Towards automatic recovery in protocol-based Web service composition

Nardjes Menadjelia

N° d'ordre: D.U: 2370

E D S P I: 617

Université Blaise Pascal - Clermont-Ferrand II

École Doctorale des Sciences pour l'Ingénieur de Clermont-Ferrand

Thèse de Doctorat

Présentée par

Nardjes MENADJELIA

pour obtenir le grade de

Docteur d'Université

Spécialité: Informatique

Towards Automatic Recovery in Protocol-based Web Service Composition

Soutenue publiquement le 15 Juillet 2013 devant le jury:

Mme. Marianne HUCHARD	U. Montpellier II	Présidente
M. Mohand-Said HACID	U. Claude Bernard, LIRIS, Lyon 1	Rapporteur
M. Abdelkader HAMEURLAIN	U. Paul Sabatier, IRIT, Toulouse	Rapporteur
M. Franck MORVAN	U. Paul Sabatier, IRIT, Toulouse	Examinateur
Mme. Marinette BOUET	U. Blaise Pascale, Clermont-Ferrand	Examinatrice
M. Farouk TOUMANI	U. Blaise Pascale, Clermont-Ferrand	Directeur de thèse
M. Lhouari NOURINE	U. Blaise Pascale, Clermont-Ferrand	Directeur de thèse

Abstract

In a protocol-based Web service composition, a set of available *component* services collaborate together in order to provide a new *composite* service. Services export their protocols as finite state machines (FSMs). A transition in the FSM represents a task execution that makes the service moving to a next state. An execution of the composite corresponds to a sequence of transitions where each task is *delegated* to a component service. During composite run, one or more delegated components may become *unavailable* due to hard or soft problems on the Network. This unavailability may result in a *failed execution* of the composite. We provide in this thesis a formal study of the *automatic recovery problem* in the protocol-based Web service composition. Recovery consists in transforming the failed execution into a *recovery execution*. Such a transformation is performed by compensating some transitions and executing some others. The recovery execution is an alternative execution of the composite that still has the ability to reach a final state. The recovery problem consists then in finding the best recovery execution(s) among those available. The best recovery execution is attainable from the failed execution with a minimal number of visible compensations with respect to the client. For a given recovery execution, we prove that the decision problem associated with computing the number of invisibly-compensated transitions is NP-complete. Thus, we conclude that deciding of the best recovery execution is in Σ_2^P.

Key words web service protocol, web service composition, recovery, failure, self-healing systems, dependability of systems, complexity, NP-completeness, finite state machines.

Résumée

Dans une composition de services Web basée protocole, un ensemble de services composants se collaborent pour donner lieu à un service Composite. Chaque service est représenté par un automate à états finis (AEF). Au sein d'un AEF, chaque transition exprime l'exécution d'une opération qui fait avancer le service vers un état suivant. Une exécution du composite correspond à une séquence de transitions où chacune est déléguée à un des composants. Lors de l'exécution du composite, un ou plusieurs composants peuvent devenir indisponibles. Ceci peut produire une exécution incomplète du composite, et de ce fait un recouvrement est nécessaire. Le recouvrement consiste à transformer l'exécution incomplète en une exécution alternative ayant encore la capacité d'aller vers un état final. La transformation s'effectue en compensant certaines transitions et exécutant d'autres. Cette thèse présente une étude formelle du problème de recouvrement dans une composition de service Web basée protocole. Le problème de recouvrement consiste à trouver une meilleure exécution alternative parmi celles disponibles. Une meilleure alternative doit tre atteignable à partir de l'exécution incomplète avec un nombre minimal de compensations visibles (vis-à-vis le client). Pour une exécution alternative donnée, nous prouvons que le problème de décision associé au calcul du nombre de transitions invisiblement compensées est NP-Complet. De ce fait, nous concluons que le problème de décision associé au recouvrement appartient à la classe Σ_2^P.

Mots clés Protocole de service Web, composition de services Web, recouvrement, échecs, systèmes à auto-recouvrement, fiabilité des systèmes, complexité, NP-complétude, automates à états finis.

Dedication

A la mémoire de mon frère Abdelbaki.

A tous ceux qui m'aiment et tous ceux que j'aime.

Je dédie ce modeste travail.

Nardjes MENADJELIA

Acknowledgements

Avant tout, j'exprime ma profonde reconnaissance à mon Pays l'Algérie, qui a assuré le financement de ma thèse de Doctorat, et m'a donné l'occasion de l'accomplir dans les meilleurs conditions.

Je tiens à remercier toutes les personnes qui, de près ou de loin, m'ont aidé dans la réalisation de ce travail de thèse de doctorat.

Tout d'abord, mes remerciements s'adressent aux personnes qui m'ont encadré tout au long de ces années d'étude: M. Farouk TOUMANI et M. Lhouari NOURINE. J'apprécie leurs efforts, patience, conseils, et compétence.

Je tiens à exprimer ma profonde gratitude à Mme. Marianne HUCHARD, qui m'a fait l'honneur de présider mon jury de thèse. A mes rapporteurs M. Mohand-Said HACID et M. Abdelkader HAMEURLAIN, et à mes examinateurs Mme. Marinette BOUET et M. Franck MORVAN.

Je souhaite remercier Mme. Séridi HASSINA pour l'intérêt et le soutien chaleureux dont il a toujours fait preuve.

Je tiens à exprimer ma reconnaissance à M. Boualem BENATALLAH, pour sa collaboration et ses conseils précieux qui m'ont bien guidé dans mon travail de recherche.

Je suis très reconnaissante à M. Alain QUILLOT, qui a accepté de m'accueillir dans son laboratoire LIMOS, et à tous les membres et personnels de ce laboratoire qui ont toujours fait preuve de professionnalisme et gentillesse.

Nardjes MENADJELIA

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

Nowadays, the trend in software development field is to design **self-healing** applications [26]. They are applications with an ability to perceive operating anomalies and to **recover** from execution errors automatically [18]. In a self-healing system, a monitoring layer is added to the functional layer. The functional layer implements the main function for which the system is designed. The monitoring layer should have the ability to detect errors occurring in the functional layer, explain and correct them without a human intervention. The procedure of correcting errors in a self-healing system is called an **automatic recovery** or shortly a **recovery**. The recovery guarantees the continuity of the system execution despite the presence of **faults**. It prevents the system **failure** by making it **fault tolerant**. In this work, we focus on the recovery step in a special kind of systems, which is **composite Web services**.

Web services are self-contained, self-describing and modular applications that can be published, located, and invoked across the Web [48]. In our work, a Web service is described by its behavior, also called **the business protocol** [5, 42]. A service protocol describes the valid order of operations invocation. This order specifies, thereby, all possible conversations that a service can have with its partners [1, 5]. We use state machines to model a service protocol where states represent the different phases a service may go through and transitions represent the performed tasks.

Despite the huge number of already published services, it may happen that no single service can meet a specific client requirement. In this case, the need for **Web service composition** raises. Composing services consists to combining a set of available "**component services**" in order to fulfill the client request. The collaboration of these components provides a **composite** Web service. At each execution step of the composite, it makes call to an available service that can perform the actual requested task. This fact raises challenging issue about **availability** of component services during runtime. Being on-line applications, the availability of components cannot be guaranteed because of the vulnerable nature of the Web (e.g., broken connection mediums, crashed servers, etc.). A well designed composite Web service should take this fact into account. However, ensuring the availability of component services seems a very hard challenge for the composite designer because of the privacy and the autonomy of each component service. To this fact, a good solution consists to **tolerate** such **faults** at the composite level by designing a recovery mechanism. The goal behind the recovery is to enable the composite execution continuity in case a component service becomes unavailable at runtime. The execution continuity can be ensured, for instance, by providing an alternative service to the failed one.

1.2 Problematic and contribution

This thesis presents a formal study of the **recovery problem** in the **protocol-based Web service composition**. We consider a scenario of failure in which one or more component services become unavailable at runtime. This may result in an incomplete execution that needs to be repaired. A lot of the works dealing with recovery (e.g., [9], [12], [22], [24], [29], [49], [50], [56] and [62]) provide heuristics to build recovery plans. In some cases (e.g., [29] and [49]), the established recovery plans focus on how to reconfigure the composite structure far from the faulty component(s). In the other cases (e.g., [9], [12], [22], [24], [50], [56] and [62]), recovery plans are used to guide the resulting faulty execution towards a recovery state. A recovery state is either an ”acceptable” termination state (from a client's viewpoint), or a state from which the execution can be continued naturally. Differently to these works, we present in this thesis a formal study of a specific form of recovery problems. We focus on the case where the recovery goal is to repair the faulty composite execution with a minimal number of visible compensations. Therefore, the recovery plan should guide, as transparent as possible, the faulty execution towards a recovery state. The contributions of this thesis are presented in the following sections.

Formal framework for Web service composition

We consider in this work the protocol-based Web service composition model [8, 41] where services export their protocols as finite state machines. States represent different phases a service may go through and transitions represent tasks (or, operations) performed by this service. In the protocol-based composition model, the automatic composition process consists to generate a **Delegator**. From a client's viewpoint, the Delegator mimics the **target**

service behavior by coordinating available services [8, 46]. Formally, the Delegator is an FSM where each transition is annotated with the name of the service delegated to perform the corresponding operation.

In several composition models, component services are considered as atomic elements (e.g., [24], [9], [62], [2] and [12]). Therefore, a component service can be substituted only by another component having the same functionality (e.g., [24], [2] and [9]). In the protocol-based composition model, details about services operations are available. Consequently, the substitution can be defined between operations. This may help to find more substitutes by combining operations belonging to different services. That is, a single faulty service may be substituted by a set of operations belonging to different services. In the same way, a set of faulty services may be substituted by a single service.

A second advantage of the protocol-based representation of services is the presence of semi-final states. A semi-final state expresses a possible partial rollback on the corresponding protocol. A partial rollback cannot be performed on an atomic component.

In the protocol-based composition model, an execution of the Delegator corresponds to a totally ordered set of transitions. To better meet the requirements of this work, we relaxed the execution to a **partially ordered set** (see [38] for example) using data dependencies. Such a vision to the execution concept allows to characterize the recovery transparency and then formalize the recovery problem.

Formalizing unavailability failure in Web service composition

The runtime unavailability of component services in the protocol-based composition model constitutes the failure scenario considered by this work. We formalize the unavailability failure as a set of unavailable (non-executable) transitions where each transition may belong to a different component service. This allows a better generalization by considering all unavailability cases. Those cases include the partial unavailability of one component and the partial/total unavailability of multiple components at the same time. The set of unavailable transitions may have two impacts on the running composition. Firstly, an unsuccessful delegation may occur resulting in a **failed (incomplete) execution** of the composite. Secondly, a set of branches on the Delegator can no more lead to a final state. To deal with such an effect, we propose to clean the Delegator by removing faulty branches. The cleaning allows putting out risky alternatives during recovery.

Formalizing the recovery problem

Formalizing the recovery problem in the protocol-based composition model is the main contribution of this work. As already mentioned, a failed execution cannot lead to a final state. Therefore, the recovery is a process transforming the failed execution into a **recovery execution**. The transformation is performed using **a recovery plan**. The recovery execution is an alternative execution of the Delegator that still has the ability to reach a final state. The **recovery problem** is defined as:

the problem of finding the recovery execution(s) to which the transformation from the failed execution is made with a minimal number of visible compensations.

A compensation is invisible if the compensated transition is replaced by a second transition performing the same operation. Furthermore, the set of substitute transitions must correspond to an ideal in the poset associated with the recovery execution. Similarly, the set of replaced transitions must correspond to an ideal in the poset associated with the failed execution. Therefore, minimizing the number of visible compensations amounts to maximize the number of replaced transitions. Indeed, the **Replacement problem** is defined as:

the problem of computing the number of replaced transitions that can be ensured by some candidate recovery execution.

Solving the recovery problem requires solving the replacement problem for each candidate recovery execution. In this work, we study two variants of the replacement problem: **the strict replacement problem** and **the loose replacement problem**. In the former, the order among the substitute transitions is required to be isomorphic to the order among substituted transitions. In the latter, this constraint is relaxed. Both strict and loose replacement problems have been proven NP-complete. The hardness of the replacement comes from the fact that a transition in the failed execution may have several possible substitutes in the candidate recovery execution. Thus, the choice of one substitute among candidates should be made in a way to maximize the total number of substituted transitions. Based on the complexity results related to the replacement problem, the automatic recovery problem in a protocol-based Web service composition is proven Σ_2^P.

1.3 Thesis outline

Chapter 2 The main goal of this chapter is to overview a set of relevant works dealing with recovery in Web service composition area and neighbor areas such as workflow systems. To better understand the presented state-of-the-art, this chapter is started with a set of clarifications related to key concepts used in this context such as **dependability**, **failures**, **faults**, **fault tolerance**, etc.

Chapter 3 A set of formal definitions is presented in this chapter. These definitions are essentially related to state machines and partial orders.

Chapter 4 In a first part of this chapter, we describe the protocol-based Web service composition model. In the second part, we introduce our extension by relaxing executions to posets.

Chapter 5 This chapter is the core of this work where both the unavailability failure and the recovery problem in the protocol-based Web service composition are detailed and formalized. The main concepts related to the recovery are defined and the complexity issues related to the recovery are discussed.

Chapter 6 Concludes the thesis with the summary of contribution and the major perspectives of this work.

Chapter 2

Unavailability failure and recovery problem in Web service composition

The W3C[1] defines a "Web service" as *"a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically Web Services Description Language, known by the acronym WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."* **Web service composition** means taking advantage of the great number of already published services by **composing** or **combining** them. Such a composition provides a new web service with a new "richer" functionality. Such services are called **the composite web services**. A composite service could be defined as an orchestration (or a coordination) of a set of services in order to satisfy a client requirement that cannot be satisfied by a single service alone [7, 43, 46].

[1] http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/

Services participating in a composition are called **component services**. The greatest benefit of Web service composition is to gain time and development effort by using already published services in creating new richer ones instead of creating them from scratch.

Given the vulnerable nature of the Web, a Web service may encounter **failures** that are characteristic to the special environment supporting it (the Web) in addition to the traditional failures encountered in any software application. A typical example of such failures is the broken connection to the server due to network problems. A severe form of Web service failure is its total **unavailability**. If this service is participating in a composition then its unavailability failure constitutes a **fault** for the composite. It affects the composition consistency and needs to be handled using a **recovery** mechanism. The recovery is a way to implement the **fault tolerance** of a system. The fault tolerance is a system attribute telling whether the failure of the system can be avoided in the presence of faults. In this research work, our interest is focused on the unavailability failure of component services and the associated effects of such a failure on the composite service.

The goal of this chapter is to give an overview of the recovery problem in Web service composition field and some relevant research works that have tackled this problem, especially those dealing with the unavailability failure. For this purpose, this chapter is structured as follows: first of all, we describe the reasons behind web service unavailability failure in Section 2.1. Then, we introduce the ”fault tolerance” concept in Section 2.2. Section 2.3 lists the set of important dimensions serving analysis of the related work. Finally, we resume the related work in Section 2.4 and we briefly compare our vision to the recovery problem to some of the presented works.

2.1 Unavailability failure in composite Web services

From a client's view point, if a system response time is too long with respect to his expectations, then this system is considered unavailable. **Availability** is one attribute of **Dependability** of systems besides other attributes including reliability, safety, security, survivability and maintainability [3, 4, 34, 35, 36, 40]. Dependability is a system property defined in [3] as " *the ability of a computing system to deliver service that can justifiably be trusted. The **service** delivered by a system is its behavior as it is perceived by its user(s).*"

A system **failure** is an event that occurs when the service delivered by the system does not implement the expected system function [3, 4, 15, 34, 35, 36, 44, 45, 47]. **Unavailability failure**, therefore, is a severe form of system failures which occurs when no service is delivered.

Due to their distributed nature, composite Web services greatly face the problem of unavailability failure of component services. Such a failure may be caused by physical defects on the network medium or on the server side [16]. It can also be caused by a software bug at the server level, which makes its response time very long with respect to the requester expectation. Unavailability failure of some service may be perceived as, for instance, a lost message sent to the server, a broken connection to this server or a busy server. In any way, the service cannot be accessed at that time.

Unavailability of component services may affect the composition consistency, which highlights the need for designing a recovery mechanism. It is worth noting that dealing with

unavailability reasons is out of the scope of this work. We rather deal with components' unavailability effects on the composition and we study the recovery from a failed execution.

2.2 Fault tolerance mechanism for composite Web services

A **fault** is the adjudged or hypothesized cause of the failure [4, 34, 35, 36]. It may be an abnormal condition or defect at the component, equipment, or sub-system level that leads to the failure[2]. Fault tolerance (also called: self-repair, self-healing and resilience) for a system is its ability to avoid service failure in presence of faults [4, 34, 35, 36]. A "service" here is defined as in Section 2.1.

It is quite important to note that the fault tolerance in a composite Web service is not to avoid "components failure"; it is rather to avoid the "composite failure". A component failure may be caused, for instance, by a physical fault on the network medium (tolerating such faults is beyond the scope of this thesis). A component failure is a fault for the composite. Thus, avoiding the composite failure consists of tolerating components failure. This can be done by finding, for example, alternative services to the failed ones.

The specifics of Web services and of their composition require special care in the design of supporting fault tolerance mechanisms. They have high dependability requirements due to the following reasons:

(i) The loosely coupled interactions between peers in Web service architecture, which become even more uncontrollable in case of Web service-based conversations [10].

[2]Defined in document ISO/CD 10303-226

(ii) The autonomy of component Web services (each service is designed to work independently of the others). They run on heterogeneous platforms and they have different characteristics (transactional supports, concurrency policies, etc.). This raises challenging issues in specifying the behavior of composite services in the presence of faults [55].

(iii) The interaction with Web services requires dealing with limitations of the Internet. Long responses-time delays of Web service servers and unavailability of components are major issues for composite Web services [55].

Developing fault tolerant mechanisms for composite Web services has been an active area of research over the last couple of years [54, 55]. There exist two basic ways to implement fault tolerance: **backward error recovery** and **forward error recovery** [37, 47]. The backward error recovery has been inspired by transactions principal (the all-or-nothing semantics). It attempts to restore the system state after error occurrence by rolling system components back to a previous correct state [37]. This requires some kind of checkpoints or saved states to which the system can be rolled back. The rollback to a given checkpoint is performed by compensating all operations having been executed after. Compensating an operation means cleaning its effects by executing an operation producing reverse effects. The forward error recovery involves transforming the system components into any correct state [37]. This is done by correcting errors without resorting to reversing the previous operations. The forward error recovery, thereby, usually relies on an exception handling mechanism [10, 15, 19, 24, 28] which requires some kind of fault prediction. The exception handling may substitute failed tasks in order to allow the system to move forward.

For composite Web services, the recovery may combine both the backward and the

forward error recovery in two manners:

(i) If a failed task cannot be retried or no alternative task is found for it, then the execution of this service is aborted and the executed tasks are compensated [61].

(ii) Failed tasks are first compensated and the system is rolled back to a previous consistent state, then the execution is resumed using alternatives to the failed tasks.

In this work, we define a recovery that exploits both backward and forward approaches. We do not use the exception handling mechanism to make a forward recovery but rather, alternatives in our model can be found automatically and dynamically following the failure occurrence. For this purpose, the fault prediction step is not required.

2.3 State-of-the-art analysis dimensions

The main goal of this chapter is to discuss some relevant research works that have tackled the unavailability problem in Web service composition field or neighboring related fields such as workflow systems. This requires highlighting the important dimensions against which different works can be compared with each other and with our work. This section lists our dimensions for analysis and provides definitions to the main concepts related to each of them.

2.3.1 Composition methods

Web services can be composed either (1) manually (in cooperation with domain experts); or (2) automatically (by software programs). We discuss here only the automatic composition methods, summarized in Figure 2.1, and that have been inspired by AI planning in addition

to the workflow models. The automatic composition can be done statically or dynamically.

