information needed to check satisfaction of $\varphi$, and can be generated in PSPACE without explicitly constructing any actual run or database. The high-level structure of the proof is similar to the one for decidability of verification for extended ASM transducers [56]. However, the result for the artifact model substantively extends previous ones in two significant ways: (i) runs of artifact systems may use infinitely many domain values (unlike extended ASM transducers where the domain of each run is restricted to the active domain of the finite database), and (ii) the underlying domain is ordered. These extensions require much more care in developing the pseudorun technique, and render the proof of decidability considerably more difficult.

Finally, we consider the issue of blocking pre-runs. As remarked in Section A.1, LTL-FO properties of artifact systems concern only their (infinite) runs

---

3The best lower bound we know for arbitrary arity schemas is CO-NEXPTIME, shown by reduction from validity of $\forall^* \exists^*\mathrm{FO}$ sentences, known to be CO-NEXPTIME-complete [38].
and ignore blocking pre-runs. In particular, if an artifact system has only blocking pre-runs (so no proper run) then it vacuously satisfies all LTL-FO formulas. It therefore becomes of interest to know whether all pre-runs of an artifact system are blocking. Moreover, blocking may also be of interest for reasons specific to the application (see also discussion in Section A.4). We can show the following.

**Corollary A.2.2** It is decidable, given a guarded artifact system $\Gamma$, whether all pre-runs of $\Gamma$ are blocking. Furthermore, the complexity is PSPACE for fixed-arity schemas, and EXPSPACE otherwise.

**Proof:** The result follows immediately from Theorem A.2.1. Indeed, all pre-runs of $\Gamma$ are blocking iff $\Gamma$ has no (infinite) runs iff $\Gamma \models false$. The latter is decidable with the stated complexities by Theorem A.2.1.

One may also wish to know if a given artifact system has some blocking pre-run. Interestingly, this turns out to be undecidable for guarded artifact systems (see Corollary A.3.3).

### A.3 Boundaries of Decidability

In this section we consider several variations of our artifact model and relaxations of the guarded conditions and show that they lead to undecidability of verification. This suggests that the restrictions we present in order to ensure decidability are quite tight. Due to space constraints, the presentation of the alternative models is informal.

**Attributes versus states** We first revisit the distinction between the attribute relation $R$ and the state relation $S$ in artifact classes $C = \langle R, S \rangle$. One might legitimately wonder if the separate treatment is relevant to verification. We next show that this is indeed the case. More precisely, consider a modification of the artifact model where the state $S$ is treated in the same way as $R$, except that $R$ holds a single tuple while $S$ holds an entire relation. In particular, in the definition of a service using artifact class $C = \langle R, S \rangle$:

- the pre-and-post conditions of the service are $\exists^* \text{FO}$ formulas using $R$, $S$ and the database (with $S$-atoms no longer restricted to be ground as previously);
- as before, the initial value of $S$ is empty;
- there are separate post-condition formulas $\psi_R$ and $\psi_S$ for $R$ and $S$, defining their contents in the output ($R$ consists, as before, of one arbitrary
A.3. BOUNDARIES OF DECIDABILITY

tuple satisfying $\psi_R$, while $S$ consists of the set of tuples satisfying $\psi_S$, with active domain semantics to guarantee finiteness).

We refer to $R$ as the tuple attribute set of $C$ and to $S$ as the relational attribute set of $C$. We refer to such artifact systems as hybrid-attribute. Note that, in this model, there are no longer separate state relations. Since there are no states, the guarded restriction on hybrid-attribute services now simply amounts to the $\exists^*\forall^*$FO form of the pre-and-post conditions. The guarded restriction for LTL-FO properties remains unchanged. We can show the following (the proof is by reduction from the Post Correspondence Problem).

Theorem A.3.1 It is undecidable, given a guarded hybrid-attribute artifact system $\Gamma$ and guarded LTL-FO formula $\varphi$, whether $\Gamma \models \varphi$. Moreover, this holds even for singleton artifact systems whose relational attribute set consists of a single attribute, and for a fixed LTL-FO formula $\varphi$ with no variables.

Relaxing the guarded restrictions We now consider several relaxations of the guarded restrictions. It turns out that even very small such relaxations lead to undecidability of verification. Specifically, we consider the following: (i) allowing non-ground state atoms in pre-and-post conditions, (ii) allowing state projections in state update rules (a simple form of un-guarded quantification), (iii) allowing un-guarded quantification in the LTL-FO property, and (iv) extending LTL-FO with path quantifiers.

We can show that each of the relaxations (i)-(iv) leads to undecidability of verification. The proof of (i) is similar to that of Theorem A.3.1. The proofs of (ii) and (iii) are by reduction from the implication problem for functional and inclusion dependencies, known to be undecidable [46]. The proof of (iv) is by reduction from validity of $\exists^*\forall^*$FO sentences, also known to be undecidable [38]. The proofs of (ii)-(iv) can be easily adapted from analogous results obtained for extended ASM transducers [56].