- **Static Web service composition**: It takes place during design time when the architecture and the design of the software system are planned [21]. The requester should build an abstract process model which includes a set of tasks and the data dependencies among them. This model is carried-out during execution time so that the real atomic Web services fulfilling model tasks are searched. Therefore, only the selection and binding of atomic Web services is done automatically by the program [48]. Two possible approaches exist for the static service composition: Web services orchestration and Web services choreography (see [11] for a survey). In the former, a central coordinator (also called ”orchestrator”) invokes and combines a set of available services. In the latter, Web services choreography does not use a central coordinator, it rather defines an inter-participant conversation and the overall activity is achieved as the composition of peer-to-peer interactions among the collaborating services [11, 13].

  The most commonly used static method is to specify the process model by a graph representation such as Petri nets, finite state machines, etc. For instance, the work presented in [5] uses state machines such that the states represent the different phases that a service may go through during its execution and transitions corresponds to the invocation of a service operation or to its reply.

- **Dynamic Web service composition**: In the dynamic composition, the creation of the process model is done automatically as well as the selection of atomic services [48]. Dynamic composition may be useful when services are discovered at runtime,

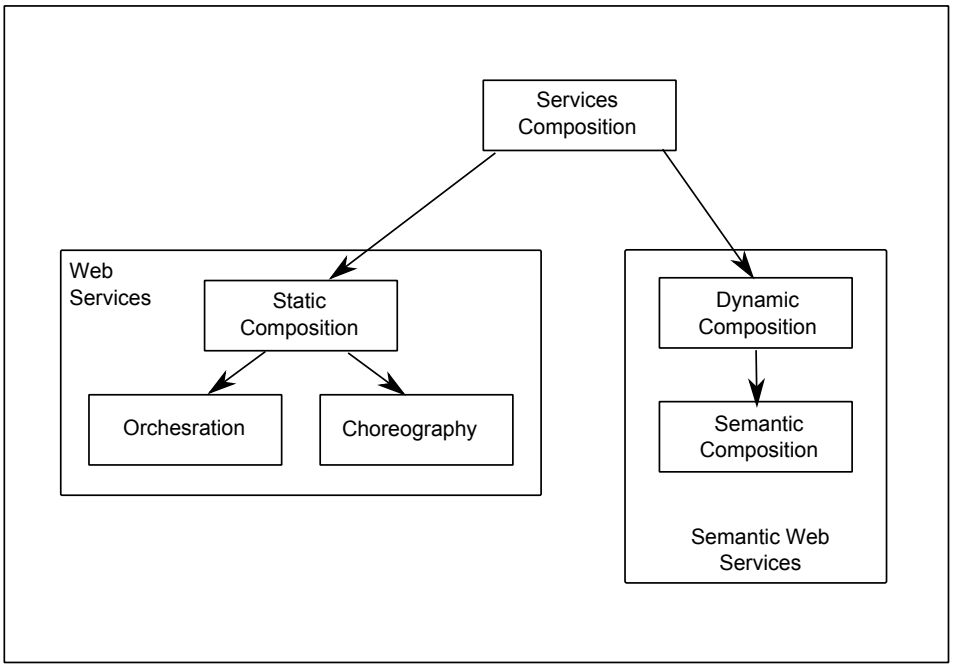

Figure 2.1: Automatic Web Services composition methods [11]

and services' interfaces are unknown [21].

The problem of dynamic Web service composition has been seen as an AI-planning problem (see [20, 21, 48] for a survey), where the user can subscribe for a defined goal, then the composition problem consists of matching the goal with Web services capabilities. In other words, services should be located based on their capabilities and matched together to create the composition i.e., to achieve the goal requested by the user. To this end, services have to provide the abstract descriptions to be discovered which can be overcome using semantic web technologies [11].

2.3.2 Compile time vs. runtime-based recovery approach

This criterion studies whether the recovery approach is fully reactive (at runtime) to failures or if it relies on predefined recovery strategies (during the design phase). In other words, if the recovery plan is fully or partially specified during the design phase of the composite process structure then the recovery approach is compile time-based. Otherwise, it comes to the runtime-based recovery approach. For instance, one way used in the compile time-based recovery approaches is to predefine one (or more) alternative component service to replace another service in case this last fails. In a runtime-based recovery approach, the choice of alternative component services should be done automatically and dynamically following failure occurrence. Generally, the recovery plans can be specified during the process design phase where a lot of information about process execution is available, such as used component services, expected failures, branching probabilities, etc.

2.3.3 Recovery operations

A recovery operation, or a ”recovery strategy” as called in [25], can be simply defined as an elementary step, in the recovery process. According to the used composition model, a recovery operation may be applied either on a component service or on an operation within a component service. In both cases, and without loss of generality, we will designate by ”activity” an element in the composite process, on which a recovery operation can be applied. In literature, most of works dealing with composition failures use some or all of the following recovery operations:

- **Retry (Redo):** Is the simplest way to keep a process running by retrying the failed activity. This operation cannot be applied in case the activity becomes unavailable.

- **Compensate:** When a crucial error occurs during composition, the process will need to go back to a previous consistent state. If some already performed activity is concerned by the rollback then all its generated effects on data should be cleaned or compensated.

- **Cancel (Undo):** Is the simplest form of compensation, when an activity needs to be rolled back without any need to clean its effects on data (because it has no effect).

- **Substitute (replace):** An activity may require to be replaced in case it can no more be used in the process. The alternative activity should be equivalent to the failed one. In some proposals, the alternative can even be richer (in terms of functionality) than the failed one.

- **Abort:** When two activities are executed in parallel (their results will be joined after achieving) and one of them fails then the execution of the second is aborted. Unlike compensation and cancellation, the abortion applies to an activity, still running.

- **Skip:** Skipping a failed activity in a process means bypassing it to its immediate successor. This is generally done by designing for each activity in the process a backup path that avoids its direct successor (if exists). This will allow continuing execution in case the successor fails.

2.3.4 Transactional aspect

In the transactional aspect, we study the **atomicity** of the composite application besides the presence of **transactional properties** on this application model.

Atomicity influences the way in which the recovery plan is designed. In some works, the whole composite service is considered as an atomic application. Therefore, the commit-or-abort decision should be ensured by the recovery plan. In some other works, the atomicity is relaxed such that the composite structure is divided into a sequence of **atomic regions**. Each atomic region is delimited by two checkpoints. In case of faults during execution, a rollback to the last reached checkpoint is performed ensuring, thereby, a partial commitment of the application. A checkpoint can also serve to look for alternative branches.

A transactional property, if taken into account, may prevent an application from making a rollback or to move forward. It tells whether an element of the composition is **compensable**, and whether it is **retriable**. This element can be either a component service or an operation within a component service. Clearly, if some element is not compensable then it cannot be rolled back. In the other hand, if it is not retriable and its first run does not succeed then the application cannot move forward. To this fact, taking the transactional properties into account in some recovery procedure is an important dimension.

2.4 Related work

This section presents some relevant works that addressed the issue of failures in composite behaviors and workflow systems. We discuss, if it is provided, how these works deal with unavailability as a type of failure. We try to provide a description with respect to the above listed analysis dimensions by refining firstly according to the compile time vs. runtime-based recovery approaches.

2.4.1 compile time-based recovery approaches

In the area of Web service composition, Friedrich **et al.** [24] described a two-steps model-based approach to repair the faulty activities in the process. The first step acts before process execution (at compile time) by studying its repairability using information about process structure, data dependencies and available repair actions for each activity among: retry, compensate and substitute. The process structure is modeled as a directed graph in which nodes represent activities and edges represent the control flow where patterns like AND, OR, and AND-split are used. In addition, branching probabilities of activities are modeled in order to be exploited for repairability analysis. The authors propose a heuristic for reasoning about repairability of process activities. The result of this step is a set of non-repairable activities and their impacts on the repairability of the process. For instance, if the branching probability of a non-repairable activity is just 0.1 then its effect on the process repairability can be considered marginal. Note that an activity is repairable if there is an execution of a set of repair actions that produces a correct state of the outputs of this activity. In the second step, the faulty instance of the process is repaired, at run time, by constructing firstly a generic repair plan, then the final repair plan is concluded by applying pruning rules on the generic one, using heuristics. This approach requires hypothesis about availability of some information concerning branching probabilities within the process structure and the failure probability of each of its activities. If a substitution of an activity is required then the designer should specify at least one equivalent activity.

Bhiri **et al.** [9] proposed an approach that ensures user-defined failure atomicity of composite services. Based on composite service skeleton, the proposed approach computes

(at compile time) a set of different kinds of dependencies, namely abortion dependencies, compensation dependencies, cancellation dependencies and alternative dependencies. The composite service skeleton is defined using workflow-like patterns just like [24] previously described. The recovery plan is therefore constructed using the process structure itself. For instance, an $AND-Split$ pattern connecting two component services engenders a cancellation and/or compensation dependency between joint services. Meaning that, if one service among them is aborted or canceled then even the other should be canceled or compensated depending on its effects following its execution. Each kind of dependency influences, then, the way in which a rollback, following a failure, is executed in order to achieve an accepted termination state of the composite. In the aforementioned example, an accepted termination state of the composite is one in which the joint services are aborted together (it is not accepted that only one of them is aborted while the other is achieved naturally).

In the area of workflow, Hamadi and Benatallah [29] suggested a *Self Adaptive Recovery Net*. It is an extended Petri net model for specifying exceptional behavior in workflow systems at compile time. The Petri net is enriched with a set of recovery policies (plans) such that each policy is designed for a specific type of failure. Each recovery policy is stored inside a transition in the Petri Net and fired at runtime following the occurrence of such a failure. They defined eight recovery policies, some of them are designed to recover from a single transition while the others are designed to recover from a whole region. A region contains a set of related transitions.

Another interesting work is that presented in [62] where authors used a transition system to model a static Web service composition. The transition system has a start and an

end node. Each path linking the start node to the end node is a possible execution path. A Web service may have different service levels and may be applied in different execution paths at different levels. A level is a trade-off between the amount of resources a service uses and the output quality such as the result precision. This trade-off is represented as a utility value on each outgoing edge from the node representing a service level on the graph. As a result, each execution path will have a different global utility value which is computed from edges utilities and the optimal path will be chosen for execution. In case a node fails in the current running instance of the process, the approach is to switch to a backup path that bypasses the failed service node. Backup paths are computed for all nodes off-line. Authors use this skip recovery strategy only for the running instance, but they propose to recompute, for the future instances, another globally optimal path without using the failed service in case this failure persists. The backup path cannot be still used in the future since it may achieve only a local optimality. This work handles a single case of failure and cannot deal with multiple faulty nodes. This is also the reason why no compensation mechanism is designed for the case where no backup path is found. Since backup paths are computed at compile time, the designers have the opportunity to ensure that each node has, at least, one backup path that will surely reach the end node successfully (because the first and only failure has already occurred) then no compensation is needed.

Urban **et al.** [56, 60] detailed the Assurance Point model for consistency and recovery in Web service composition. This model is based on the Delta-Grid service composition model deeply described in [59, 57]. The concept of Atomic Group is a basic element in the delta grid model. An atomic group is composed of an operation, its optional compensation and its optional contingency (alternative). A composite group contains, at least, one

atomic and/or composite group. A composite group may have its own compensation and/or contingency activity. The authors distinguish, thereby, the shallow from the deep compensation strategy for a composite group. The former is to execute the global and unique compensation activity defined for the whole group while the latter is to go deeply inside the composite group and to execute the compensation activity of each atomic and/or composite group inside. The concept of Assurance Point enriches the delta grid model with the checkpoints. An assurance point, therefore, is a logical and physical checkpoint for storing data and checking conditions at critical points in the execution of the process. The recovery approach within this model is compile time-based such that recovery plans are stored inside the assurance points. In fact, each time the execution of the process reaches an assurance point, a set of conditions is checked; if they are true a recovery plan is launched. The set of checked conditions are called Integration Rules and they are true only if a failure has occurred. Three possible recovery activities may appear in a recovery plan: retry, rollback and contingency. Note that, a recovery plan may order to recover until the precedent assurance point or even earlier.

In the work presented in [12], authors focused on the fault-tolerant composition of Web services that, in case of some component's failure, it is recovered with a minimal cost in term of compensation. They describe a method to automatically build a fault-tolerant workflow based on rollback dependencies. The authors in [12] restrict the semantics behind a rollback dependency. In fact, their rollback dependency imposes neither an execution order nor a compensation order among concerned services. It just indicates an "all-or-nothing" policy. Each service is associated with a rollback cost that indicates the amount of impact on the whole composition in case this service is rolled-back. For instance, if the "Flight

Reservation" is not a rollback supporting service, its rollback cost will be equal to the whole ticket price. Clearly, the cost is equal to zero if rolling-back a service has no impact. Both costs and rollback dependencies participate in building a workflow that sequences the execution of the services in a way that on a service failure, the mean rollback cost of the service composition becomes minimal. Building this sequencing may also require a supposed-available information about the probability of the service failure at a given position. From which, each permutation of Web services has a distinct recovery cost.

2.4.2 Runtime-based recovery approaches

Simmonds **et al.** in their series of works [50, 51, 52] proposed a framework for runtime monitoring and recovery of BPEL applications. The monitoring is performed against behavioral correctness properties specified as transition systems. A correctness property is violated only when a failure occurs. Following the failure, the proposed framework takes as inputs both the BPEL application (formalized also as a transition system) and the violated property then it outputs a set of ranked recovery plans according to user preferences (in terms of time, cost, etc.). Depending on the type of violated property, a recovery plan may include just the "going back" until an alternative path that avoids the fault can be found. Another violated property may require replanning the achievement of the goal state. Re-planning makes call to the rollback, execution and re-execution of tasks in the application.

Sardina **et al.** [27, 49] proposed a behavior composition model (clearly, a behavior can be a Web service), based on finite state machines and **simulation preorder**. In simulation-

based composition, both the composite and the components export their behaviors as finite state automata. Indeed, the automaton representing the composite is just a virtual description of the protocol that we want to compose. For each operation requested by the composite at a certain level of execution, the computed simulation preorder has the role to show all the components that can perform the current requested operation, in their current states. The composition, thereby, consists in delegating each requested operation to a service already proven capable of performing it, according to the computed simulation. More details about simulation preorder and simulation-based composition are provided in Chapters 3 and 4 respectively.

Following the approach proposed in [27, 49], if a given behavior momentarily freezes (i.e., stops responding) while the simulation suggests an alternative behavior in the current level of execution, then the composition can be continued naturally hoping that the frozen behavior resumes. In case of a permanent unavailability of a participant service, simulation refinement is computed so as to take into account its unavailability. However, the presented work in [49, 27] does not consider the impact of permanent unavailability failure on execution history in case the computed composition has already run partially and the failed service has already participated. In other words, they focus on the impact of unavailability on the already computed simulation and provide an efficient manner to refine it, without studying the possibly happening changes in the executed part of the composition. This works well only if all operations within all behaviors are considered independent, i.e., all states are final and whenever the moment in which a used service fails, it leaves the composition legally.

Another interesting work is [22, 23], where the authors used state machines-based model as well. They consider essentially the problem of the runtime substitution of one service by another, in case the former becomes unavailable. A set of candidate substitutes is firstly selected based on their ontological annotations. They are services having a functionality similar of richer than that provided by the failed service (the service to be substituted). Secondly, one of the candidates is selected based on its ability to be synchronized with the failed service state. Therefore, the failed service state should be proactively stored in order to be provided to the selected substitute. Measuring the ability of a substitute to be synchronized relies on computing its compatibility degree with the service to be substituted. The compatibility degree expresses the ability of the candidate to interpret and use a state provided by the failed service. If the substitute cannot be fully synchronized with the latest state of the unavailable service; the synchronization is then tried using an earlier state than the last stored one.

In this work, authors consider also the case where no synchronization is possible. In such a case, the sequence of messages that have been exchanged between the unavailable service and the client is transparently adapted and replayed on the substitute in order to reproduce the same computation effect (avoiding thereby the user intervention).

In their work, Angarita **et al.** [2] detailed a framework, so called ”FaCETa”, that combines both backward and forward recovery in handling transactional composite failures. They consider transactional-QoS driven composition of Web services where the composite ensures an all-or-nothing semantics and components selection takes into account QoS attributes further than functional and transactional attributes. Transactional attributes of some

component service are expressed in terms of ”compensatable”, ”retriable” and ”pivot”. Their composition model is detailed in [14]. Following failure occurrence, the proposed approach reacts at runtime by trying firstly a forward recovery. In the forward recovery, if the faulty service is retriable then it will be re-executed. If it is not retriable, a substitute service is searched among a set of candidate services. Clearly, the substitute is selected if its functional and non-functional attributes are adequate compared to the faulty service. Finally, if no substitute is found, a backward recovery will ensure to run the system components back until the initial state.

2.4.3 Discussion

In this work, we are interested in a scenario where recovery plans are built at runtime and alternatives to the failed component services are found automatically and dynamically following failure occurrence. We do not provide technical solutions in this thesis. We rather provide a formal study of the recovery problem in the protocol-based Web service composition model. Thereafter, we briefly compare our vision to the recovery problem to some of the works presented above. The comparison is made with respect to the used composition model, the unavailability failure instance, and the recovery problem definition.

Composition model In many works, the composite structure is described by the sequencing over component services which are considered as atomic elements such as in [9, 12, 24, 62]. To better meet the requirements of the recovery problem, we shared with [49] the behavior composition model based on state machines and simulation preorder. Component services export their protocols as finite state machines. Such a model has essentially two advantages:

- It allows reasoning about services operations. The presence of services operations increases substitution possibilities. That is, if no direct alternative is found for some failed service, then some combination of transitions belonging to different services may substitute the failed service. Similarly, a set of failed services may be substituted by a single service.

- The presence of semi-final (or, hybrid) states allows the partial rollback on component services. This possibility is not offered if component services are seen as atomic elements.

Compared to the model described in [7, 8, 49], we consider an additional feature that better serves the recovery process. More specifically, we use information about data dependencies with some differences regarding [9], [12] and [24]:

- The generalization from inter-services dependencies to inter-transitions dependencies. Two dependent transitions may belong to the same service or to different services. Furthermore, to each composite execution corresponds a set of data dependencies that relaxes the execution form to a partially ordered set of transitions. Therefore, the compensation order over executed transitions is relaxed. That is, in a total order, the rollback should be made in the reverse execution sense (last in, first out). In a partial order, the rollback can be made in several senses; as long as no data dependency is violated.

- Compared to [9], we do not differentiate compensation and cancellation dependencies. They are all represented by one sort of dependency that, if it goes from a transition t_1 to another t_2, then t_2 must always be compensated (or, canceled) before t_1.

- In [24], the Web service designer should specify, at least, one alternative activity to that failed. This is equivalently replaced by defining alternative dependencies in [9]. In the work presented by Fredj **et al.** [22, 23], they used services signatures to identify services that offer a functionality similar or richer than that provided by the failed one. This allows choosing candidate substitutes. In our work, we use services protocols for doing so. Substitute transitions can be selected automatically due to the composition model we are using. Concretely, a transition may substitute a second transition if they are labeled by a same operation name. Note that in our work, substituted transitions as well as compensated transitions are not necessarily failed however, this may be applied when needed depending on the computed recovery plan.

Unavailability failure instance In several works (such as in [23] and [49]), the unavailability failure occurs as a set of non-useful component services. Actually, a component service may disappear partially i.e., only a subset of its transitions becomes non-useful. This may happen, for instance, if the component service is itself a composite and some of its own components disappear. To this fact and for a better generalization, we formalize the unavailability failure instance as a set of non-useful transitions. An unavailable transition may belong to any of the component services.