Functional dependencies It is natural to ask whether the decidability of verification holds under the assumption that the database satisfies certain integrity constraints. Unfortunately, we show that even simple key dependencies lead to undecidability.

Theorem A.3.2 It is undecidable, given a guarded singleton artifact system $\Gamma$, a set of functional dependencies $F$ over $DB$, and a guarded LTL-FO sentence $\varphi$, whether $\rho \models \varphi$ for every run $\rho$ of $\Gamma$ on a database satisfying $F$. Moreover, this holds even if $DB$ consists of one binary and one unary relation, and $F$ consists of a single key constraint on the binary relation.

The proof is done by reduction from the PCP, similarly to Theorem A.3.1.
Existence of a blocking pre-run  Recall the question raised in Section A.1: does an artifact system have (i) only blocking pre-runs, or (ii) some blocking pre-run? We showed in Section A.2 that (i) is decidable for guarded artifact systems (Corollary A.2.2). Interestingly, (ii) turns out to be undecidable.

Corollary A.3.3 It is undecidable, given a guarded artifact system $\Gamma$, whether $\Gamma$ has some blocking pre-run.

The result is shown similarly to Theorems A.3.1 and A.3.2, by reduction from the PCP. The key idea is to first search for a match to the PCP (without assurance that the key dependency assumed in Theorem A.3.2 is satisfied), and in case of success make continuance of the run contingent upon violation of the dependency. This reduces the existence of a solution to the PCP to the existence of a blocking pre-run.

Order versus successor  Recall that decidability of verification holds under the assumption that the domain $D$ is countable and equipped with a dense, total order $\leq$ with no endpoints. If $\leq$ is replaced by a successor relation on $D$, verification becomes undecidable. The proof is, again, by reduction from the PCP.

We note that it remains open whether verification remains decidable if some of the assumptions on $\leq$ do not hold, for instance if $\leq$ is not dense.

A.4 Further Applications

We next discuss several problems previously raised in the context of artifact systems, to which our results on verification can be beneficially applied.

Business rules  We consider an extension of the artifact formalism in support of service reuse and customization. In practice, services are often provided by autonomous third-parties, who typically strive for wide applicability and impose as unrestrictive pre-conditions as possible. In contrast, the designer who incorporates third-party services into the business process often requires more control over when these services apply, in the form of more restrictive pre-conditions. Such additional control may also be needed to ensure compliance with business regulations formulated by third parties, independently of the specific application. To address such needs, [31] introduces business rules, which are conditions that can be super-imposed on the pre-conditions of existing services without changing their implementation.

We adopt the notion here and formalize it as follows. Given an artifact system $\Gamma = (A, \Sigma)$, we associate a set $B = \{ \beta_{\sigma} \mid \sigma \in \Sigma \}$ of business rules
A.4. FURTHER APPLICATIONS

to the services in $\Sigma$. A business rule is a sentence in $L_A$, just like a service pre-condition.

For instance, we revisit our running example and assume that order shipment is modeled by the ship service, whose pre-condition only checks that the ORDER artifact is in state done. We also assume the existence of a collect\_payment service, which applies when the ORDER is in state done. Finally, we assume that the ORDER artifact is extended with a paid boolean state flag which is set by the collect\_payment service. Now we wish to super-impose the following business rule, which implements the policy that only platinum customers with excellent credit may get their order shipped before payment is received:

$$\beta_{\text{ship}}:\exists o, c, n, p, s, r\ R_O(o, c, n, p, q) \land \text{CUSTOMER}(c, s, r) \land (s = \text{"platinum"} \land r = \text{"excellent"} \lor \text{paid})$$

Verification under business rules The verification problem for artifact system $\Gamma$ and property $\varphi$ under business rules $B$, denoted $\Gamma \models_B \varphi$, means checking that every run of $\Gamma'$ satisfies $\varphi$, where $\Gamma'$ is obtained by adding each business rule as a pre-condition conjunct to its corresponding service in $\Gamma$. We say that a business rule is guarded if it is guarded when viewed as a service pre-condition. It follows immediately as a corollary of Theorem A.2.1 that verification under $B$ is decidable if $\Gamma$, $\varphi$ and all business rules in $B$ are guarded.

A related problem concerns incremental verification under business rules. Note that, if $\Gamma \models \varphi$, then $\Gamma \models_B \varphi$. However, $\Gamma \not\models \varphi$ does not imply that $\Gamma \not\models_B \varphi$. Thus, properties such as reachability of a configuration satisfying some desired property are not inherited when business rules are added. It is of interest whether such properties can be verified incrementally; however, we do not address this here.

Redundant business rules Towards streamlining the specification, a desirable goal is the removal of redundant business rules. This involves checking whether, given an artifact system $\Gamma = (A, \Sigma)$, a new business rule $\beta$ associated to some service $\sigma \in \Sigma$ has any effect on $\Gamma$, i.e. excludes at least one of its runs. The latter problem amounts to verifying that at any point in a run of $\Gamma$, the pre-condition $\pi$ of $\sigma$ implies $\beta$: $\Gamma \models G(\pi \rightarrow \beta)$. If $\Gamma$ is guarded, and $\beta$ is guarded in the sense of guarded FO components of LTL properties, then, again as a corollary of Theorem A.2.1, checking if $\beta$ has an effect on $\Gamma$ is decidable. Indeed, if $\pi$ is guarded, then we have $\pi \models \exists x\ f(\bar{x})$ with $f$ a quantifier-free
formula in $\mathcal{L}_A$. Then

$$\Gamma \models G(\exists \bar{x} f(\bar{x}) \rightarrow \beta) \text{ iff } \Gamma \models G(\forall \bar{x} \neg f(\bar{x}) \vee \beta)$$

if $\Gamma \models \forall \bar{x} G(\neg f(\bar{x}) \vee \beta)$

where $\varphi$ is a guarded LTL property if $\beta$ is. For example, $\beta_{ship}$ above is guarded.

**Redundant attributes**  Another design simplification consists of redundant attribute removal, a problem raised in [31]. We formulate this as follows. We would like to test whether there is a way to satisfy a property $\varphi$ of runs without using one of the attributes, say $a$, of artifact $A$. Checking redundancy of $a$ reduces to the following verification problem:

$$\Gamma \models \not\models \varphi \rightarrow F(\exists \bar{x} R_A(\bar{x}, a) \land a \neq \omega)$$

where we assume wlog that $a$ is last in $A$’s attribute relation $R_A$. Recall from Section A.1 the convention of representing undefined attributes using a constant $\omega$. The argument of the temporal operator $F$ (eventually) checks that attribute $a$ is defined. If $\varphi$ is guarded and has no global variables (i.e. its FO components are all sentences), then $\varphi'$ is a guarded LTL property. Therefore Theorem A.2.1 applies, yielding decidability.

**Verifying termination properties**  Recall that our semantics of artifact systems and LTL properties ignores blocking runs. However, in some applications, one would like to verify properties relating to termination. As discussed in Section A.2, it is decidable if all pre-runs of an artifact system are blocking (Corollary A.2.2). However, it may be desirable to verify more expressive properties involving blocking configurations. To this end, one can modify the semantics to render all runs infinite by repeating forever blocking configurations, whenever reached. It can be shown that our results continue to hold with this semantics. Note that one can state, within a guarded LTL-FO property, that a configuration of a guarded artifact system is blocking (all variables in negations of the $\exists$ FO pre-conditions become globally quantified universally).

### A.5 Running Example

We present here the full running example used throughout Chapter A. We model a scenario where a manufacturer fills customer purchase orders, negotiating the price of each line item on a case-by-case basis. We focus on two
artifacts manipulated by the negotiation process, ORDER and QUOTE. During the workflow, the customer repeatedly adds new line items into (or updates existing ones in) the purchase order modeled by the ORDER artifact. Each line item specifies a product and its quantity. Line items are first tentatively filled in the ORDER attributes \texttt{li\_prod} and \texttt{li\_qty}. Every tentative line item spawns a negotiation process, in which manufacturer and customer complete rounds of declaring ask and bid prices, until agreement is reached or the negotiation fails. The prices at every round are stored in the QUOTE artifact, which also holds the manufacturer’s initially desired price, the lowest bid he is willing to entertain, and the final negotiated price. Once the negotiation on a tentative line item succeeds, its outcome is scrutinized by a human executive working for the manufacturer. Upon the executive’s approval, the line item is included into the purchase order (by insertion into the ORDER state \texttt{line\_items}), and the final price is archived (in the QUOTE state \texttt{li\_quotes}). During the negotiation, the manufacturer consults an underlying database, which lists information about available products (e.g. manufacturing cost) and about customers (e.g. credit rating and status).

The corresponding artifact system $\Gamma_{ex} = (A, \Sigma)$ is formally described below. As a font convention, we use $R$ to refer to an artifact’s attribute relation and $S$ for state relations.

The artifact schema is $A = (ORDER, QUOTE, DB)$, detailed as follows.

1. $DB = (PRODUCT, CUSTOMER)$ is the database schema, where:
   - \texttt{PRODUCT}(\texttt{prod\_id, manufacturing\_cost, min\_order\_qty}) is the relation containing products and production information, and
   - \texttt{CUSTOMER}(\texttt{customer\_id, status, credit\_rating}) contains information about customers, such as customer status and credit rating.