Recovery problem definition All the works presented in sections 2.4.1 and 2.4.2 provide heuristics to build recovery plans. In some cases (such as in [29] and [49]), recovery plans are used to reconfigure the composite structure far from the faulty components. For instance, the reconfiguration is done in [49] by refining the computed simulation preorder

so that all delegations to unavailable services are cleaned out. In some other cases (such as in [9], [12], [22], [24], [50], [56] and [62]), the recovery plan is used to repair the resulting faulty execution by orienting the current execution towards an accepted termination state (from the client's perspective). The goal of the above-cited works is then to specify concrete solutions with respect to their model constraints and chosen quality criteria. Unlike these works, the goal behind our work is mainly to formalize the recovery problem and study its complexity. To this fact, we do not present a concrete or technical solution in this report.

We use the simulation-based composition model such as in [49]. However, we address the repair of a failed execution rather than the computed simulation. We define the recovery as the process of transforming the failed execution into an alternative execution of the composite, using a recovery plan. Clearly, the alternative execution should not make call to an unavailable service in all its possible future evolutions. We define the recovery problem as the problem of finding the alternative execution(s) to which the transformation is made with a minimal number of visible compensations towards the client.

Chapter 3

Preliminaries

In this chapter, we present some formal concepts that will be used later in this thesis. These concepts are essentially related to finite state machines (FSMs) and partially ordered sets (posets). In a first part, Section 3.1 defines concepts related to FSMs. In a second part, Section 3.2 focuses on posets and some concepts connected thereto.

3.1 Finite State Machines [31]

A finite state machine (FSM) is a tuple $A = \langle \Sigma, S, s_0, F, \lambda \rangle$, where :

- Σ is a finite set of alphabet,
- S is the finite set of states,
- $s_0 \in S$ is the initial state,
- $F \subseteq S$ is the set of final states and,
- $\lambda \subseteq S \times \Sigma \times S$ is the transition function of the FSM.

An FSM is said to be **deterministic** (**DFSM**) if whenever $(s,a,s_i) \in \lambda$ and $(s,a',s_j) \in \lambda$ with $s_i \neq s_j$ then $a \neq a'$. Otherwise, it is said **non-deterministic**.

Executions An **execution** of an FSM A is a sequence $\sigma = s_0 \xrightarrow{a_1} s_1 \ldots \xrightarrow{a_n} s_n$ alternating states s_i and operations a_i such that: for $i \in [0,n-1]$, we have $(s_i,a_{i+1},s_{i+1}) \in \lambda$. We say that the execution σ starts at state s_0 and ends at state s_n. The set $\{a_1,\ldots,a_n\}$ is denoted $Op(\sigma)$. We also have $Op(s_{i-1} \xrightarrow{a_i} s_i) = a_i$. The transition $s_{n-1} \xrightarrow{a_n} s_n$ is denoted $last(\sigma)$.

The set $Path(A)$ denotes all possible executions on A. We denote by $trans(\sigma) \subseteq \lambda$ the set of transitions appearing on σ i.e., all $(s_i,a_{i+1},s_{i+1}) \in \lambda$ such that $\sigma = s_0 \xrightarrow{a_1} s_1 \ldots s_i \xrightarrow{a_{i+1}} s_{i+1} \ldots \xrightarrow{a_n} s_n$.

Simulation preorder Let $A = \langle \Sigma, S, s_0, F, \lambda \rangle$ and $A' = \langle \Sigma', S', s'_0, F', \lambda' \rangle$, be two FSMs. A state $s_1 \in S$ is **simulated by** a state $s'_1 \in S'$, noted $s_1 \preceq s'_1$, iff:

- $\forall a \in \Sigma$ and $\forall s_2 \in S$ s.t. $(s_1,a,s_2) \in \lambda$ there is $(s'_1,a,s'_2) \in \lambda'$ s.t. $s_2 \preceq s'_2$ and,
- if $s_1 \in F$, then $s'_1 \in F'$.

We say that A is simulated by A', noted $A \preceq A'$ iff $s_0 \preceq s'_0$.

Note that, a state within A can be simulated by more than one state within A'. The **largest simulation relation** of A by A' is the relation that associates to each state $s \in S$ the set of all states that simulate it.

Asynchronous product Let $R = \{A_1,\ldots,A_k\}$ be a set of k FSMs with $A_i = \langle \Sigma_i, S_i, s_0^i, F_i, \lambda_i \rangle$, $i \in [1,k]$. The **asynchronous product** (or, shuffle) of the A_is, denoted $\odot(R) = A_1 \times \ldots \times A_k$, is the FSM $\langle \Sigma_R, S_R, s_R^0, F_R, \lambda_R \rangle$ with :

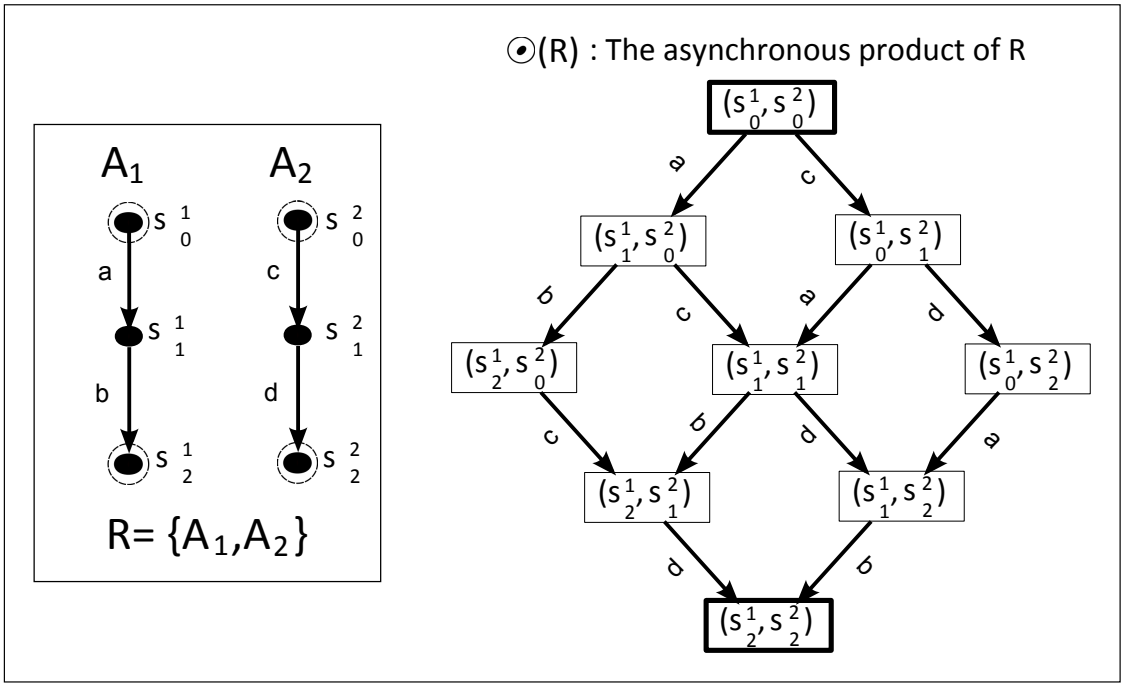

Figure 3.1: Asynchrounous product of two protocols

- $\Sigma_R = \bigcup_{i=1}^{k} \Sigma_i$,
- $S_R = S_1 \times \ldots \times S_k$,
- $s_0^R = (s_0^1, \ldots, s_0^k)$,
- $F_R = F_1 \times \ldots \times F_k$,
- λ_R is the transition function defined as follows: $\lambda_R = \{((s_{i_1}^1, \ldots, s_j^v, \ldots, s_{i_k}^k), a, (s_{i_1}^1, \ldots, s_{j'}^v, \ldots, s_{i_k}^k))$ s.t. $(s_j^v, a, s_{j'}^v) \in \lambda_v$, for some $v \in [1,k]$.

Note that the product is asynchronous in the sense that at each transition of the FSM $A_1 \times \ldots \times A_k$, only one transition of one FSM A_i is moved. An example is depicted in Figure 3.1.

Projection Let $R = \{A_1, \ldots, A_k\}$ be set of FSMs with $A_v = \langle \Sigma_v, S_v, s_v^0, F_v, \lambda_v \rangle, v \in [1,k]$ and let σ be an execution of $\odot(R)$ with $\odot(R) = \langle \Sigma_R, S_R, s_0^R, F_R, \lambda_R \rangle$. A projection of σ on the FSM $A_v, v \in [1,k]$, denoted by $\pi_{A_v}(\sigma)$, is the execution $\sigma' \in Path(A_v)$ such that: $\forall((s_{i_1}^1, \ldots, s_j^v, \ldots, s_{i_k}^k), a, (s_{i_1}^1, \ldots, s_{j'}^v, \ldots, s_{i_k}^k)) \in trans(\sigma) \Leftrightarrow (s_j, a, s_{j'}) \in trans(\sigma'), v \in [1,k]$. Informally, an execution of $\odot(R)$ involves a set of executions on participant FSMs. Then, the projection of σ on A_v is the execution of A_v involved by σ.

3.2 Partial orders and ideals [17]

A partial order (or, a **poset**) is a binary relation "$\leq$" over a set X, denoted $P = (X, \leq)$, which is reflexive, antisymmetric, and transitive, i.e., for all a, b, and c in X, we have:

- $a \leq a$ (reflexivity),
- if $a \leq b$ and $b \leq a$ then $a = b$ (antisymmetry),
- if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity).

In other words, a partial order is an antisymmetric preorder.

A **totally ordered set** (also called a **linearly ordered set** or a **chain**) is a poset $P = (X, \leq)$ in which $\forall a, b \in X, a \leq b$ or $b \leq a$, means that any pair of elements in X are mutually comparable under the relation $\leq$.

Ideals Let $P = (X, \leq)$ be a partial order. $I \subseteq X$ is said to be an **ideal** of P if for every $y \in I$, if $x \leq y$ then $x \in I$. The set of all ideals of P is denoted $\mathfrak{I}(P)$. An example of an ideal of a poset P is depicted in Figure 3.2.

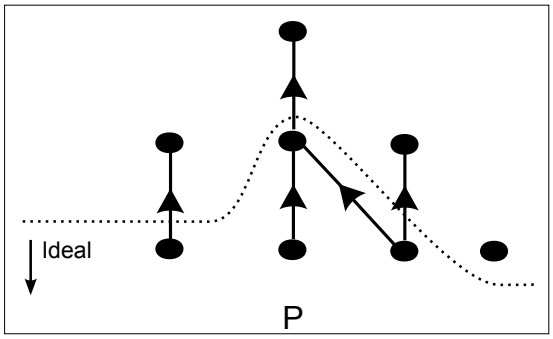

Figure 3.2: An ideal of a poset

Chapter 4

Web Service Composition Model

The need for Web service composition raises when no available service can meet a specific client requirement. The composition consists in combining a set of available services, that we call **component services**, with the purpose of building a desired service, referred to as the **target service** [8, 46]. A lot of models for Web services and their composition have been developed [11, 21, 48, 53]. In this work, we adopt a **protocol-based composition model** [8, 41]. In such a model, services export their protocols as FSMs. The composition process results in a **Delegator**. It coordinates available services executions so to mimicking the target service behavior [8, 46].

A composition of Web services may encounter problems due to the vulnerable nature of the Web. Component services may disappear at runtime resulting in an incomplete execution of the composite. In this work, we focus on the recovery of an incomplete execution. The recovery may use the compensation in order to set back the execution in a consistent state. In the protocol-based composition model, an execution of the Delegator corresponds to a totally ordered set of transitions. To better meet the requirements of this work, we relaxed the execution to a **partially ordered set** using **data dependencies**. This relaxes the compensation order among the transitions of a given execution. That is, the rollback in a total order should always be made in the reverse execution sense (last in, first out). In a partial order, the rollback can be made in several senses; as long as no precedence constraint is violated.

The goal of this chapter is to describe the composition model used in this work. Section 4.1 describes the protocol-based representation of Web services. Then, Section 4.2 details the protocol-based composition model. In Section 4.3, we re-formalize the execution concept using partial orders.

4.1 Protocol-based Web service modeling

In a protocol-based representation [5, 6, 7, 8, 46], a Web service exports its conversational behavior i.e., the sequences of atomic interactions that services can potentially have with clients. A client can be a human or another service. Behaviors are represented by means of deterministic finite state machines (DFSMs), where each transition label refers to a possible atomic interaction between a client and an available service, also referred to as **operation**.

The states indicate the phases that a service can go through while final states indicate correct haltings. Note that final states are surrounded by a circle in the subsequent figures. Typically, an atomic interaction results from the following steps [46]:

- According to its current state, the available service proposes a choice of operations the client can ask for;
- the client selects one of such operations;
- the available service executes client's selection, moves to a new state, according to its behavioral specification, and iterates the procedure.

An execution of a service protocol formalized by means of a DFSM $A = \langle \Sigma, S, s_0, F, \lambda \rangle$ (Σ symbolizes service operations) is an execution of A. We consider deterministic protocols 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.

Example 1. (A service protocol) *Consider the Retailer service in Figure 4.1. Its protocol deals with two types of clients: regular and premium. In the former case, a regular client asks for a product catalog then he makes an order. Clearly, a client can directly makes the order without going through the catalog. An invoice is sent to the client then a payment phase is required. Finally, the requested goods will be shipped. If the client is premium then goods may be shipped immediately after placing an order (the invoice is sent later).*

The Repository of available services A repository, denoted R, of available services is a set of services that are directly available to the client and can be used for the composition. Formally, it is a set of protocols, each of which is represented by a DFSM i.e.,

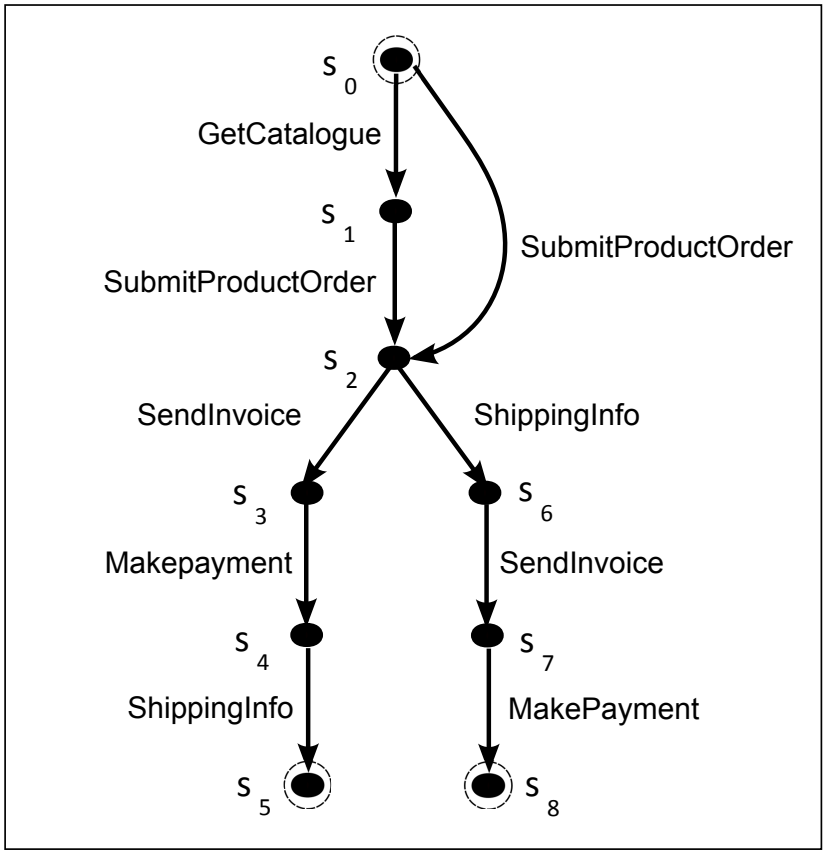

Figure 4.1: The business protocol of the *retailer* service

$R = \{A_1, A_2, \ldots, A_k\}$. All the services in R share a common understanding over the set of operations in the alphabet Σ_R with $\Sigma_R = \bigcup_{i=1}^{k} \Sigma_i$. This means, two operations labeled similarly are performing the same task whatever these operations belong to same or different services. In the subsequent paragraphs, we use shortly the term "repository" to designate a repository of available services.

Example 2. (Repository) *Consider the repository depicted in Figure 4.2. Four services are available: an authentication service, two equivalent services for the train booking (deployed by different companies) and, finally, a flight booking service.*

The authentication service is a common service that allows a user to login. Once done, the user can close his session and go to a final state.

The train and flight booking services are a simplified form of the travel booking on the Web. The user is firstly asked to find among the available travels the one meeting his requirements. These requirements can include the departure city, the arrival city and the date. Once found, the service asks the user to pay for his booking before going to a final state. The final state, in this service, reflects the booking validation.

The repository R can be associated to an FSM that formalizes its "global" behavior. Such a global behavior is the result of combining in all possible ways the available behaviors in R. Formally, it consists in the asynchronous product of all DFSMs in R i.e., $\odot(R) = \langle \Sigma_R, S_R, s_0^R, F_R, \lambda_R \rangle$. Thus, an execution of R is some execution of $\odot(R)$. Informally, an execution of $\odot(R)$ is some evolution of its services i.e., a possible "legal" alternation between transitions belonging to the services of R.