2. $ORDER = (R_O, \text{line\_items}in\_process\_done, S_O)$ is the artifact class containing the information about a customer’s order:
   - $R_O(\texttt{order\#}, \texttt{customer\_id, need\_by, li\_prod, li\_qty})$ is the attribute relation holding the order number, the identifier of the customer who placed the order, the day it is needed by. The role of attributes \texttt{li\_prod} and \texttt{li\_qty} is described below.
   - $\text{line\_items}(\texttt{prod\_id, qty})$ is a state relation that acts as a “shopping cart” holding the collection of line items requested so far.
• in\_process and done
  are nullary state relations (boolean flags) keeping track of the stage
  the artifact is in.  

The intention is that, in stage in\_process, the customer repeatedly up-
dates the shopping cart by specifying an individual, tentative line item
described by attributes \textit{li\_prod} and \textit{li\_qty}. Subsequently, this line item
is inserted into, deleted from, or replaces in \textit{line\_items} an item with the
same \textit{prod\_id}, provided the price negotiation succeeds. When the cus-
tomer completes the purchase order, the ORDER artifact transitions to
stage done.

3. \textit{QUOTE} = (\textit{R}_Q, \\
\{\textit{li\_quotes, idle, desired\_price\_calc, \\
  negotiation, approval\_pending, archive}\} : S_Q \\
is the artifact class modeling quotes, with:

• \textit{R}_Q(\textit{order\#}, \textit{desired\_price}, \\
  \textit{lowest\_acceptable\_price, ask, bid, \\
  final\_price, approved, li\_prod, li\_qty, \\
  manufacturing\_cost})
  is the attribute relation.

• \textit{li\_quotes}(\textit{prod\_id, qty, price})
  is a state relation holding the final negotiated price quotes for the
  line items in the corresponding ORDER artifact.

• \textit{idle, desired\_price\_calc, negotiation, \\
  approval\_pending, archive}
  are nullary state relations.

When inactive, the QUOTE artifact is in state idle, but moves to de-
sired\_price\_calc as soon as the customer fills in the product id and quantity
of a line item. In this stage, desired\_price attribute is set (from the manu-
facturer’s point of view), possibly taking into account the \textit{need\_by} date
attribute in the corresponding ORDER artifact and the manufacturing
cost listed in the PRODUCT database. During the ensuing \textit{negotiation}
stage, the ask and bid prices are repeatedly set (in attribute \textit{ask} by
the manufacturer, respectively \textit{bid} by the customer) until a final price

\footnote{Artifact class ORDER illustrates an extension of Definition A.1.1 that allows several
state relations. This extension is for convenience only: it is easy to show that it provides
no additional expressive power and preserves our decidability results. Indeed, given artifact
system \(\Gamma\) with multiple states per artifact and LTL-FO sentence \(\varphi\), we can construct in
polynomial time artifact system \(\Gamma'\) and sentence \(\varphi'\) such that \(\Gamma \models \varphi\) iff \(\Gamma' \models \varphi'\). Moreover,
if \(\Gamma\) and \(\varphi\) are guarded, then so are \(\Gamma'\) and \(\varphi'\).}
is established and recorded in attribute final_price, or the negotiation fails. Final prices may require approval by a human executive to whom the negotiator reports. While approval is awaited, the QUOTE artifact is in stage approval_pending. Approval is granted by setting boolean attribute approved. Approved final prices are then archived in state relation li_quotes (while the QUOTE artifact is in stage archive).

The operations allowed on artifacts are modeled by the set Σ of available services, summarized below.

- Service initiate_order = (πio, ψio, Sio) initializes the ORDER artifact, modeling the input of the order number and customer id by the manufacturer, and the need-by date by the customer.

- Service add_or_modify_line_item = (πam, ψam, Sam) models the customer’s choice of a tentative line item to be added to the purchase order, or to replace another line item for the same product. The service records this choice in attributes li_prod and li_qty of the ORDER artifact, and initializes the QUOTE artifact in view of the upcoming negotiation. This involves copying the order number and line item information from ORDER to QUOTE, and filling the QUOTE’s manufacturing_cost attribute with the corresponding value looked up in the PRODUCT database.

- Service include_line_item = (πil, ψil, Sil) includes into the purchase order the current tentative line item, by storing it in ORDER state line_items. The corresponding final negotiated price is archived in QUOTE state li_quotes.

- Service commit_order = (πco, ψco, Sco) simply switches the ORDER artifact to the done stage, which disables any further line item modifications. This service models the customer’s non-deterministic decision to finalize the purchase order.

- Service set_quote_interval = (πsq, ψsq, Ssq) sets the desired_price and lowest_acceptable_price attributes of the QUOTE artifact to frame the subsequent negotiation. This service abstracts a complex sub-task, possibly taking into account the ORDER’s need_by attribute, the manufacturer’s desired profit margin, the customer’s status, and input from a human manager.