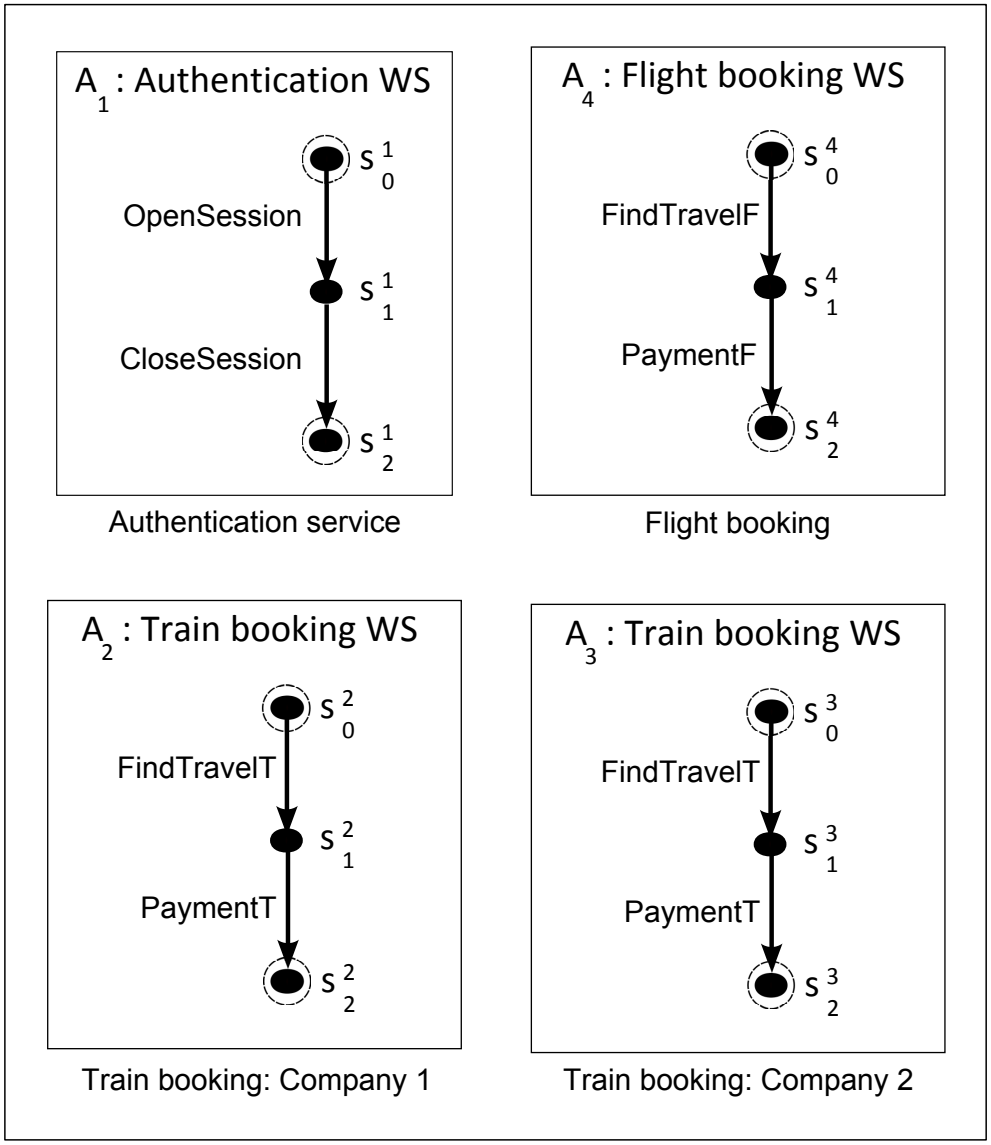

Figure 4.2: A repository of services

Example 3. (Repository execution) *Consider the repository of Figure 4.2. If we use the authentication service and the train booking service (take company 1) to run executions then we can have, for instance:*

1. $\langle s_0^1, s_0^2, s_0^3, s_0^4\rangle \xrightarrow{OpenSession} \langle s_1^1, s_0^2, s_0^3, s_0^4\rangle \xrightarrow{CloseSession} \langle s_2^1, s_0^2, s_0^3, s_0^4\rangle$
2. $\langle s_0^1, s_0^2, s_0^3, s_0^4\rangle \xrightarrow{OpenSession} \langle s_1^1, s_0^2, s_0^3, s_0^4\rangle \xrightarrow{FindTravelT} \langle s_1^1, s_0^2, s_1^3, s_0^4\rangle \xrightarrow{PaymentT} \langle s_1^1, s_0^2, s_2^3, s_0^4\rangle$
3. $\langle s_0^1, s_0^2, s_0^3, s_0^4\rangle \xrightarrow{OpenSession} \langle s_1^1, s_0^2, s_0^3, s_0^4\rangle \xrightarrow{FindTravelT} \langle s_1^1, s_0^2, s_1^3, s_0^4\rangle \xrightarrow{PaymentT} \langle s_1^1, s_0^2, s_2^3, s_0^4\rangle \xrightarrow{CloseSession} \langle s_2^1, s_0^2, s_2^3, s_0^4\rangle$

Assumption 1. *Each service in R has one instance in run.*

4.2 Automatic composition synthesis

When no available behavior meets the client specification, the **automatic composition synthesis** can be used to compose the requested behavior. It consists to **synthesize** a new behavior, using existing behaviors. The role of the synthesized behavior is to **delegate** each requested operation (in some execution level) to an available service that can perform it. From a client's viewpoint, the synthesized behavior will be similar to the firstly requested behavior. The synthesized behavior is called a **Delegator** and the requested behavior is called a **target** service. In the following, we firstly define the target and the Delegator concepts independently in sections 4.2.1 and 4.2.2. Then, we show in Section 4.2.3 that the Delegator can be generated using simulation preorder.

4.2.1 The target service

It is the desired service i.e., it needs to be composed in order to meet client requirements. Its specification is provided by the client as a deterministic FSM, denote by A_T. Example 4 provides a description of a target service.

Example 4. (A Target service)

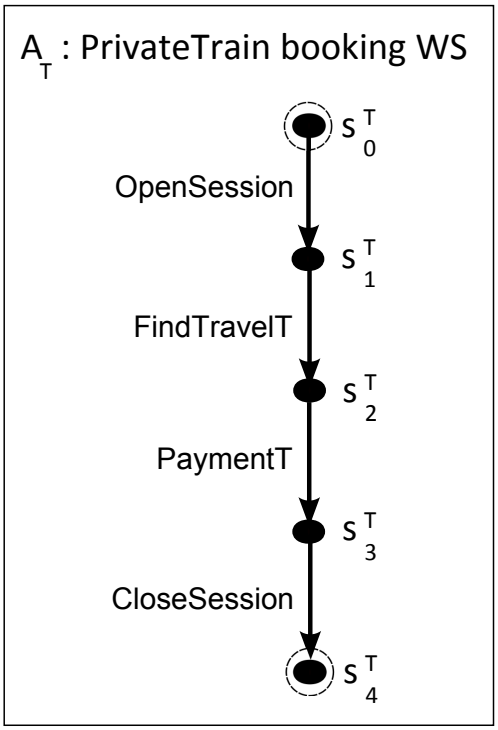

Figure 4.3: A private train booking Web service (the target service)

Let us reconsider the repository of Figure 4.2. Assume we need a private train booking service. The private booking ensures train booking but only to registered customers. For this purpose, the desired service has firstly to go through an authentication phase. Then the booking phase must proceed within a session that opens only for registered users. At the

end, the user has to close his session. The protocol corresponding to such a specification is depicted in Figure 4.3. It is not directly available in the repository of Figure 4.2. Therefore, it becomes a target service that needs to be composed using already available services.

4.2.2 The Delegator

Informally, the Delegator (also called "orchestrator") [8, 27, 41, 46] is a component able to delegate the execution of each requested operation to an available service. It has full observability on available services' states. That is, it can keep track (at runtime) of the current state available services are in. The Delegator coordinates available services executions in order to mimic the target service behavior [46].

A Delegator can be seen as an FSM where the transitions are annotated with suitable **delegations** in order to specify to which component each operation of the target service is delegated (see Example 5). Formally, the Delegator can be defined as follows:

Definition 1. (The Delegator) *Let $R = \{A_1, \ldots, A_k\}$ be a repository with $A_i = \langle \Sigma_i, S_i, s_0^i, F_i, \lambda_i \rangle$, $i \in [1,k]$ and $A_T = \langle \Sigma_T, S_T, s_0^T, F_T, \lambda_T \rangle$ is a target protocol. A Delegator using R for synthesizing A_T is a tuple $D(A_T, R) = \langle \Sigma_D, S_D, s_0^D, F_D, \lambda_D, delegates_D \rangle$, such that:*

- *$\langle \Sigma_D, S_D, s_0^D, F_D, \lambda_D \rangle$ is an FSM and,*
- *$delegates_D : \lambda_D \longrightarrow R$; is a function indicating to which protocol in R each transition in λ_D is delegated.*

Example 5. (Delegator) *Consider the example of Figure 4.4. Delegators in Figures 4.4(3) and 4.4(4) are two possibilities to synthesize the behavior of the Target specified*

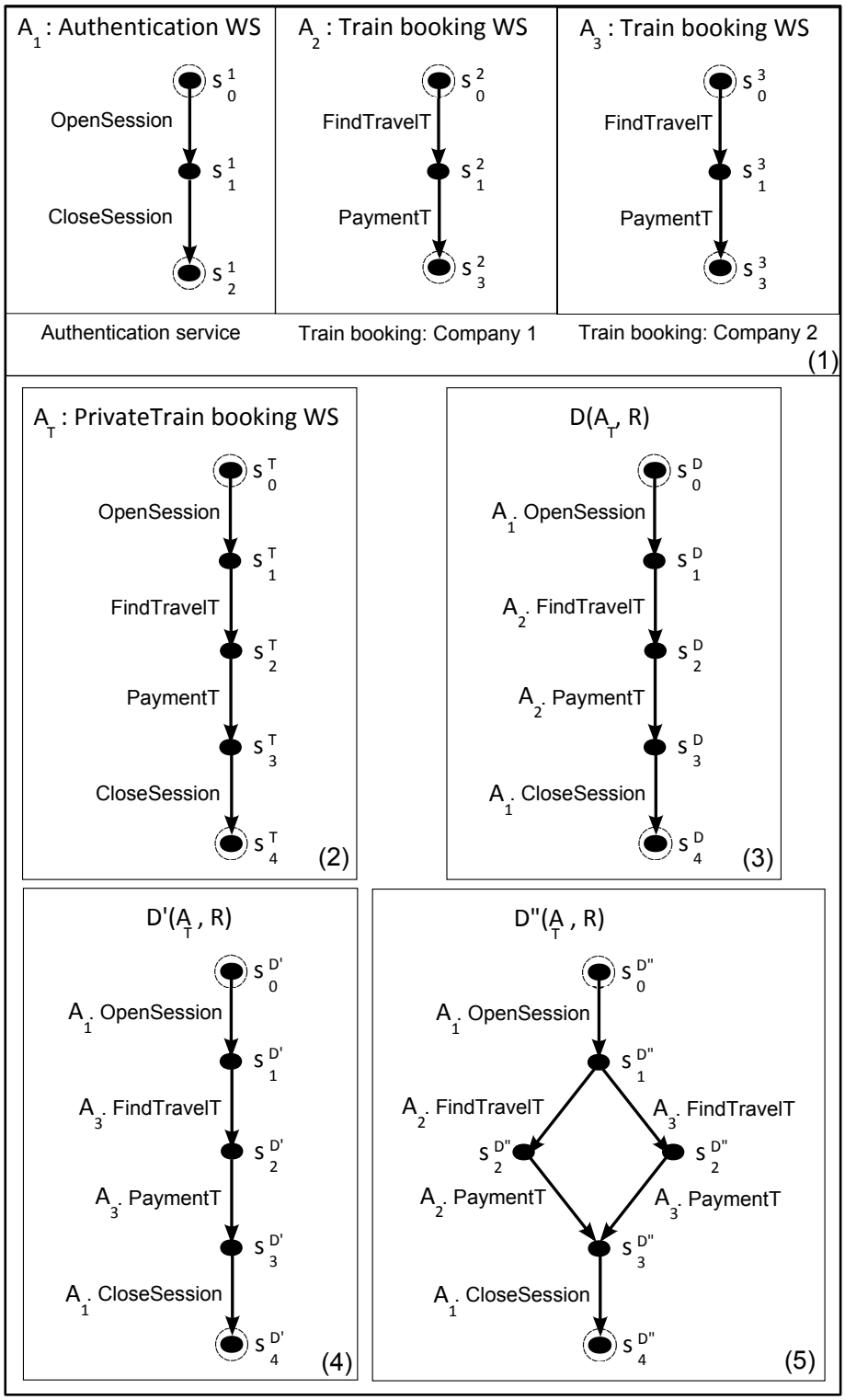

Figure 4.4: Possible Delegators

in Figure 4.4(2), using available services in Figure 4.4(1). The "largest" Delegator that can be obtained is depicted in figure 4.4(5) and it consists in gathering all delegation possibilities in a single FSM.

Definition 2. (Delegator execution) *Let* $R = \{A_1, \ldots, A_k\}$ *be a repository and* $D(A_T, R) = \langle \Sigma_D, S_D, s_0^D, F_D, \lambda_D, delegates_D \rangle$ *is a Delegator using R. An execution of* $D(A_T, R)$ *is a sequence* $\sigma = s_0 \xrightarrow{(a_1, A_{v_1})} s_1 \ldots \xrightarrow{(a_n, A_{v_n})} s_n$ *such that:*

- $s_0 = s_0^D$, *and*
- *for* $i \in [0, n-1]$, *we have* $(s_i, a_{i+1}, s_{i+1}) \in \lambda_D$ *and* $delegates_D((s_i, a_{i+1}, s_{i+1})) = A_{v_{i+1}}$.

Example 6. (Delegator execution) *Continuing with the example of Figure 4.4. One possible execution of the Delegator in Figure 4.4(3) is* $\sigma = s_0^D \xrightarrow{(OpenSession, A_1)} s_1^D \xrightarrow{(FindTravelT, A_2)} s_2^D \xrightarrow{(PaymentT, A_2)} s_3^D$.

An execution of the Delegator corresponds to a sequence of delegations. Each delegation involves the execution of some transition within the corresponding delegated service. We say that the moved service transition is **inherited** from the Delegator transition. Therefore, to each Delegator execution corresponds to a set of inherited services' transitions that we define by the following.

Definition 3. (Inherited transitions) *Let R be a repository,* $D(A_T, R) = \langle \Sigma_D, S_D, s_0^D, F_D, \lambda_D, delegates_D \rangle$ *is a Delegator using R and* $\sigma = s_0 \xrightarrow{(a_1, A_{v_1})} \ldots \xrightarrow{(a_n, A_{v_n})} s_n$ *is a Delegator execution. Let* $\sigma^{\mapsto t_D}$ *denote a prefix of* σ *such that* $last(\sigma^{\mapsto t_D}) = t_D$. *We say that the service transition* $t \in \lambda_i, i \in [1, k]$ *is an inherited transition of* t_D *and we write* $inh(t_D) = t$ *if and only if* $last(\pi_{A_i}(\sigma^{\mapsto t_D})) = t$.

Example 7. (Inherited transitions) *Take the Delegator at Figure 4.4(3). We have:*

- $inh(s_0^D \overset{(OpenSession,A_1)}{\longrightarrow} s_1^D) = (s_0^1, OpenSession, s_1^1)$,
- $inh(s_1^D \overset{(FindTravelT,A_2)}{\longrightarrow} s_2^D) = (s_0^2, FindTravelT, s_1^2)$, *and*
- $inh(s_2^D \overset{(PaymentT,A_2)}{\longrightarrow} s_3^D) = (s_1^2, PaymentT, s_2^2)$.

In this work, the Delegator is one input to our problem which is supposed to be correct. That is, a protocol A is delegated to execute an operation a (requested by the target A_T) only under the following conditions:

- (1) A is in a state allowing it to execute a and,
- (2) If A_T reaches a final state then A has to do same.

It is out of the scope of this thesis to check Delegator correctness. Nevertheless, we provide in the following a small overview of the simulation-based composition process that allows generating correct Delegators.

4.2.3 Delegator generation

The Delegator results from a composition process that uses simulation preorder to check whether the behavior of the Target can be "simulated" by a possible combination of services in R. Such a simulation-based method is called **composition synthesis** [7, 8, 27, 30, 41]. Theorem 1 [8] below is reformulated in our context. It shows that checking for the existence of a service composition i.e., a Delegator, can be reduced to checking whether the target FSM A_T is simulated by $\odot(R)$.

Theorem 1. *[8] Let R be a repository of services and A_T a target service. A Delegator $D(A_T, R)$ exists if and only if $A_T \preceq \odot(R)$.*

Example 8. (Simulation-based composition) *In the example of Figure 4.5, we dropped*

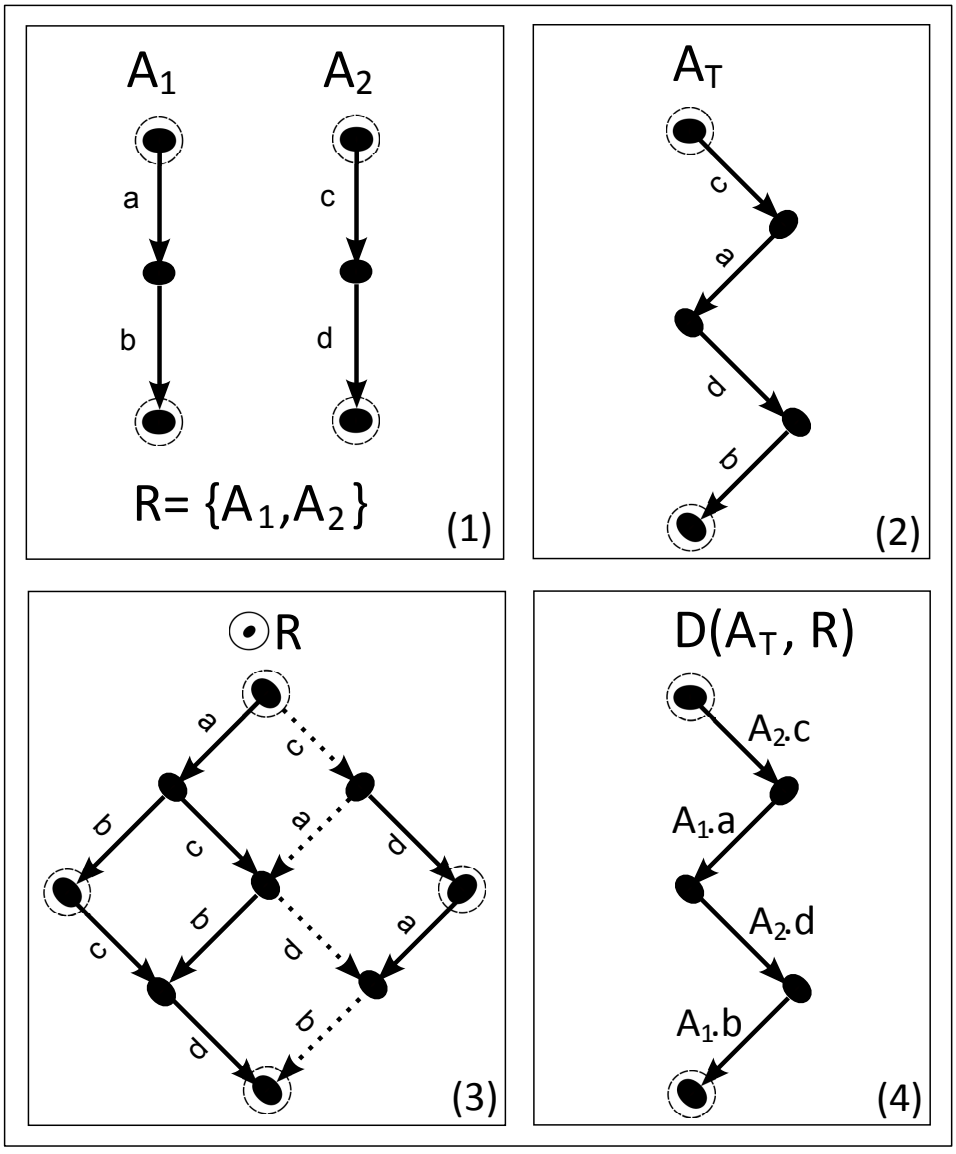

Figure 4.5: Simulation-based composition

states for simplicity reasons. Figure 4.5(1) represents a simple repository that contains two

FSMs. Figure 4.5(2) is the FSM of the target A_T. In order to check whether the behavior of A_T can be simulated by the behaviors in R, $\odot(R)$ is computed and depicted at Figure 4.5(3). It is clear that A_T is simulated by $\odot(R)$ (see dashed arrows within $\odot(R)$). Then a Delegator $D(A_T, R)$ is generated based on the computed simulation and is depicted in Figure 4.5(4).

4.2.4 Discussion

The composition synthesis problem of services that export their protocols has raised a lot of research work such as [6, 7, 41]. In [41], Muscholl and Walukiewicz reduce the protocol synthesis problem to the problem of testing a simulation relation between the target protocol and the product of component protocols. They also show the Exptime completeness of the bounded instances protocol synthesis problem in which each service protocol that can be involved in a composition is bounded by a constant k fixed a priori. This case can be trivially reduced to the case where $k = 1$ for each protocol by duplicating each service k times and allowing each service to run just one instance. The complexity of the unbounded case has been studied in [30]. In [8], it is proven that checking for the maximal (or, the largest) simulation preorder is still Exptime hard.

Following [8, 46, 49], using the maximal simulation has a very interesting property: it contains enough information to allow for extracting every possible composition, through a suitable choice function. This property opens the possibility of devising composition in a ”just-in-time” fashion: the maximal simulation is computed a priori then, equipped with such a simulation, the composition is started, choosing the next step in the composition

according to the criteria that can depend on information that is available only at run-time (actual availability of services). Indeed, it suffices that the next step chosen for execution leads to service states that remain in the simulation relation.

In this work, we assume that our Delegator is correct i.e., generated using simulation-based method. Clearly, if the concerned Delegator was built on the basis of a maximal simulation preorder then its set of possible executions may be richer. Therefore, the recovery process may deal with more candidate recovery executions.