- Service quote_approval = (πqa, ψqa, Sqa) models the human supervisor who reviews the quote on behalf of the manufacturer. The process is a black box, about which is only known that it switches the QUOTE artifact to archive stage, and it sets the approved attribute to either "yes" or "no".
APPENDIX A. AUTOMATIC VERIFICATION OF DATA-CENTRIC BUSINESS PROCESSES

To showcase the artifact model’s natural ability to specify processes even partially, we describe the negotiation process at two levels of abstraction.

- In a first, coarser cut, the process is abstracted as service \textit{abstract\_negotiation} = \langle \pi^{an}, \psi^{an}, S^{an} \rangle about which we only know that the final price is reached when the ask and bid prices coincide, and that it is guaranteed to lie between the allowed margins stored in attributes \textit{desired\_price} and \textit{lowest\_acceptable\_price} of artifact QUOTE.

- Alternatively, we use service \textit{refined\_negotiation} = \langle \pi^{rn}, \psi^{rn}, S^{rn} \rangle to refine the negotiation process all the way to the level of individual negotiation rounds, each of which sets the current ask and bid prices.

\textbf{Conventions} We adopt the following conventions:

(i) We model uninitialized attributes by setting them to the reserved constant \( \omega \).

(ii) We model Boolean states by nullary state relations, and drop the parentheses from atoms using them: \( S() \) becomes \( S \). We assume the usual encoding of \textit{true} as the singleton nullary relation, and \textit{false} as the empty nullary relation. In particular, all Boolean states are initially \textit{false} (since all state relations are initially empty).

(iii) For convenience, we use the following syntactic sugar for post-conditions: we write post-conditions as non-Horn rules \( h(\bar{x}) := b(\bar{y}) \) where the head \( h \) is a conjunction of atoms over attribute relations in \( A \), with variables \( \bar{x} \), and the body \( b \) is a formula in \( L_A \) with free variables \( \bar{y} \), where \( \bar{y} \subseteq \bar{x} \). The semantics is that whenever \( A \xrightarrow{\sigma} A' \) holds, \( A' \models h(\bar{x} \leftarrow \bar{u}) \) for some tuple \( \bar{u} \), and \( A \models b(\bar{y} \leftarrow \bar{u} | \bar{y}) \). Moreover, artifact relations not mentioned in \( h \) remain unchanged. Clearly, this syntactic sugar can be simulated by the official post-conditions, and conversely.

Service \textit{initiate\_order} = \langle \pi^{io}, \psi^{io}, S^{io} \rangle initializes the ORDER artifact, where:

- \( \pi^{io} \models \neg \text{in\_process} \),
  i.e. the service applies when the ORDER artifact is not used to process another order;

- The post-condition \( \psi^{io} \) given by

\[
R_{O}(o, c, n, \omega, \omega) := o \neq \omega \wedge n \neq \omega \wedge \\
\exists r, s \text{ CUSTOMER}(c, s, r)
\]
guarantees that the attributes order#, customer_id and need_by are initialized (set distinct from ω). By the semantics of post-conditions, the pick of o, c, n is non-deterministic. The pick of n models the customer’s input, while that of o models the assignment of an order number by the manufacturer. Note that no further constraints are imposed on these values, they are simply picked from the infinite domain. In contrast, the customer c must be one of the existing customers listed in the database relation CUSTOMER. The pick of c also models the manufacturer’s input.

The tentative line item’s price p and quantity q are left uninitialized (they equal ω) and will be set by the customer during an activity modeled by service add_or_modify_line_item below.

- The state rules in $S^{in}$ include the following:
  - in_process ← true,
    an insertion rule that sets the in_process boolean flag.
  - ¬done ← true,
    a deletion rule that resets boolean state flag done, ensuring it is mutually exclusive with in_process. (it is the responsibility of all other services operating on ORDER to keep them so — see add_or_modify_line_item below).

No rule refers to state relation line_items, as no line item exists yet.

Service add_or_modify_line_item models the customer’s choice of a tentative line item to be added to the purchase order, or to replace another line item for the same product. The service affects both the ORDER and the QUOTE artifact, and it is given as $⟨\pi^{am}, \psi^{am}, S^{am}\rangle$, where:

- $\pi^{am} \equiv \exists o, c, n \ \text{in}_{\text{process}} \land R_O(o, c, n, \omega, \omega) \land \text{idle} \land R_Q(\omega, \omega, \omega, \omega, \omega, \omega, \omega, \omega, \omega)$,
  i.e. the service applies only if the ORDER artifact is in the in_process stage, no other line item is currently being processed, and if the QUOTE artifact is currently unused (in the idle stage, with all attributes uninitialized).

- The post-condition $\psi^{am}$ is

$$R_O(o, c, n, p, q) \land R_Q(o, \omega, \omega, \omega, \omega, p, q, m) := \exists q' R_O(o, c, n, \omega, \omega) \land p \neq \omega \land q \neq \omega \land q \geq q' \land \text{PRODUCT}(p, m, q')$$
Note that the customer’s input of product id $p$ and quantity $q$ is modeled as a non-deterministic pick from the infinite domain. The picked $p$ must appear in the PRODUCT catalog stored in the database. The quantity $q$ is less restricted: we only know that it is defined ($q \neq \omega$) and, reflecting the manufacturer’s policy, it exceeds the minimum-order quantity $q'$ listed in the PRODUCT catalog.

According to the post-condition, the service reacts as follows to the customer’s input of the tentative line item. It stores the values $p$ and $q$ into the attributes $li_{prod}$, $li_{qty}$ of the ORDER artifact. Note that attributes $order\#$, $customer\_id$ and $need\_by$ remain unchanged. The service also initializes the $order\#$ attribute of the QUOTE artifact to refer to the corresponding order, and also stores in it $p$, $q$ and the manufacturing cost $m$ for product $p$, which is looked up in the database catalog PRODUCT. The QUOTE artifact’s remaining attributes are left undefined, to be set during negotiation.

- $S^{sm}$ contains no ORDER state rule as the order’s state is left unchanged. It contains the following state rules that move the QUOTE artifact to the desired\_price\_calc stage, which enables the sub-task of quote negotiation:
  
  - insertion rule

  \[
  \text{desired\_price\_calc} \leftarrow \text{true}
  \]

  - deletion rule

  \[
  \neg \text{idle} \leftarrow \text{true}.
  \]

Service $include\_line\_item = (\pi^{il}, \psi^{il}, S^{il})$ includes into the purchase order the current tentative line item, by storing it in ORDER state line\_items. The corresponding final negotiated price is archived in QUOTE state li\_quotes. We have:

- The pre-condition

\[
\pi^{il} \equiv \text{in\_process} \land \exists o, c, n, p, q \ R_O(o, c, n, p, q) \land \\
\ o \neq \omega \land \ c \neq \omega \land \ n \neq \omega \land \ p \neq \omega \land \ q \neq \omega \land \\
\exists d, l, f, m \ R_Q(o, d, l, f, f, f, "yes", p, q, m) \land \\
\text{archive}
\]

ensures that the service applies only if a current line item $p, q$ exists, the ORDER artifact is in stage in\_process, and the QUOTE artifact lists a successful and approved negotiation for this line item and order (notice the common occurrence of $o, p, q$ in both the QUOTE and the ORDER atoms). Successful negotiation occurs when the ask, bid and final price coincide, and the artifact is in state archive. The final price is approved when the approved attribute is set to “yes”.
• The post-condition $\psi_{il}$ given by

$$R_O(o, c, n, \omega, \omega) \land R_Q(\omega, \omega, \omega, \omega, \omega, \omega, \omega, \omega, \omega, \omega, \omega, \omega, \omega, \omega) := \exists p', q' R_O(o, c, n, p', q')$$

guarantees that, regardless of the current value $p', q'$ of the line item, in the successor ORDER artifact the values are reset to undefined ($\omega$), to make room for the next tentative line item. Notice that the ORDER attributes order#, customer_id and need_by are preserved. The QUOTE attributes are all reset, in preparation for the next negotiation.

• The state rules in $S_{il}$ include the following:

  – The state insertion rule

    \[ \text{line_items}(p, q) \leftarrow \exists o, c, n R_O(o, c, n, p, q) \]

    operates on artifact ORDER, inserting the values of attributes \(li\_prod\) and \(li\_qty\) into state relation \(\text{line_items}\).

    The state deletion rule

    \[ \neg \text{line_items}(p, q) \leftarrow \exists o, c, n, q' R_O(o, c, n, p, q') \land \text{line_items}(p, q) \]

deletes any other entry pertaining to the same product $p$ (if any). Recall that, according to the possible successor definition, if state \(\text{line_items}\) already contains an entry for product $p$, the combined effect of the insertion and deletion rule is that of updating the quantity of product $p$ to the latest customer-provided (and successfully negotiated) value. If no prior entry for $p$ exists, then the deletion rule has no effect.

  – The final negotiated price for this line item is archived in QUOTE state \(li\_quotes\) by the following insertion rule:

    \[ li\_quotes(p, q, f) \leftarrow \exists o, d, l, m R_Q(o, d, l, f, f, \"yes\", p, q, m). \]

    The following insertion and deletion rules move the QUOTE artifact to state \text{idle}, signaling its availability for a new negotiation sub-task:

    \[ \text{idle} \leftarrow \text{true} \text{ and } \neg \text{archive} \leftarrow \text{true} \]
Service \( \text{commit\_order} = (\pi^{co}, \psi^{co}, S^{co}) \) simply switches the ORDER artifact to the \text{done} stage, which disables any further line item modifications. This service models the customer’s non-deterministic decision to finalize the purchase order.