4.3 Relaxing executions with dependencies

In the protocol-based Web service composition model, an execution of the composite corresponds to a totally ordered set (a chain) of transitions. In this section, we relax the execution to a **partially ordered set (poset)**. We essentially exploit information about **data dependencies** over transitions. That is, for a given Delegator execution σ, a data dependency goes from a transition $t_D \in trans(\sigma)$ to a transition $t'_D \in trans(\sigma)$ if, at least, one output of t_D is an input to t'_D. This is translated to $t_D \leq t'_D$ in the order relation over $trans(\sigma)$.

This novel vision to the execution concept will help to characterize the recovery transparency, given a failed and a candidate recovery execution. Actually, the more the failed execution is similar to the candidate recovery execution, the more the recovery is transparent. The similarity between two executions can not be measured solely in terms of shared operations, but also must take into account the direction of data flow between shared operations. This last constraint is captured by the partial order concept since it is constructed

using data dependencies.

A lot of works (e.g., [9]) deal with data dependencies as an available knowledge that can be exploited offline to design a recovery mechanism. A second direction (e.g., [58]) deals with data dependencies as a discovered knowledge that becomes available only at runtime. Without loss of generality, we deal with data dependencies as an available knowledge. Indeed, the unavailability of information about data dependencies can simply be considered as a total order over Delegator executions.

Definition 4. (Data dependencies and relaxed executions) *Let* σ *be a Delegator execution. We define data dependencies on* σ *as a partial order over* $trans(\sigma)$ *denoted* $P(\sigma) = (trans(\sigma), \leq_\sigma)$. $P(\sigma)$ *is a relaxed execution.*

Note that, for an execution σ, we assume that $P(\sigma)$ is transitively reduced. The example in Figure 4.6 illustrates the difference between an execution and a relaxed execution.

We make a hypothesis about intra-service dependencies; we suppose that each two successive transitions within a same service and belonging to a possible Delegator execution are related by a data dependency. Formally, if $P(\sigma) = (trans(\sigma), \leq_\sigma)$ is a relaxed execution then we have $\forall t_D, t'_D \in trans(\sigma)$ such that $inh(t_D) = (s^v_j, a, s^v_{j'})$ and $inh(t'_D) = (s^v_{j'}, a', s^v_{j''}) \Rightarrow t_D \leq_\sigma t'_D$ in $P(\sigma)$, with $v \in [1,k]$. In fact, this assumption tells to respect services protocols in the sense that we cannot decompose the order defined by the designer of some component service.

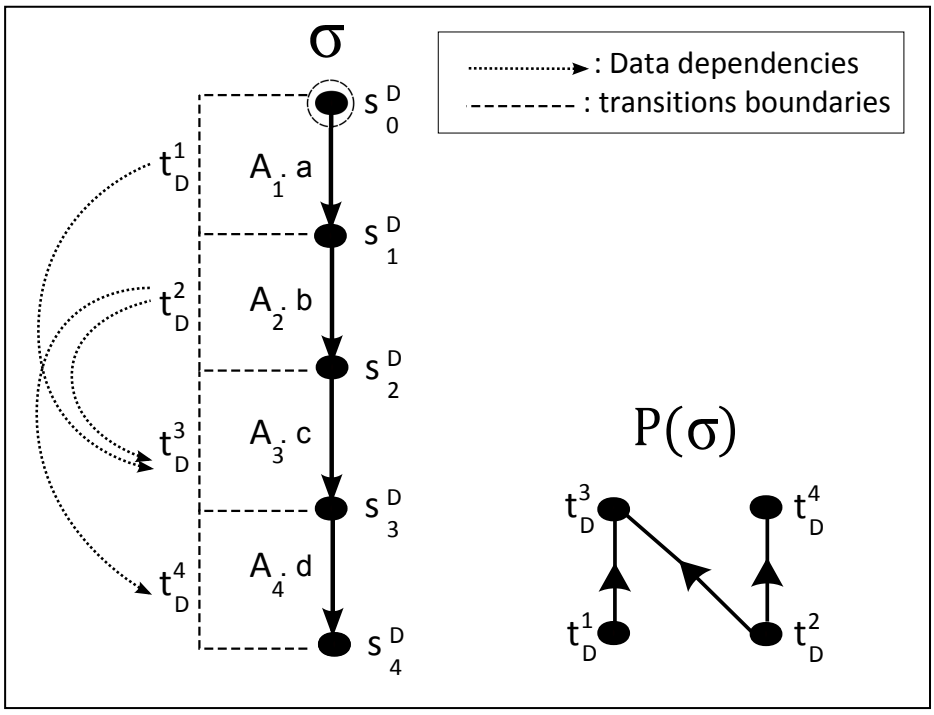

Figure 4.6: Execution vs. relaxed execution

4.4 Summary

This chapter described mainly the composition model adopted in this work. A first part presented an existing protocol-based composition model where services export their conversational behaviors as FSMs. The composition process results in a Delegator that mimics the target service behavior. It delegates its requested operations to component services at runtime. In such a model, an execution is a totally ordered set of transitions. In a second part of this chapter, we relaxed the execution form to a partially ordered set by exploiting information about data dependencies. The order among transitions will serve to characterize the transparency level of the recovery process. This will be detailed in the next chapter.

Chapter 5

Automatic recovery in Web service composition

The unavailability of component services is a common failure that may encounter a Web service composition. Ensuring the availability of components is a hard challenge because of their autonomy and privacy. Therefore, a recovery mechanism is required in order to preserve the composite service consistency. In this work, we consider the unavailability failure in the protocol-based composition model. The unavailability failure results in an incomplete execution of the composite. Thus, the recovery process should transform the failed execution into a recovery execution. The recovery execution is an alternative execution of the composite that still has the ability to reach a final state. Clearly, several candidate recovery executions may be available. Therefore, we present in this work a formal study of the recovery problem where the goal is to find the best recovery execution(s). A best recovery execution must be attainable from the failed execution with a minimal number of visible compensations in the recovery plan.

We provide in this chapter a formal and detailed description to the unavailability failure and the recovery problem. In Section 5.1, our focus is made on the unavailability failure. We deeply describe the event of unavailability and its side-effects on the running composition. In Section 5.2, we turn our attention to the recovery problem. The main concepts related to the recovery are defined, the core theorems and lemmas of this work are announced and the complexity issues are discussed.

5.1 Formalizing unavailability failure in Web service composition

In this section we give a closer look at the unavailability failure in the protocol-based Web service composition. First of all, we intuitively describe the unavailability failure event in Section 5.1.1. Then, we deal with the unavailability failure effects on the Delegator in Section 5.1.2.

5.1.1 Unavailability failure occurrence

In the protocol-based Web service composition, the Delegator makes call to component services to perform requested operations. It may happen that soft or hard problems cause the runtime unavailability of one or more component services. When the Delegator makes call to an unavailable service then an unsuccessful delegation occurs. This results in a failed execution of the Delegator that will need to be repaired. In the sequel, a failed execution is denoted σ^F. To generalize, we assume that unavailability failure occurs as a set

of unavailable transitions over the repository. Each unavailable transition may belong to a different component service. By doing this, we allow all cases of unavailability including the partial unavailability of a single service (only a subset of its transitions becomes unavailable) and the total/partial unavailability of multiple services at the same time. The set of all unavailable transitions within the repository is assumed discovered the moment of the first unsuccessful delegation.

Each unavailable transition on the repository may result in an invalid delegation. This effect is expressed as a set of **invalid transitions** on the automaton corresponding to the Delegator, such as depicted in the example of Figure 5.1. We present in the next section an algorithm that allows cleaning the Delegator automaton by removing the set of invalid transitions.

5.1.2 Delegator cleaning

We deal in this section with the effect of unavailability failure on the Delegator structure. Clearly, the occurrence of an unavailability failure decreases the number of possible executions of the Delegator. Consequently, some branches on the Delegator can no more be useful and should be removed such as depicted in Figure 5.2. A branch is removed if:

- It no longer leads to a final state because of some future invalid transitions. For instance, the branch leading to the state s_5^D on Figure 5.2(1) should be removed.
- It is no longer reachable from the initial state because of some past invalid transitions such as the branch leading to the state s_{10}^D on Figure 5.2(1). In this case, the branch need not be cleaned since it is unreachable.

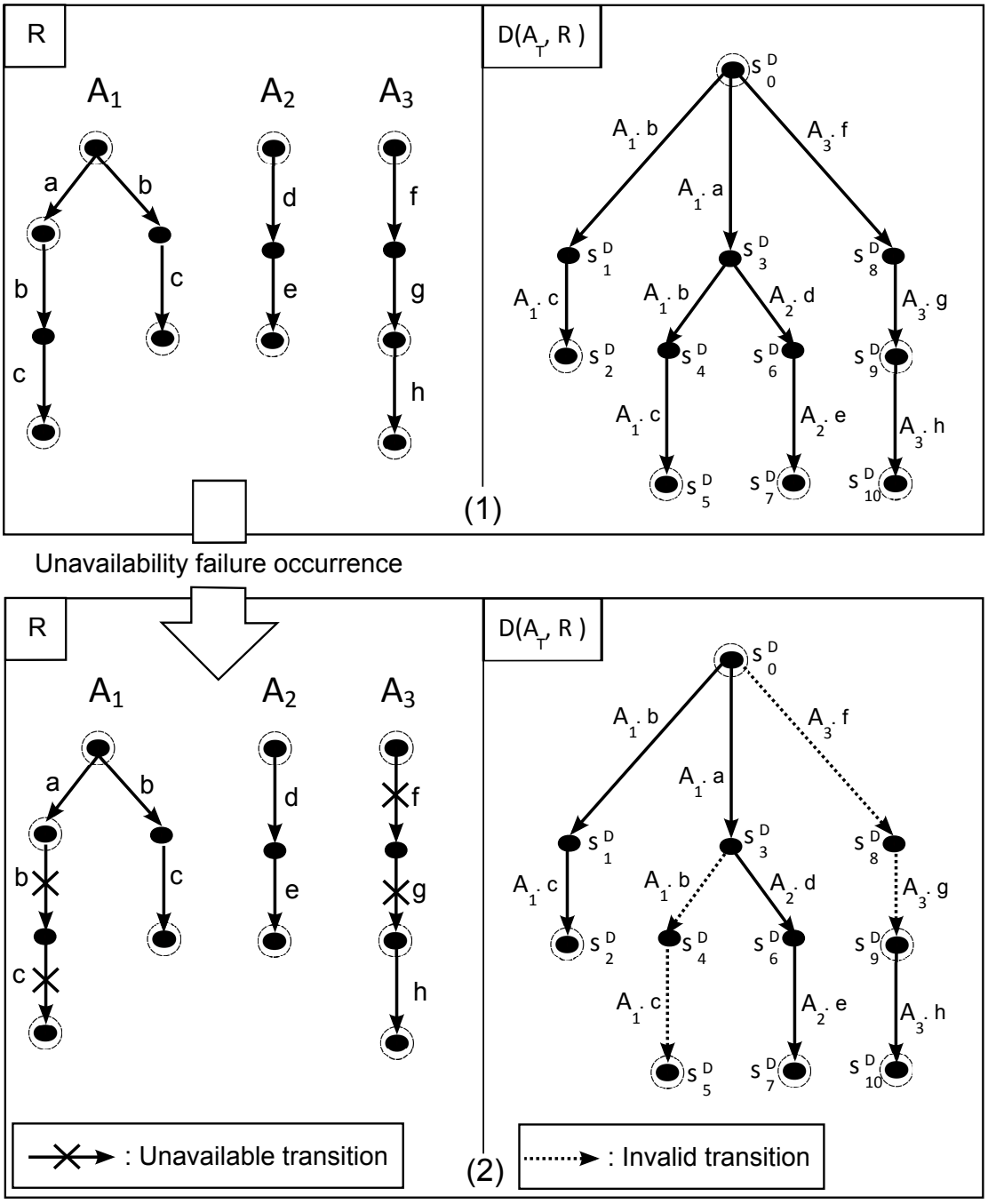

Figure 5.1: Unavailability failure occurrence

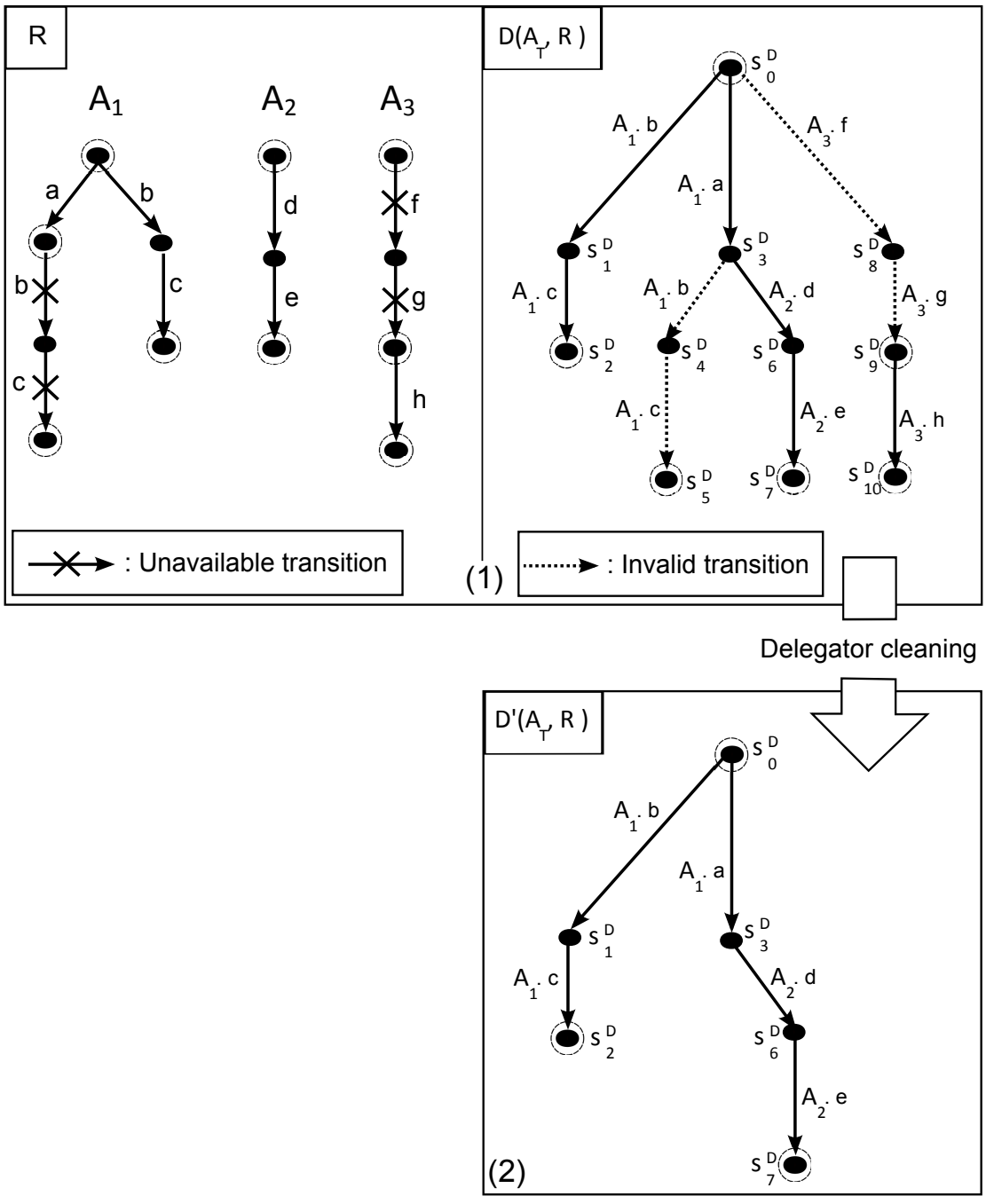

Figure 5.2: A Delegator cleaning

We use the cleaning Algorithm 1, described below, to create a cleaned version of $D(A_T,R) = \langle \Sigma_D, S_D, s_0^D, F_D, \lambda_D, delegates_D \rangle$ in which all non-useful branches are deleted. The resulting Delegator, denoted $D'(A_T,R) = \langle \Sigma_D, S_{D'}, s_0^{D'}, F_{D'}, \lambda_{D'}, delegates_D \rangle$, will be used in the recovery process when looking for alternatives to the failed execution. Indeed, it reduces the search space by putting out risky executions.

Let IT denotes the set of invalid transitions on the Delegator. Then Algorithm 1 essentially goes through two steps:

- **Lines 2→6:** We create an initial cleaned Delegator. It is a copy of the original Delegator except for the set of transitions i.e., $\lambda_{D'} \leftarrow \lambda_D \setminus IT$.
- **Lines 7:** All transitions of $\lambda_{D'}$ that no more can serve to reach a final state are removed using the Procedure Clean. We base on the following property: Each transition which the arrival state has no successors and does not belong to final states should be removed. To deal with loops and cycles, each state s is set to *visited* when calling Clean(s). However, the state s may be successor to other states belonging to branches not yet traversed. Therefore, the state s is reset to *nonvisited* at the end of each call to Clean(s).

In Algorithm 1, $succ(s)$ and $pred(s)$ denote respectively the set of all successors and predecessors of a state $s \in S_D$. $succ_i(s)$ and $pred_i(s)$ are respectively the i^{th} successor and the i^{th} predecessor of s.

Both automata corresponding to the initial and the cleaned Delegators are given by adjacency matrix. The set $F_{D'}$ is a binary table where each state i receives 1 if $i \in F_{D'}$ and

receives 0 otherwise. Therefore, the worst-case computational complexity of the Algorithm 1 is $O(|\lambda_{D'}|)$. This corresponding to the complexity of the depth-first traversing of $D'(A_T, R)$.

Algorithm 1: Delegator Cleaning

input : $D(A_T, R), IT$
output: $D'(A_T, R)$

begin
  $\lambda_{D'} \leftarrow \lambda_D \setminus IT$;
  $S_{D'} \leftarrow S_D$;
  $s_0^{D'} \leftarrow s_0^D$;
  $F_{D'} \leftarrow F_D$;
  // Creating the initial cleaned Delegator
  $D'(A_T, R) \leftarrow \langle \Sigma_D, S_{D'}, s_0^{D'}, F_{D'}, \lambda_{D'}, delegates_D \rangle$;
  // Cleaning branches not leading to final states
  Clean($s_0^{D'}$);
return $D'(A_T, R)$;

The automaton corresponding to the cleaned Delegator $D'(A_T, R)$ may not simulate the target behavior for which the original Delegator $D(A_T, R)$ was generated i.e., $A_T \not\preceq \langle \Sigma_D, S_{D'}, s_0^{D'}, F_{D'}, \lambda_{D'} \rangle$. Nevertheless, still-possible executions on $D'(A_T, R)$ can be exploited for the recovery. That is, a target specification may contain different but close execution paths i.e., they semantically attempt to reach similar goals but with different manners. Such specification is designed in order to offer to the client a range of choices and preferences via possible execution paths. If the running execution cannot be continued then the client is oriented towards another choice.

Procedure Clean(s)

```
begin
    Set s as visited;
    if |succ(s)| ≠ ∅ then
        forall the s' ∈ succ(s) do
            if s' is nonvisited then
                if |succ(s')| = 0 and s' ∉ F_D then
                    λ_D' ← λ_D' \ {(s,a,A,s')};
                else
                    Clean(s');
                    if |succ(s')| = 0 and s' ∉ F_D then
                        λ_D' ← λ_D' \ {(s,a,A,s')}
            else
                if s' ∉ F_D' then
                    λ_D' ← λ_D' \ {(s,a,A,s')}
    Set s as nonvisited;
```

5.2 Formalizing the recovery problem

In section 5.1, our focus was made on the unavailability failure. In this section, we focus on the recovery problem. Firstly, we try to explain intuitively what ”recovery” means before going deep into formal details.