- \( \pi^{co} \models \in\_process \land \exists o, c, n, p, q \ R_O(o, c, n, p, q) \land p = \omega \land q = \omega, \)
  i.e. the full order can be committed only if no tentative line item is still being processed (which would make \( p \neq \omega, q \neq \omega \)).

- \( \psi^{co} \) is given by
  \[
  R_O(o, c, n, p, q) := R_O(o, c, n, p, q)
  \]
  i.e. the artifact’s attributes do not change.

- \( S^{co} \) contains only the rules
  \[
  \text{in\_process} \leftarrow \text{false} \quad \text{and} \quad \text{done} \leftarrow \text{true}.
  \]

The following services model the negotiation process.

Service \( \text{set\_quote\_interval} \) sets the \text{desired\_price} and \text{lowest\_acceptable\_price} attributes of the QUOTE artifact to frame the subsequent negotiation. This service abstracts a complex sub-task, possibly taking into account the ORDER’s \text{need\_by} attribute, the manufacturer’s desired profit margin, the customer’s status, and input from a human manager.

\( \text{set\_quote\_interval} = (\pi^{sq}, \psi^{sq}, S^{sq}) \), where:

- \( \pi^{sq} \models \text{desired\_price\_calc} \).

- Post-condition \( \psi^{sq} \) given by
  \[
  R_Q(o, d, l, d, o, \omega, o, \omega, p, q, m) := \\
  d \neq \omega \land l \neq \omega \land d \geq l \geq m \land \\
  R_Q(o, \omega, \omega, \omega, \omega, \omega, \omega, p, q, m).
  \]

models only what is known about the quote generation procedure viewed as a black box: namely that the desired price is higher than the lowest acceptable one, which in turn exceeds the manufacturing cost. It also sets the initial asking price to the desired price in preparation for the negotiation stage.

- The rules in \( S^{sq} \) simply switch the artifact to the \text{negotiation} stage, and are omitted.
A.5. RUNNING EXAMPLE

To showcase the artifact model’s natural ability to specify processes even partially, we describe the negotiation process at two levels of abstraction.

In a first, coarser cut, the process is abstracted as service

\[ \text{abstract\_negotiation} = \langle \pi^{an}, \psi^{an}, S^{an} \rangle \]

about which we only know that the final price is reached when the ask and bid prices coincide, and that it is guaranteed to lie between the allowed margins stored in attributes \textit{desired\_price} and \textit{lowest\_acceptable\_price} of artifact QUOTE.

- \( \pi^{an} = \text{negotiation} \),
  since the process can only start when the QUOTE artifact is ready, which is signaled by setting this state flag.

- Post-condition \( \psi^{an} \), given as

\[
R_Q(o,d,l,f,f,f,app,p,q,m) :=
\exists a', b', f' \ R_Q(o,d,l,a',b',f',app,p,q,m) \land
l \leq f \leq d
\]

guarantees that the final price \( f \) agrees with the final ask and bid prices regardless of their initial values \( a', b' \), and that \( f \) lies between the desired price \( d \) and the lowest acceptable price \( l \).

- We omit the rules in \( S^{an} \), which move the artifact to state \textit{approval\_pending}.

Alternatively, we can refine the negotiation process all the way to the level of individual negotiation rounds, each of which sets the current ask and bid prices. A post-condition ensures that the negotiation is well-formed, i.e. the bid never exceeds the ask price, and that across rounds, asking prices never increase, while bids never decrease. The negotiation is successful when ask and bid prices agree, at which time the QUOTE artifact moves to the \textit{approval\_pending} state, the \textit{final\_price} attribute is set, and and no further rounds are conducted.

Service \textit{refined\_negotiation} = \( \langle \pi^{rn}, \psi^{rn}, S^{rn} \rangle \) is described as follows:

- \( \pi^{rn} = \text{negotiation} \)
  ensures that the service applies only as long as the boolean state flag \textit{negotiation} is set in the QUOTE artifact.

- Post-condition \( \psi^{rn} \) is given as

\[
R_Q(o,d,l,a,b,f,app,p,q,m) :=
(\exists a', b' \ R_Q(o,d,l,a',b',\omega,app,p,q,m) \land
a' \neq b' \land l \leq a' \land b' \leq b \land f = \omega)
\lor
(R_Q(o,d,l,a,b,\omega,app,p,q,m) \land a = b = f).
\]
According to the first disjunct, the negotiation is well-formed, i.e. the bid never exceeds the ask price, and in each round, asking prices $a$ never increase while bids $b$ never decrease. Moreover, as long as ask and bid price differ, the final price remains undefined (equal to $\omega$). Notice that the values of $a$ and $b$ are otherwise unconstrained, being simply drawn from the infinite domain. This models their external input by the manufacturer, respectively customer. The second disjunct states that once ask price $a$ and bid price $b$ coincide, the final price $f$ is automatically set to the common value.