Given a failed execution, a recovery is a process enabling to **transform** (or, to migrate) the failed execution into a second execution so called **a recovery execution**. The recovery execution is selected among available executions of the cleaned Delegator $D'(A_T, R)$ in order to guarantee its ability to reach a final state. Furthermore, it must share, at least, one operation name with the failed execution. This is used to limit the number of candidate recovery executions in case the cleaned Delegator contains loops or cycle.

Transforming a failed execution into a recovery execution is performed using a sequence of **recovery operations** including **execution** and **compensation**. Such a sequence of recovery operations builds **a recovery plan**. Some candidate recovery executions may be better than the others with respect to a given quality criterion. One major goal of this work is to characterize best recovery executions with respect to the number of invisibly-compensated transitions. In fact, if the recovery **replaces** (or, substitutes) compensated transitions then the compensations are invisible from a client's perspective. A transition can replace another one if it is performing the same operation and respecting some order constraints.

The present section formalizes the recovery problem. We firstly characterize candidate recovery executions in Section 5.2.1. We define in Section 5.2.2 both recovery operations and recovery plans. We formalize the ”replacement” concept in Section 5.2.3. Finally, in Section 5.2.4, we properly formalize the recovery problem.

5.2.1 Candidate recovery executions

As already discussed, the recovery consists to find the best recovery execution(s) among those available. They are executions ensuring a minimal number of visible compensations. We know that an execution may be candidate if, at least, it shares its last operation label with the failed execution. However, this is insufficient to limit the number of candidates. Actually, the number of possible executions on the Delegator may be infinite because of the loops and the cycles. Therefore, we limit the number of candidates by traversing loops and cycles only once. In the following, we characterize candidate recovery executions.

Definition 5. (Candidate recovery executions) *Let σ^F be a failed execution and let $D'(A_T,R) = \langle \Sigma_D, S_{D'}, s_0^{D'}, F_{D'}, \lambda_{D'}, delegates_D \rangle$ be a cleaned Delegator. An execution $\sigma \in Path(\langle \Sigma_D, S_{D'}, s_0^{D'}, F_{D'}, \lambda_{D'} \rangle)$ is a candidate recovery execution and denoted σ^R if and only if:*

- $Op(last(\sigma^R)) \in Op(\sigma^F)$, *and*
- *If* $t_D \in trans(\sigma^R)$ *then* $|t_D| = 1$ *(each transition occurs just one time).*

In the case where no candidate recovery execution is available, the recovery will consists simply to compensate all transitions of σ^F.

5.2.2 Recovery operations and recovery plans

In this section, recovery operations and recovery plans are respectively defined. A recovery operation is an elementary step in a recovery process. We use two kinds of recovery operations: the execution moves forward some transition while compensation moves it backward. In the following, we formally define both the execution and the compensation recovery operations.

Definition 6. (Execution recovery operation) *let* $\sigma^F = s_0 \xrightarrow{(a_1,A_{v_1})} \dots \xrightarrow{(a_n,A_{v_n})} s_n$ *be a failed execution. We denote by* $\sigma^F \xrightarrow{\mathsf{exec_D}(\sigma^\mathsf{F},\mathsf{t_D})} \sigma'^F$ *the application of a single execution recovery operation* $\mathsf{exec_D}(\sigma^\mathsf{F},\mathsf{t_D})$ *on* σ^F *which produces an execution* σ'^F *such that: If* $t_D = (s_n, a_{n+1}, A_{v_{n+1}}, s_{n+1})$, $v_{n+1} \in [1,k]$ *then* $\sigma'^F = s_0 \xrightarrow{(a_1,A_{v_1})} \dots \xrightarrow{(a_n,A_{v_n})} s_n \xrightarrow{(a_{n+1},A_{v_{n+1}})} s_{n+1}$.

Definition 7. (Compensation recovery operation) *let* σ^F *be a failed execution and* $P(\sigma^F) = (trans(\sigma^F), \leq_{\sigma^F})$ *its associated poset. We denote by* $\sigma^F \xrightarrow{\mathsf{comp_D}(\sigma^\mathsf{F},\mathsf{t_D})} \sigma'^F$ *the application of a single compensation recovery operation* $\mathsf{comp_D}(\sigma^\mathsf{F},\mathsf{t_D})$ *on* σ^F *which produces an execution* σ'^F *such that* $trans(\sigma'^F) \in \mathfrak{I}(P(\sigma^F))$.

Example 9. (Recovery operations) *Figure 5.3 depicts some possible recovery operations that can immediately be applied on the execution* σ. *We can either execute some transition that we called* t_D^3 *(other executions may be possible depending on the Delegator), or compensate* t_D^2. *The transition* t_D^1 *cannot be compensated at this stage because of its data dependency going to* t_D^2. *Therefore,* t_D^2 *should be firstly compensated.*

A recovery plan is a sequence of recovery operations transforming a failed execution σ^F into a candidate recovery execution.

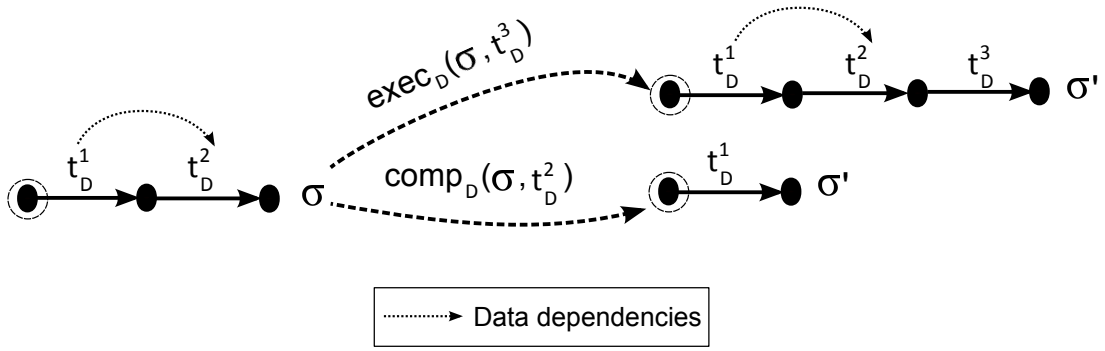

Figure 5.3: Applied recovery operations

Definition 8. (Recovery plan) *Let σ^F and σ^R be respectively a failed and a candidate recovery executions. A recovery plan is a sequence of recovery operations $(ro_1, \ldots, ro_{n-1})$ such that $\sigma^F \overset{ro_1}{\mapsto} \sigma_2 \overset{ro_2}{\mapsto} \ldots \sigma_{n-1} \overset{ro_{n-1}}{\mapsto} \sigma_R$. The recovery plan transforming σ^F to σ^R is denoted $Plan(\sigma^F, \sigma^R)$.*

Example 10. (Recovery plans) *Let σ^F be a failed execution. Figures 5.4(1) and 5.4(2) are respectively recovery executions σ_1^R and σ_2^R obtained from σ^F by applying the following recovery plans:*

- $Plan(\sigma^F, \sigma_1^R) = \langle \mathsf{comp_D}(\sigma^\mathsf{F}, \mathsf{t}_\mathsf{D}^2), \mathsf{comp_D}(\sigma^\mathsf{F}, \mathsf{t}_\mathsf{D}^1), \mathsf{exec_D}(\sigma^\mathsf{F}, \mathsf{t}_\mathsf{D}^4), \mathsf{exec_D}(\sigma^\mathsf{F}, \mathsf{t}_\mathsf{D}^5) \rangle$
- $Plan(\sigma^F, \sigma_2^R) = \langle \mathsf{comp_D}(\sigma^\mathsf{F}, \mathsf{t}_\mathsf{D}^2), \mathsf{comp_D}(\sigma^\mathsf{F}, \mathsf{t}_\mathsf{D}^3), \mathsf{exec_D}(\sigma^\mathsf{F}, \mathsf{t}_\mathsf{D}^5), \mathsf{exec_D}(\sigma^\mathsf{F}, \mathsf{t}_\mathsf{D}^6) \rangle$

Generating recovery plans Given a failed and a candidate recovery execution, Algorithm 2 generates the recovery plan by simply compensating all transitions of the failed execution in the reverse execution sense, then executing all transitions of the recovery execution. In Algorithm 2, $\sigma^X(i)$, with $X \in \{F, R\}$, denotes the i^{th} transition of the execution σ^X. If executions are given by lists of transitions then Algorithm 2 runs in linear time equal to $max(|\sigma^F|, |\sigma^R|)$.

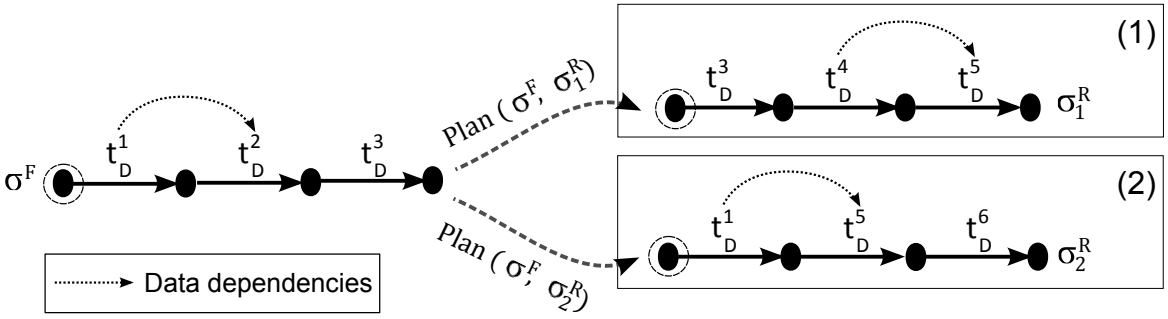

Figure 5.4: Obtained recovery executions

Algorithm 2: Recovery Plan Generation

input : σ^F, σ^R

output: $Plan(\sigma^F, \sigma^R)$ // $Plan(\sigma^F, \sigma^R)$ is a queue

1 **begin**

2 $Plan(\sigma^F, \sigma^R) \leftarrow \emptyset$;

3 **for** $i \leftarrow |\sigma^F|, 1$ **do**

4 $enqueue(Plan(\sigma^F, \sigma^R), \mathsf{comp_D}(\sigma^F, \sigma^F(i)))$;

5 **for** $i \leftarrow 1, |\sigma^R|$ **do**

6 $enqueue(Plan(\sigma^F, \sigma^R), \mathsf{exec_D}(\sigma^F, \sigma^R(i)))$;

5.2.3 The replacement problem

During a recovery procedure, several candidate recovery executions may exist, but some ones may be better than the others with respect to a given quality criterion. In this work, we attempt to minimize the number of visible compensations i.e., the best recovery execution is that reached with a minimal number of visible compensations. A compensation is invisible if the compensated transition is replaced by another transition performing the same operation and respecting some order constraints. The **replacement** concept is defined in the following and illustrated in Example 11.

Definition 9. (The Replacement) *Let* $P(\sigma^F) = (trans(\sigma^F), \leq_{\sigma^F})$ *and* $P(\sigma^R) = (trans(\sigma^R), \leq_{\sigma^R})$ *be respectively the posets associated to a failed execution* σ^F *and a recovery execution* σ^R*, and let* $l : \lambda_D \rightarrow \Sigma_D$ *be a labeling function. A replacement of* $P(\sigma^F)$ *by* $P(\sigma^R)$ *is an injective partial function* $\varphi : trans(\sigma^F) \rightarrow trans(\sigma^R)$ *such that:*

1. $Dom(\varphi) \in \Im(P(\sigma^F))$ *with* $Dom(\varphi) = \{x$ *such that* $x \in trans(\sigma^F)$ *is defined* $\}$,
2. $Img(\varphi) \in \Im(P(\sigma^R))$ *with* $Img(\varphi) = \{\varphi(x)$ *such that* $x \in trans(\sigma^F)$ *is defined* $\}$,
3. $\forall x \in Dom(\varphi), l(x) = l(\varphi(x))$,
4. $(Dom(\varphi), \leq_{\sigma^F})$ *and* $(Img(\varphi), \leq_{\sigma^R})$ *are isomorphic via the injection* φ *i.e.,* $\forall x, y \in Dom(\varphi), x \leq_{\sigma^F} y \Leftrightarrow \varphi(x) \leq_{\sigma^R} \varphi(y)$.

If condition (4) is not required to be true then the replacement is said to be ***loose*** *and is denoted* φ^{loose}*, else it is* ***strict*** *and denoted* φ^{strict}*. The set* $Dom(\varphi)$ *is the set of* ***replaced transitions***.

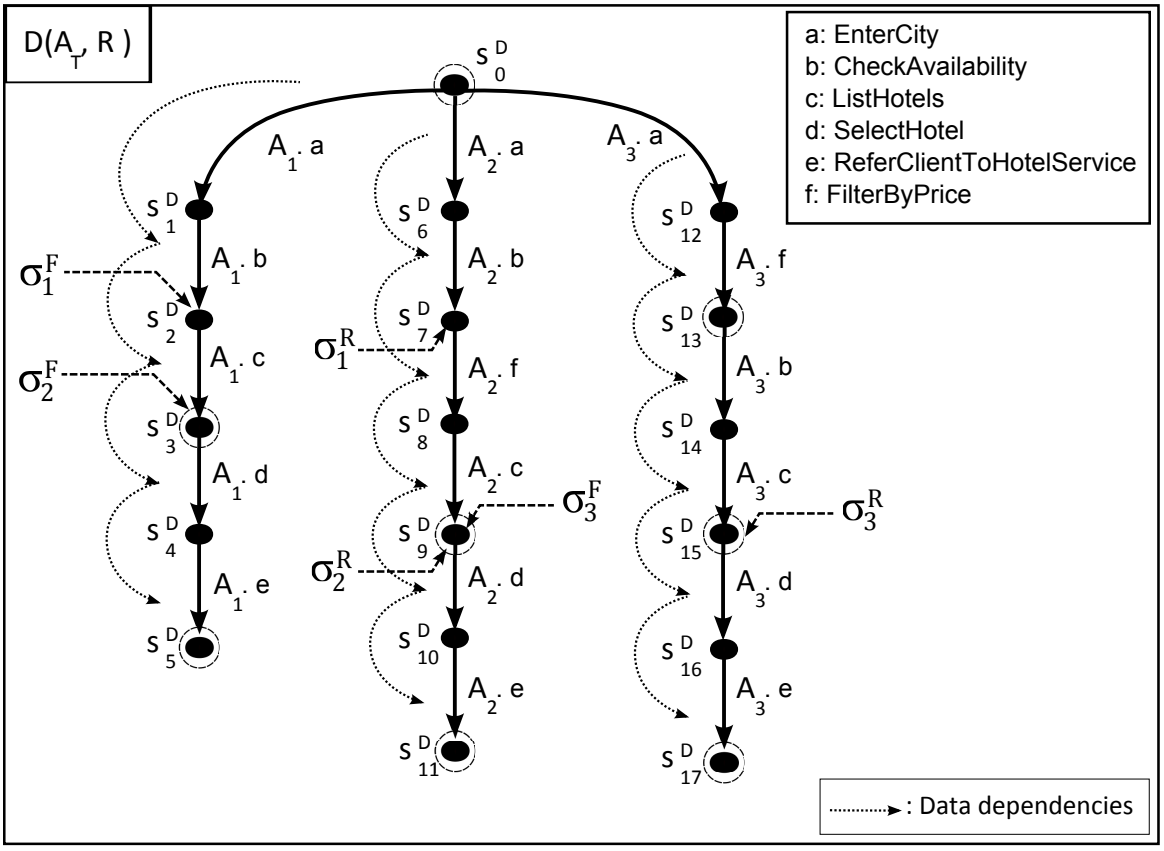

Figure 5.5: A Search-Hotel composite service

Example 11. (Replacement and invisible compensation)

Consider The Delegator in Figure 5.5. This Delegator gathers three services (A_1, A_2 and A_3) where each of them is used to look for an available hotel room in some city entered as parameter. Six operations may be executed in the whole Delegator:

- `EnterCity`*: Allows the client to enter the city name where he seeks a hotel.*
- `CheckAvailability`*: The system searches for hotels situated on the specified city and that have available rooms.*
- `ListHotels`*: A list of hotels is displayed.*
- `SelectHotel`*: Allows the client to choose among proposed hotels.*
- `ReferClientToHotelService`*: The client is oriented towards the reservation Web service associated with the selected hotel.*
- `FilterByPrice`*: A functionality that sorts available hotels from the least to the most expensive.*

Building on what precedes, we can describe the global functionality of A_1, A_2 and A_3 as follows:

- ***The service** A_1**:** Simply takes the city name as an input and displays the list of all city hotels having available rooms.*
- ***The service** A_2**:** Also takes the city name as parameter and looks for hotels having available rooms. Once hotels are found, they are displayed from the least to the most expensive thanks to the additional operation* `FilterByPrice`*.*

- ***The service*** A_3***:*** *The same functionality as* A_2*. However, hotels are firstly filtered by price then only those having available rooms are displayed in the list.*

Let us consider the following recovery scenarios:

1. $Plan(\sigma_1^F, \sigma_1^R) = \langle \mathsf{comp_D}(\sigma_1^\mathsf{F}, (s_1^\mathsf{D}, A_1, b, s_2^\mathsf{D})), \mathsf{comp_D}(\sigma_1^\mathsf{F}, (s_0^\mathsf{D}, A_1, a, s_1^\mathsf{D})),$
$\mathsf{exec_D}(\sigma_1^\mathsf{F}, (s_0^\mathsf{D}, A_2, a, s_6^\mathsf{D})), \mathsf{exec_D}(\sigma_1^\mathsf{F}, (s_6^\mathsf{D}, A_2, b, s_7^\mathsf{D}))\rangle$.

   Before failure occurrence, the client has executed a then b with a data dependency from a to b. After the recovery procedure, the client is in a state (s_7^D) *that is reached also after an execution of a followed by b and a data dependency from a to b. Therefore, from a client's perspective, nothing is changed. The recovery is completely transparent and the compensation of both* (s_1^D,A_1,b,s_2^D) *and* (s_0^D,A_1,a,s_1^D) *is invisible.*

2. $Plan(\sigma_2^F, \sigma_2^R) = \langle \mathsf{comp_D}(\sigma_1^\mathsf{F}, (s_2^\mathsf{D}, A_1, c, s_3^\mathsf{D})), \mathsf{comp_D}(\sigma_1^\mathsf{F}, (s_1^\mathsf{D}, A_1, b, s_2^\mathsf{D})),$
$\mathsf{comp_D}(\sigma_1^\mathsf{F}, (s_0^\mathsf{D}, A_1, a, s_1^\mathsf{D})), \mathsf{exec_D}(\sigma_1^\mathsf{F}, (s_0^\mathsf{D}, A_2, a, s_6^\mathsf{D})), \mathsf{exec_D}(\sigma_1^\mathsf{F}, (s_6^\mathsf{D}, A_2, b, s_7^\mathsf{D})),$
$\mathsf{exec_D}(\sigma_1^\mathsf{F}, (s_7^\mathsf{D}, A_2, f, s_8^\mathsf{D})), \mathsf{exec_D}(\sigma_1^\mathsf{F}, (s_8^\mathsf{D}, A_2, c, s_9^\mathsf{D}))\rangle$

   Being on the state s_3^D*, the client has as a result of execution a list of all hotels having available rooms. Following recovery, the client will have on the state* s_9^D *a list of hotels sorted by price. Then a small difference between results will be visible to the client which makes the recovery not completely transparent.*

3. $Plan(\sigma_3^F, \sigma_3^R) = \langle \mathsf{comp_D}(\sigma_1^\mathsf{F}, (s_9^\mathsf{D}, A_2, c, s_8^\mathsf{D})), \mathsf{comp_D}(\sigma_1^\mathsf{F}, (s_8^\mathsf{D}, A_2, f, s_7^\mathsf{D})),$
$\mathsf{comp_D}(\sigma_1^\mathsf{F}, (s_7^\mathsf{D}, A_2, b, s_6^\mathsf{D})), \mathsf{comp_D}(\sigma_1^\mathsf{F}, (s_6^\mathsf{D}, A_2, a, s_0^\mathsf{D})), \mathsf{exec_D}(\sigma_1^\mathsf{F}, (s_0^\mathsf{D}, A_3, a, s_{12}^\mathsf{D})),$
$\mathsf{exec_D}(\sigma_1^\mathsf{F}, (s_{12}^\mathsf{D}, A_3, f, s_{13}^\mathsf{D})), \mathsf{exec_D}(\sigma_1^\mathsf{F}, (s_{13}^\mathsf{D}, A_3, b, s_{14}^\mathsf{D})), \mathsf{exec_D}(\sigma_1^\mathsf{F}, (s_{14}^\mathsf{D}, A_3, c, s_{15}^\mathsf{D}))\rangle$

In this case, the client on both s_9^D and s_{15}^D has as an execution result a list, filtered by price, of hotels having available rooms. Despite the complete transparency of such a recovery, the data on the failed execution σ_3^F did not circulate in the same way as in σ_3^R (from b to f in σ_3^F and from f to b in σ_3^R). Such a recovery is then ***loose*** *but in which all made compensations are invisible to the client.*

Computing the set of replaced transitions ensured by a given candidate recovery execution is a hard task. Indeed, to each transition in the failed execution may correspond multiple substitutes. Choosing one among substitutes may change the total number of replaced transitions (see Example 12 below). Thus, substitutes must be selected in a manner to maximize the total number of replaced transitions. Given a failed execution σ^F, the **Replacement problem** is informally the problem of finding the maximal replacement ensured by some candidate recovery execution σ^R. The maximal replacement correctly reflects the number of invisibly compensated transitions. This will later enable to compare available candidate recovery executions and select the one ensuring a maximal number of replaced transitions. The replacement problem in both cases (strict and loose) is formalized in Definition 10 below.

Definition 10. (Strict and Loose Replacement problems (Optimization problems))

- ***Instance:*** *$P(\sigma^F) = (trans(\sigma^F), \leq_{\sigma^F})$ and $P(\sigma^R) = (trans(\sigma^R), \leq_{\sigma^R})$ are two posets corresponding to the executions σ^F and σ^R.*

- ***Question (for the Strict Replacement problem):*** *Find a strict replacement φ^{strict} : $trans(\sigma^F) \rightarrow trans(\sigma^R)$ such that $|Dom(\varphi^{strict})|$ is maximal.*

- ***Question (for the Loose Replacement problem):*** *Find a loose replacement φ^{loose} :*

$trans(\sigma^F) \rightarrow trans(\sigma^R)$ *such that* $|Dom(\varphi^{loose})|$ *is maximal.*

Example 12. (The replacement problem hardness) *Take the example of Figure 5.6 where a failed and a candidate recovery executions are depicted. Note that in Figures 5.7 and 5.8, the notation* $t^i_D : a$*, with* $a \in \Sigma$*, means that the transition* t^i_D *has the label a.*

If we consider the strict replacement problem then two possible strict replacements are found. The first is denoted φ_1^{strict} *and depicted in Figure 5.7 where* $|Dom(\varphi_1^{strict})| = 3$*. The second possible replacement is denoted* φ_2^{strict} *and is depicted in Figure 5.8 where* $|Dom(\varphi_2^{strict})| = 2$*. Therefore, it is clear that several replacements may exist for each couple of executions which makes the best replacement difficult to compute.*

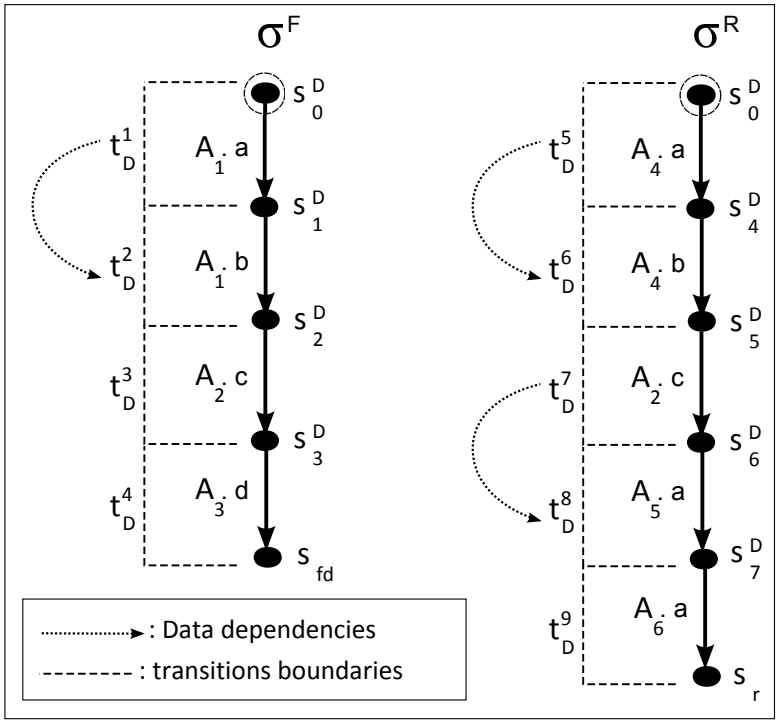

Figure 5.6: A failed and a candidate recovery executions

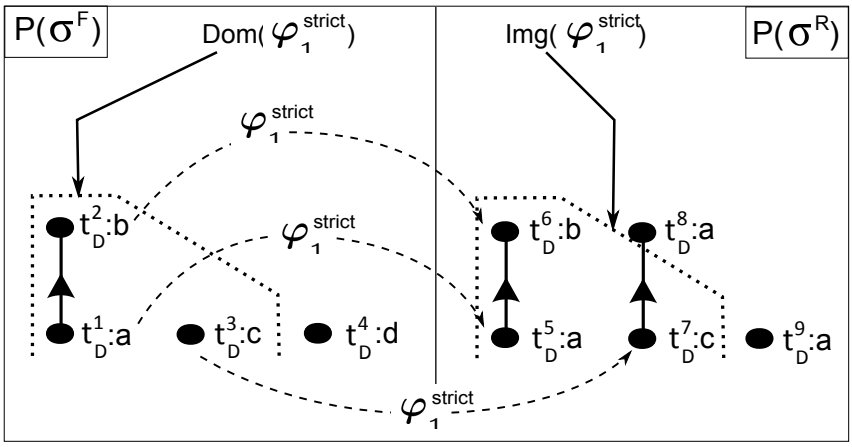

Figure 5.7: A first possible strict replacement

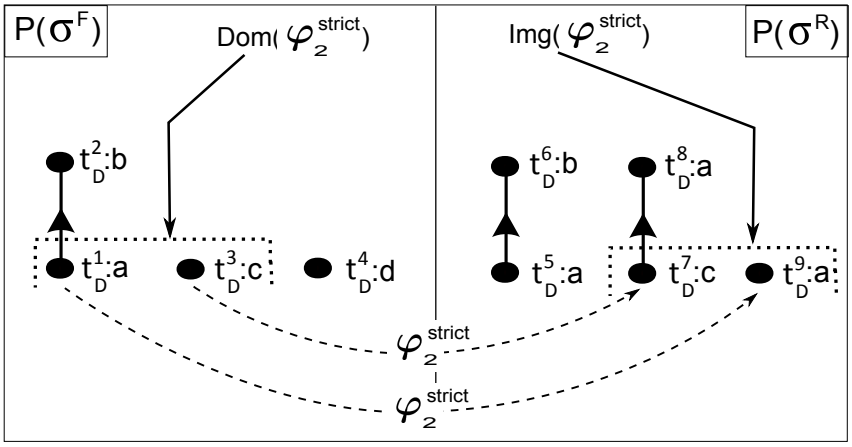

Figure 5.8: A second possible strict replacement

We announce in the following the decision problems associated with both strict and loose replacements and we show the NP-Completeness of each of them.

Definition 11. (The decision problem of the Strict-Replacement (Rep^{strict}))

- ***Instance:*** *$P(\sigma^F)$ and $P(\sigma^R)$ are two posets corresponding to the executions σ^F and σ^R, and k is a positive integer.*
- ***Question:*** *Is there a strict replacement $\varphi^{strict} : trans(\sigma^F) \rightarrow trans(\sigma^R)$ such that $|Dom(\varphi^{strict})| \geq k$?*

Theorem 2. *$Rep^{strict}(P(\sigma^F), P(\sigma^R), k)$ is NP-Complete.*

Proof. The Rep^{strict} problem is proven NP-Complete by reducing from the Partial Subgraph Isomorphism problem (PSI). Details of the reduction are provided in the Appendix A. □

Definition 12. (The decision problem of the Loose-Replacement (Rep^{loose})

- ***Instance:*** *$P(\sigma^F)$ and $P(\sigma^R)$ are two posets corresponding to the executions σ^F and σ^R, and k a positive integer.*
- ***Question:*** *Is there a loose replacement $\varphi^{loose} : trans(\sigma^F) \rightarrow trans(\sigma^R)$ such that $|Dom(\varphi^{loose})| \geq k$?*

Theorem 3. *$Rep^{loose}(P(\sigma^F), P(\sigma^R), k)$ is NP-Complete.*

Proof. Details of the reduction are provided in the Appendix B □

5.2.4 The recovery problem

As already mentioned, the recovery consists to find the best recovery execution. It is the execution ensuring a maximal replacement. The decision problem associated to the recovery can be announced as follows:

Definition 13. (Strict and Loose Recovery problems (Rec^{strict} and Rec^{loose}))

- ***Instance:*** *$P(\sigma^F)$ is a poset corresponding to failed execution, $\{P(\sigma_1^R),\ldots,P(\sigma_n^R)\}$ is a set of posets corresponding to candidate recovery executions and k is a positive integer.*
- ***Question for*** *Rec^{strict}**:** Is there a candidate recovery execution σ_i^R such that the answer to the decision problem $Rep^{strict}(P(\sigma^F),P(\sigma_i^R),k)$ is YES?*
- ***Question for*** *Rec^{loose}**:** Is there a candidate recovery execution σ_i^R such that the answer to the decision problem $Rep^{loose}(P(\sigma^F),P(\sigma_i^R)),k)$ is YES?*

Lemma 1. *Both Strict and Loose Recovery problems belong to Σ_2^P.*

Proof. To see that Rec^{strict} is in Σ_2^P, notice that the problem of verifying whether $Rep^{strict}($ $P(\sigma^F),P(\sigma^R),k)$ is a true instance for some given execution σ^R is in NP (Theorem 2). Then, a set of candidate recovery executions is a certificate that can be verified in polynomial time if an NP-oracle is available. Thus clearly Rec^{strict} is in $NP^{NP} = \Sigma_2^P$. In the same way, Rec^{loose} can be easily shown to be in Σ_2^P using Theorem 3.

□

5.3 Summary and discussion

In a first part of this chapter, we formalized the unavailability failure and its side effects on a running composition. Further than the resulting failed execution, the number of possible executions of the Delegator may decrease. For this purpose, we proposed to clean the Delegator by removing branches not leading to final states. This allows putting out risky alternatives during recovery.

In a second part, the recovery problem is formalized. We defined the recovery as the process of transforming the failed execution into a recovery execution using a recovery plan. The recovery execution is an alternative execution of the cleaned Delegator that shares, at least, one operation with the failed execution.

We defined the recovery problem as the problem of finding the best recovery execution(s) among those available. The best recovery execution should be attainable from the failed execution with a minimal number of visible compensations. A compensation is invisible if the compensated transition is replaced by another transition performing the same operation. We distinguished two kinds of replacement. If the order among the set of replaced transitions is required to be isomorphic to the order among the set of substitute transitions then the replacement is strict. Otherwise, the replacement is loose. In both cases, we proved that the decision problem associated to computing the number of invisibly-compensated transitions is NP-complete (strict and loose replacement problems). Thus, we concluded that deciding of the best recovery execution is in Σ_2^P.

Chapter 6

Conclusion

Through this work, we provided a formal study of the recovery problem in the protocol-based Web service composition. Such a problem may be caused by the runtime unavailability of component services. We focused on the recovery that migrates the failed execution into a recovery execution with a minimal number of visible compensations. A transition is invisibly compensated if it is replaced by another transition performing the same operation and respecting some given order constraints. We mainly exploited the partial orders formalism in characterizing the best recovery executions. Our work can be summarized in the following points:

- We used the protocol-based Web service composition model where both the composite and the components export their conversational behaviors as FSMs. The composition consists in delegating each requested operation (at runtime) to an available component service. We enhanced the composite structure with a set of data dependencies over transitions. Thus, an execution takes the form of a poset.

- We formalized the unavailability failure as a set of unavailable (non-executable) tran-

sitions on the Delegator. By doing this, we capture all cases of unavailability. Those cases include the partial unavailability of one component and the partial/total unavailability of multiple components at the same time.

- The unavailability failure may lead to an unsuccessful delegation which results in a failed execution. We defined the recovery as the process of transforming the failed execution into a second still possible execution on the Delegator using a recovery plan. The recovery problem is then defined as the problem of finding the best recovery execution among those available. It is the one to which the transformation is made with a minimal number of visible compensations towards the client.

- The compensation is invisible whether the compensated transition is replaced by a second transition performing the same operation. Furthermore, if the replacement is required to be strict then the order among the set of replaced transitions is required to be isomorphic to the order among the set of substitute transitions. Otherwise, the replacement is loose.

- We defined the replacement problem as the problem of computing the number of replaced transitions ensured by some given candidate recovery execution. This allows comparing candidate recovery executions and selecting the one ensuring a maximal number of replaced transitions. Indeed, the number of replaced transitions allows computing the number of visible compensations. To this fact, solving the recovery problem requires solving the replacement problem in a first place.

- Complexity results related to the automatic recovery issue are depicted. We have shown the NP-Completeness of the replacement problems (loose and strict). We

have then concluded that the recovery problem in the protocol-based Web service composition is in Σ_2^P.

In the following, we briefly detail some improvement opportunities regarding our research work.

Towards a richer composition model In a composition scenario, an execution is fired and forwarded by a client. To this fact, the client is a very important factor to take into account during recovery. This can be done if the composition model allows to reason about user preferences. For instance, a user may prefer some service attributes over the others such as time, cost, quality of results, etc. Therefore, it may prefer an alternative over the others. Beyond user preferences, it will be interesting if the compensatability constraint is taken into account. That is, if some failed transitions are non-compensable (pivots), then they must be conserved by the recovery procedure.

Providing heuristics The replacement problems in both cases (strict and loose) have been proven NP-Complete (Theorem 2 and Theorem 3). In the present work, no heuristics are provided to approximately solve the replacement problems. To this fact, a good continuation of this work will be to think about those heuristics. In fact, a poset is some kind of directed graph. Thus, we can draw on heuristics developed to solve the graph isomorphism problems.

Bibliography

[1] G. Alonso, F. C. an, H. A. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, Berlin, 2004.

[2] R. Angarita, Y. Cardinale, and M. Rukoz. Faceta: Backward and forward recovery for execution of transactional composite ws. In *CEUR Workshop Proceedings*, Heraklion, Grèce, 2012.

[3] A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability. Technical Report 1145, LAAS-CNRS, April 2001.

[4] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004.

[5] B. Benatallah, F. Casati, and F. Toumani. Representing, analysing and managing web service protocols. *Data Knowl. Eng.*, 58(3):327–357, September 2006.

[6] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *ICSOC*, pages 43–58, 2003.

[7] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic service composition based on behavioral descriptions. *International Journal of Cooperative Information Systems*, 14(4):333–376, 2005.

[8] D. Berardi, F. Cheikh, G. D. Giacomo, and F. Patrizi. Automatic service composition via simulation. *International Journal of Foundations of Computer Science*, 19(2):429–451, 2008.

[9] S. Bhiri, O. Perrin, and C. Godart. Ensuring required failure atomicity of composite web services. In *14th international conference on World Wide Web*, pages 138–14, Chiba, Japan, May 2005.

[10] M. Brambilla, S. Ceri, S. Comai, and C. Tziviskou. Exception handling in workflow driven web applications. In *14th International Conference on World Wide Web*, pages 170–179, Chiba, Japan, 2005. ACM Press.

[11] A. Bucchiarone and S. Gnesi. A survey on services composition languages and models. In *International Workshop on Web Services Modeling and Testing*, pages 37–49, 2006.

[12] O. Bushehrian, S. Zare, and N. Rad. A workflow-based failure recovery in web services composition. *Journal of Software Engineering and Applications*, 5:89–95, 2012.

[13] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Towards a formal framework for choreography. In *WETICE*, pages 107–112, 2005.

[14] Y. Cardinale, M. Rukoz, M. Manouvrier, and J. El Haddad. Cpn-tws: A coloured petri-net approach for transactional-qos driven web service composition. *International Journal of Web and Grid Services (IJWGS)*, 7, 2011.

[15] F. Casati and G. Cugola. Error handling in process support systems. In *Advances in Exception Handling Techniques*, volume 2022 of *Lecture Notes in Computer Science*, pages 251–270. Springer Berlin / Heidelberg, 2001.

[16] K. M. Chan, J. Bishop, J. Steyn, L. Baresi, and S. Guinea. A fault taxonomy for web service composition. In *ICSOC Workshops*, pages 363–375, 2007.

[17] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order (2. ed.)*. Cambridge University Press, 2002.

[18] G. Debanjan, S. Raj, R. R. H., and U. Shambhu. Self-healing systems - survey and synthesis. *Decision Support Systems*, 42(4):2164–2185, January 2007.

[19] J. Dietmar and G. Alexander. The evolution of conceptual modeling. chapter Exception handling in web service processes, pages 225–253. Springer-Verlag, Berlin, Heidelberg, 2011.

[20] L. A. Digiampietri, J. J. Prez-Alczar, and C. B. Medeiros. Ai planning in web services composition: a review of current approaches and a new solution. *SBC*, pages 983–992, 2007.

[21] S. Dustdar and W. Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, Augest 2005.

[22] M. Fredj. *Reconfiguration dynamique des architectures orientées services*. PhD thesis, Université de Paris VI, 2010.

[23] M. Fredj, N. Georgantas, V. Issarny, and A. Zarras. Dynamic service substitution in service-oriented architectures. In *SERVICES I*, pages 101–104, 2008.

[24] G. Friedrich, M. Fugini, E. Mussi, B. Pernici, and G. Tagni. Exception handling for repair in service-based processes. *IEEE Transactions on Software Engineering*, 99:198–215, April 2010.

[25] L. Gao, S. Urban, and J. Ramachandran. A survey of transactional issues for web service composition and recovery. *International journal of Web and Grid Services*, 7(4):331–356, January 2011.

[26] D. Garlan, J. Kramer, and A. Wolf, editors. *Proceedings of the First Workshop on Self-Healing Systems*, Charleston, South Carolina, USA, November 2002. ACM.

[27] G. D. Giacomo, F. Patrizi, and S. Sardiña. Automatic behavior composition synthesis. *Artif. Intell.*, 196:106–142, 2013.

[28] C. Hagen and G. Alonso. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10):943–958, October 2000.

[29] R. Hamadi and B. Benatallah. Recovery nets: Towards self-adaptive workflow systems. In *WISE*, pages 439–453, 2004.

[30] R. R. HASSEN. *Automatic Composition of Protocol-based Web Services*. PhD thesis, Blaise Pascal - Clermont-Ferrand II, 2009.

[31] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[32] D. S. Johnson. The np-completeness column: An ongoing guide. *J. Algorithms*, 7(4):584–601, 1986.

[33] A. Kandel, H. Bunke, and M. Last, editors. *Applied Graph Theory in Computer Vision and Pattern Recognition*, volume 52 of *Studies in Computational Intelligence*. Springer, 2007.

[34] J. Laprie. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, 1992.

[35] J. Laprie. Dependability - its attributes, impairments and means. In *Predictably Dependable Computing Systems*, pages 1–24. Springer-Verlag Heidelberg, Berlin, Germany, 1995.

[36] J. Laprie. Dependable computing and fault tolerance: concepts and terminology. In *Proceedings 15th IEEE International Symposium on Fault-Tolerant Computing (FTCS-15)*, pages 2–11, Atlanta, Georgia, June 1999.

[37] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag, Secaucus, NJ, USA, 2nd edition, 1990.

[38] F. Mattern. Virtual time and global states of distributed systems. In C. M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

[39] H. A. Maurer, J. H. Sudborough, and E. Welzl. On the complexity of the general coloring problem. *Information and Control*, 51(2):128 – 145, 1981.

[40] J. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, New York, 1990.

[41] A. Muscholl and I. Walukiewicz. A lower bound on web services composition. In *FoSSaCS*, pages 274–286, 2007.

[42] N.Desai, A.U.Mallya, A.K.Chopra, and M.P.Singh. Interaction protocols as design abstractions for business processes. *IEEE Transactions on Software Engineering*, 31(12):1015–1027, 2005.

[43] M. P. Papazoglou and D. Georgakopoulos. Service oriented computing (special issue). *Communications of the ACM*, 46(10), 2003.

[44] B. Parhami. From defects to failures: a view of dependable computing. *Computer Architecture News*, 16(4):157–168, September 1988.

[45] B. Parhami. A multilevel view of dependable computing. *Computers and Electrical Engineering*, 20(4):347–368, July 1994.

[46] F. Patrizi. *Simulation-based Techniques for Automated Service Composition*. PhD thesis, University of Roma, 2009.

[47] B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surv.*, 10(2):123–165, January 1978.

[48] J. Rao and X. Su. A survey of automated web service composition methods. In *Proceedings of First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, pages 43–54, 2004.

[49] S. Sardina, F. Patrizi, and G. D. Giacomo. Behavior composition in the presence of failure. In *11th International Conference on Principles of Knowledge Representation and Reasoning*, pages 640–650, Sydney, Australia, 2008.

[50] J. Simmonds, S. Ben-David, and M. Chechik. Guided recovery for web service applications. In *8th International Symposium on the Foundations of Software Engineering*, pages 247–256, 2010.

[51] J. Simmonds, S. Ben-David, and M. Chechik. Monitoring and recovery of web service applications. In *The Smart Internet*, pages 250–288, 2010.

[52] J. Simmonds, S. Ben-David, and M. Chechik. Optimizing computation of recovery plans for bpel applications. In *TAV-WEB*, pages 3–14, 2010.

[53] B. Srivastava and J. Koehler. Web service composition - current solutions and open problems. In *ICAPS 2003, Workshop on Planning for Web Services*, pages 28–35, Trento, Italy, June 2003.

[54] F. Tartanoglu, V. Issarny, N. Levy, and A. Romanovsky. Dependability in the web service architecture. In *Architecting dependable systems*, pages 90–109. Springer, 2003.

[55] F. Tartanoglu, V. Issarny, A. B. Romanovsky, and N. Lévy. Coordinated forward error recovery for composite web services. In *SRDS*, pages 167–176, 2003.

[56] S. Urban, L. Gao, R. Shrestha, Y. Xiao, Z. Friedman, and J. Rodriguez. The assurance point model for consistency and recovery in service composition. In *Innovations, Standards and Practices of Web Services: Emerging Research Topics*, pages 250–287, 2011. IGI Global publication.

[57] S. D. Urban, A. Courter, L. Gao, and M. Shuman. Supporting data consistency in concurrent process execution with assurance points and invariants. In *RuleML America*, pages 140–154, 2011.

[58] S. D. Urban, Z. Liu, and L. Gao. Decentralized communication for data dependency analysis among process execution agents. *International Journal of Web Service Reseach*, 8(4):1–28, 2011.

[59] Y. Xiao, S. Urban, and N. Liao. The deltagrid abstract execution model: service composition and process interference handling. In *Proceedings of the 25th international conference on Conceptual Modeling*, ER'06, pages 40–53, Berlin, Heidelberg, 2006. Springer-Verlag.

[60] Y. Xiao and S. D. Urban. The deltagrid service composition and recovery model. *Int. J. Web Service Res.*, 6(3):35–66, 2009.

[61] C. Ye, S. C. Cheung, W. K. Chan, and C. Xu. Atomicity analysis of services composition across organizations. *IEEE Transactions on Software Engineering*, 35(1):2–28, January/Ferbruary 2009.

[62] T. Yu and K. J. Lin. Adaptive algorithms for finding replacement services in autonomic distributed business processes. In *7th International Symposium on Autonomous Decentralized Systems*, Chengdu, China, April 2005.

Appendix A

NP-completeness of the Strict-Replacement problem

The NP-completeness of the Rep^{strict} problem, announced in theorem 2 of Chapter 5 can be proven by reducing from the Partial Subgraph Isomorphism problem (PSI) i.e. $PSI \ll Rep^{strict}$. Some definitions are required before going deep in proof details.

Definition 14. (Patial subgraphs) *Let $G_1 = (V_1, E_1)$ be an undirected graph. $G_2 = (V_2, E_2)$ is a partial subgraph of G_1 if $V_2 \subseteq V_1$ and $E_2 \subseteq E_1 \cap V_2 \times V_2$.*

Definition 15. (Graphs isomorphism) *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two undirected graphs. G_1 is isomorphic to G_2 if there exists a bijection $\mu : V_1 \to V_2$ which preserves arcs, i.e., $\forall x, y \in V_1, xy \in E_1 \iff \mu(x)\mu(y) \in E_2$.*

Definition 16. (Incidence poset of a graph) *The incidence poset of an undirected graph $G = (V, E)$ is a poset $P(G) = (V \cup E, <)$ where $x < y$ if and only if x is a vertex, y is an edge, and x is an endpoint of y.*

Definition 17. (Posets isomorphism) *Let $P_1 = (X_1, <_1)$ and $P_2 = (X_2, <_2)$ be two posets. We say that P_1 and P_2 are isomorphic if there is a bijection $f : X_1 \to X_2$ such that $x <_1 y \Longleftrightarrow f(x) <_2 f(y)$.*

Definition 18. (Partial Subgraph Isomorphism problem (PSI)) *Also called Graph Monomorphism problem in [33] and Subgraph Homomorphism in [32]. This problem is known to be NP-Complete [39] and it is described as follows:*

- ***Instance:*** *Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.*
- ***Question:*** *Do there exist a partial subgraph $G_2' = (V_2', E_2')$ of G_2 which is isomorphic to G_1? Or, is there an injective function $\mu : V_1 \to V_2$ such that $\forall x, y \in V_1, xy \in E_1 \Rightarrow \mu(x)\mu(y) \in E_2$?*

Recall: The Strict Replacement problem (unlabeled version)

- **Instance:** $P(\sigma^F)$ and $P(\sigma^R)$ are two posets corresponding to executions σ^F and σ^R, and k is a positive integer.
- **Question:** Is there a strict replacement $\varphi^{strict} : trans(\sigma^F) \to trans(\sigma^R)$ such that $|Dom(\varphi^{strict})| \geq k$? (We suppose here that $\forall x, y \in \lambda_D, l(x) = l(y)$ i.e., all transitions have a same label then condition (3) of Definition 9 can be dropped in the sequel proof).

Proof.

1. $Rep^{strict} \in NP$ **?**

Given a function φ, we can easily check it in polynomial time then Rep^{strict} is in NP.

2. $PSI \ll Rep^{strict}$ **?**

As already mentioned, the reduction is made from the PSI problem. We construct the incidence posets $P(G_1) = (V_1 \cup E_1, <_1)$ and $P(G_2) = (V_2 \cup E_2, <_2)$ associated to the undirected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ respectively. An instance of the problem Rep^{strict} is $I = (P(\sigma^F), P(\sigma^R), k)$. Take $J = (G_1, G_2)$ as an instance of the PSI problem and we transform it to an instance to the Rep^{strict} problem by replacing G_1 by $P(G_1)$ and G_2 by $P(G_2)$ and we put $k = |V_1| + |E_1|$. Then we get $I = (P(G_1), P(G_2), |V_1| + |E_1|)$ is an instance of the Rep^{strict}. Let us prove that:

G_2 has a partial subgraph isomorphic to G_1 if and only if the answer to

$Rep^{strict}(P(G_1), P(G_2), |V_1| + |E_1|)$ is YES.

a. We start by proving that the answer "Yes" to the PSI instance implies an answer "Yes" to the Rep^{strict} one. Suppose there exists a partial subgraph $G'_2 = (V'_2, E'_2)$ (with $V'_2 \subseteq V_2$ and $E'_2 \subseteq E_2$) of G_2 which is isomorphic to G_1. That means, $\exists \mu : E'_2 \to E_1$ s.t. $\forall x, y \in V'_2, xy \in E'_2 \Rightarrow \mu(x)\mu(y) \in E_1$. Take the subposet, noted $P(G'_2)$, associated to the partial subgraph G'_2. Proving that $P(G_1)$ and $P(G'_2)$ are isomorphic, then, proving that $V'_2 \cup E'_2$ is an ideal of $P(G_2)$.

We have: $\forall x, y \in V'_2, xy \in E'_2 \Rightarrow x < xy$ and $y < xy$ in $P(G'_2)$ and $\mu(x)\mu(y) \in E_1 \Rightarrow \mu(x) < \mu(x)\mu(y)$ and $\mu(y) < \mu(x)\mu(y)$ in $P(G_1)$. Clearly, $P(G_1)$ and $P(G'_2)$ are isomorphic. Let us prove that $E'_2 \cup V'_2$ is an ideal of $P(G_2)$. Suppose that: $\exists x \in V_2 \cup E_2, y \in V'_2 \cup E'_2$ s.t. $x <_2 y$ but $x \notin V'_2 \cup E'_2$. We have $x <_2 y$ means that x is one end point of y and $y \in E'_2$ then $x \in V'_2 \to$ contradiction.

b. Now, proving that if we have an answer "Yes" of an instance $I = (P(G_1), P(G_2), |V_1| + |E_1|)$ of the Rep^{strict} problem then we have surely a partial subgraph G'_2 of G_2 which is isomorphic to G_1. We have an answer YES to the instance I means there is strict replacement $\varphi^{strict} : V_1 \cup E_1 \to V_2 \cup E_2$ s.t. $\forall x, y \in V_1 \cup E_1, x <_1 y \iff \varphi^{strict}(x) <_2 \varphi^{strict}(y)$. Take $Img(\varphi^{strict}) = V'_2 \cup E'_2$ with $V'_2 \subseteq V_2$ and $E'_2 \subseteq E_2$.

We have $x <_1 y$ means x is endpoint of the edge y, then $x \in V_1$ and $y \in E_1$. Similarly, $\varphi^{strict}(x) <_2 \varphi^{strict}(y)$ means $\varphi^{strict}(x)$ is endpoint of the edge $\varphi^{strict}(y)$, then $\varphi^{strict}(x) \in V'_2$ and $\varphi^{strict}(y) \in E'_2$. We conclude that for all $x \in V_1$ we have $\varphi^{strict}(x) \in V'_2$ and for all $y = xz \in E_1$ we have $\varphi^{strict}(y) = \varphi^{strict}(x)\varphi^{strict}(z) \in E'_2$, therefore, the partial subgraph $G'_2 = (V'_2, E'_2)$ is isomorphic to G_1.

□

Appendix B

NP-completeness of the Loose-Replacement problem

The NP-completeness of the Rep^{loose} problem, announced in theorem 3 of Chapter 5, can be proven by reducing from the Maximal Independent Set problem (MIS) i.e. $MIS \ll Rep^{loose}$. Some definitions are required before going deep in proof details.

Definition 19. (Independent Set) *An independent set in a graph $G = (V,E)$ is a set $S \subseteq V$ such that $\forall x,y \in S, xy \notin E$.*

Definition 20. (Maximal Independent Set problem (MIS))

- ***Instance:*** *A graph $G = (V,E)$ and a positive integer k.*
- ***Question:*** *Do there exist an independent set $S \subseteq V$ such that $|S| \geq k$?*

Recall: The Loose Replacement problem

- **Instance:** $P_1 = (X_1, \leq_1)$ and $P_2 = (X_2, \leq_2)$ are two posets, $l : X_1 \cup X_2 \to \Sigma$ is a labeling function, and k is a positive integer.

- **Question:** Is there a loose replacement $\varphi^{loose} : X_1 \to X_2$ such that $|Dom(\varphi^{loose})| \geq k$?

Proof.

1. $Rep^{loose} \in NP$ **?**

Given a function φ and a positive integer k, we can check in polynomial time if φ corresponds to a loose replacement with $Dom(\varphi) \geq k$ (it corresponds to checking conditions of Definition 9). Then Rep^{loose} is in NP.

2. $MIS \ll Rep^{loose}$ **?**

Given a graph $G = (V,E)$, we construct two posets $P_1(G)$ and $P_2(G)$ (depicted in Figure B.1) and a labeling function l as follows:

- $P_1(G) = (V \cup E \cup E', \leq_1)$ where:
  - $E' = \{xy' \text{ such that } xy \in E\}$
  - $<_1 = \{(x,e) \mid x \in V, e \in E \cup E', x \in e\}$
- $P_2(G) = (V \cup E, \leq_2)$ where $\leq_2 = \emptyset$ ($P_2(G)$ is an antichain),
- $l(x) = x$ if $x \in V \cup E$ and $l(x') = x$ if $x' \in E'$.

Let us prove that $G = (V,E)$ has an independent set S with $|S| \geq k$ if and only if $\exists \varphi^{loose} : V \cup E \cup E' \to V \cup E$ such that $|Dom(\varphi^{loose})| \geq k + |E|$.

a. Suppose $G = (V,E)$ has an independent set S with $|S| \geq k$. We prove that $\exists \varphi^{loose} : S \cup E \cup E' \to E \cup V$ with $|Dom(\varphi^{loose})| \geq k + |E|$. Let define φ^{loose} and show by the following that it is injective and that $|Dom(\varphi^{loose})| \geq k + |E|$.

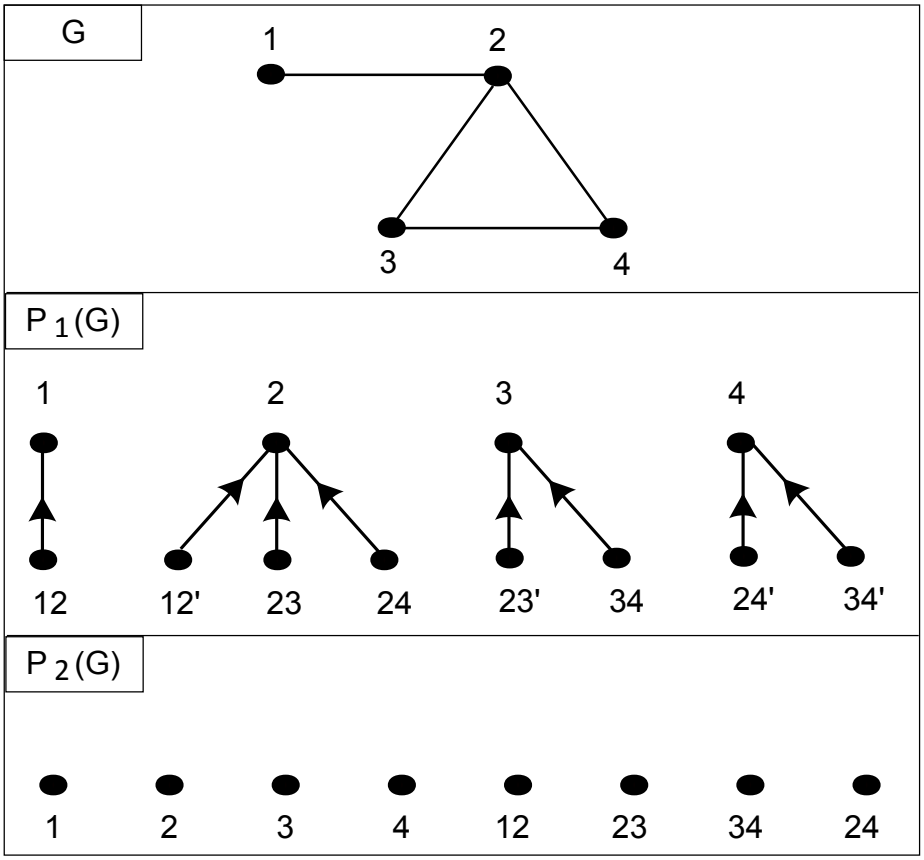

Figure B.1: A graph *G* and its corresponding posets

Put $Dom(\varphi^{loose}) = S \cup Pred(S) \cup \overline{Pred(S)}$ such that $\forall x \in Dom(\varphi^{loose}), \varphi^{loose}(x) = y$ if $l(x) = l(y)$, where:

- $Pred(S) = \{e \in E \cup E' \mid e \leq x, x \in S\}$, and
- $\overline{Pred(S)} = \{e \in E \mid e_1 \notin Pred(S) \text{ and } e'_1 \notin Pred(S)\}$.

Clearly $|Dom(\varphi^{loose})| \geq k + |E|$ since $|S| \geq k$ and $|Pred(S) \cup \overline{Pred(S)}| = |E|$ (since S is an independent set).

Let us show that φ^{loose} is injective. Surly, to each vertex in S it corresponds a unique image in $P_2(G)$ (since a vertex labeling is different from an edge labeling and each vertex occurs once per poset by definition of the posets). Furthermore, to each edge in $Pred(S) \cup \overline{Pred(S)}$ it corresponds a unique image in $P_2(G)$ (since each edge occurs only once in both $P_2(G)$ and $Pred(S) \cup \overline{Pred(S)}$).

Therefore, φ^{loose} corresponds to a loose replacement with $|Dom(\varphi^{loose})| \geq k + |E|$.

b. Suppose we have a loose replacement $\varphi^{loose} : V \cup E \cup E' \rightarrow V \cup E$ with $|Dom(\varphi^{loose})| \geq k + |E|$. Let us prove that $Dom(\varphi^{loose}) \cap V$ corresponds to an independent set on $G = (V, E)$ with $|Dom(\varphi^{loose}) \cap V| \geq k$.

We have $|Dom(\varphi^{loose})| \geq k + |E| \Longrightarrow |Dom(\varphi^{loose}) \cap V| \geq k$ (since φ^{loose} is injective and $|Img(\varphi^{loose}) \cap E| = |E|$ in the worst case). Also, we know that:

- $x \in Dom(\varphi^{loose}) \cap V \Longrightarrow \forall e \leq x, e \in Dom(\varphi^{loose}) \cap (E \cup E')$ (since $Dom(\varphi^{loose})$ is an ideal and $xy \leq x$ by definition of the posets).

- $\forall e_1, e_2 \in Dom(\varphi^{loose}) \cap (E \cup E'), e_1 \neq e_2$ (since φ^{loose} is anjective and $Img(\varphi^{loose}) \cap E = E$ in the worst case).

Therefore, $\forall x_1, x_2 \in Dom(\varphi^{loose}) \cap V$ we have $e_1 \leq x_1$ and $e_2 \leq x_2$ implies $e_1 \neq e_2$. It means that $Dom(\varphi^{loose}) \cap V$ corresponds to an independent set with $|Dom(\varphi^{loose}) \cap V| \geq k$.

□