- $S^{rn}$ contains rules that, upon detecting successful negotiation, switch the QUOTE artifact to stage approval. If the customer does not enjoy preferred status with excellent credit. If he does, then the approval is short-circuited and the QUOTE goes to stage archive. The negotiation is successful when ask and bid prices agree.

$$\text{approval} \leftarrow \neg\text{negotiation}\leftarrow (\exists o, d, l, a, f, app, p, q, m \ R_Q(o, d, l, a, a, f, app, p, q, m) \land \exists c, n \ R_Q(o, c, n, p, q) \land \neg\text{CUSTOMER}(c,"preferred", "excellent"))$$

$$\text{archive} \leftarrow (\exists o, d, l, a, f, app, p, q, m \ R_Q(o, d, l, a, a, f, app, p, q, m) \land \exists c, n \ R_Q(o, c, n, p, q) \land \text{CUSTOMER}(c,"preferred", "excellent"))$$

Finally, service quote_approval = ($\pi^{qa}, \psi^{qa}, S^{qa}$) models the human supervisor who reviews the quote on behalf of the manufacturer. The process is a black box, about which is only known that it switches the QUOTE artifact to archive stage, and it sets the approved attribute to either "yes" or "no".

- $\pi^{qa} \doteq \text{approval}$. pending.
- $\psi^{qa}$ is given by
\[ R_Q(o,d,l,f,f,\text{app},p,q,m) := \]
\[ R_Q(o,d,l,f,f,\omega,p,q,m) \land \]
\[ (\text{app} = "yes" \lor \text{app} = "no"). \]

- \( S^{qa} \) comprises the state rules that switch the artifact to stage archive, and are omitted.

We illustrate desirable properties for \( \Gamma_{ex} \), which pertain to its global evolution, as well as to the consistency of the specification.

One such consistency property requires the state flags \textit{in\_process} and \textit{done} in class \textit{ORDER} to always be mutually exclusive:

\[ \mathbf{G}(\neg(\text{in\_process} \land \text{done})). \]

A more data-centric consistency property requires line item quotes archived in state \textit{li\_quotes} of the QUOTE artifact to pertain only to tentative line items previously input by the customer (into attributes \textit{li\_prod} and \textit{li\_qty} of the ORDER artifact), and which underwent successful negotiation and approval. Successful negotiation occurs when \textit{ask}, \textit{bid} and final price coincide and the QUOTE artifact is in state \textit{archive}:

\[ \forall \text{pid}, \text{qty}, \text{prc} \]
\[ \mathbf{G}((\exists o,c,n \ R_Q(o,c,n,\text{pid}, \text{qty}) \land \]
\[ \exists d,l,m \ R_Q(o,d,l,prc,prc,prc,"yes",\text{pid}, \text{qty}, m) \land \]
\[ \text{archive}) \]
\[ \mathbf{B} \]
\[ \neg \text{li\_quotes}(\text{pid}, \text{qty}, \text{prc}) \]

Notice the use of the \textit{before} operator \( \mathbf{B} \) (requiring its first argument to hold before its second argument fails).

The following property is more semantic in nature, capturing part of the manufacturer’s business model. It requires that if the customer’s status is not “preferred” and the credit rating is worse than “good”, then before archiving a line item with final negotiated price lower than the manufacturer’s desired price, explicit approval from a human executive must have been requested. We assume the following ordering on the constants indicating the credit rating:

“poor” < “fair” < “good” < “excellent”.

\[ \varphi_3 : \]
\[ \forall o,c,n,p,q,d,l,f,m,s,r \]
\[ \mathbf{G}((R_Q(o,c,n,p,q) \land \text{in\_process} \land \text{negotiation} \land \]
\[ R_Q(o,d,l,f,f,\omega,p,q,m) \land f < d \land \]
\[ \text{CUSTOMER}(c,s,r) \land s \neq "preferred" \land \]
\[ r < "good" \rightarrow \]
\[
\text{approval\_pending}\]
\[
B
\]
\[
\neg{(\text{archive} \land \text{li\_quotes}(p, q, f))}
\]

Note that \( \varphi_3 \) involves both artifacts and the underlying database. If the negotiation process is described by service \text{refined\_negotiation}, then the property happens to be satisfied: indeed, recall from its state rules that this service requests approval whenever the customer’s status is not preferred and his credit rating is not excellent. In particular, this applies to customers whose rating is worse than good, according to the above ordering of credit ratings.
Bibliography


