# Automatic Signature Matching in Component Composition

by

Seyyed Vahid Hashemian

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2008

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Seyyed Vahid Hashemian

# Abstract

Reuse is not a new concept in software engineering. Ideas, abstractions, and processes have been reused by programmers since the very early days of software development. In the beginning, since storage media was very expensive, software reuse was basically to serve computers and their mechanical resources, as it substantially conserved memory. When the limitations on physical resources started to diminish, software engineers began to invent reuse approaches to save human resources as well. In addition, as the size and complexity of software systems constantly grow, organized and systematic reuse becomes essential in order to develop those systems in timely and cost-effective fashion. That is one main reason why new technologies and approaches for building software systems, such as object-oriented and component-based development, emerged in the last two or three decades.

The focus of this thesis is on software components as building blocks of today's software systems. We consider components as software black boxes whose specification and external behavior are known. We assume that this information can somehow be extracted for each deployed software component. The first and basic assumption then would be the availability of a searchable repository of software components and their external behavioral specifications. Web services are a good example of such components.

The most important advantage of software components is that they can be reused repeatedly in building different software systems. Reuse presents challenging problems, one of which is studied in this thesis. This problem, the composition problem, simply is creating a composite component from a collection of available components that, by interacting with each other, provide a requested functionality. When there are a large number of components available to be reused, finding a solution to the composition problem manually would require a considerable time and human effort. This could make the search practically impossible or unwieldy. However, performing the search automatically would save a significant amount of development time, cost and human effort. Solving this problem would be a huge step forward in the component-based software development.

In this thesis, we concentrate on a subproblem of the composition problem, composition planning or synthesis, which is defined as finding a collection of useful components from the repository and the necessary communications among them to satisfy a requested functionality. For scalability purposes, we study automatic solutions to composition planning and propose two approaches in this regard. In one, we take advantage of graphs to model the repository, which is the collection

of available components along with their behavioral specification. Graph search algorithms and a few composition-specific algorithms are used to find solutions for given component requests. In the other approach, we extend a logical reasoning algorithm and come up with algorithms for solving the composition planning problem. In both approaches we provide algorithms for finding the possibility of a composition, as well as finding the composition itself.

We propose different types of composition and show how applying each would impact the behavior of a composite component. We provide the necessary formalism for capturing these types of composition through two different models: interface automata and composition algebra. Interface automata is an automaton-based model for representing the behavior of software components. The other model in this regard is composition algebra, which is an algebraic model based on CSP (Communicating Sequential Processes), CCS (Calculus of Communicating Systems), and interface automata. These formal models are used to validate the results returned by the composition approaches.

We also compare the two composition approaches and show why each of them is suitable for specific types of the problem according to the repository attributes. We then evaluate the performance of the reasoning-based approach and provide some experimental results. In these experiments, we study how different attributes of the repository components could impact the performance of the reasoning-based approach in solving the composition planning problem.

**Acknowledgements**

During the course of my PhD studies, I have worked with a group of people, whose contribution to my research and the making of this thesis deserves special mention.

First and foremost, I offer my sincerest gratitude to my supervisor, Prof. Farhad Mavaddat, who supported and encouraged me throughout these years with his patience and knowledge whilst allowing me to work in my own way. Without him this thesis would not have been completed or written. He always made me feel welcomed whenever I needed to talk to him regarding the progress of my work or any relevant matter.

I gratefully thank Prof. Michael Godfrey, my co-supervisor, for his advice and significant help in completing this thesis. Although he joined my committee in the last year of my studies, he had an important impact on the outcome. He took the time to catch up with the work that had already been done, and made effective suggestions on the steps that still needed to be taken.

Many thanks go to the rest of my committee, Prof. Frank Tompa, Prof. Paulo Alencar, Prof. Todd Veldhuizen, and Prof. Schahram Dustdar. Prof. Tompa and Prof. Alencar were members of this committee since 2005, and during these years helped me follow the right research direction. Prof. Dustdar and Prof. Veldhuizen joined the committee recently, and provided me with valuable comments and suggestions regarding both the research and the presentation of the thesis. I greatly appreciate the extra time Prof. Dustdar, my external examiner, took to travel all the way from Austria to attend my PhD defense. I hereby would like to thank all the members of my committee, and appreciate their time and contribution. It has been a pleasure to know them and benefit from their expertise.

I would not have had any achievement in my life without the constant support and prayers of my parents. My mom and dad always believed in me and supported me at their best so that I could continue with my studies, even when it meant that I had to live very far from them. They have been my greatest mentors throughout my life, and I owe all my success to them.

Words fail me to express my appreciation to my dear wife, Solmaz Kolahi, whose dedication, love, and persistent confidence in me gave me the strength to finish this thesis. She has always been my pillar, my joy, and my guiding light.

I also benefited from the advice, experience, and friendship of a great group of friends and colleagues during these past years. My special thanks go to Shahram

Esmaeilsabzali, Mahdi Mirzazadeh, Abbas Heydarnoori, Amir H. Chinaei, Vahid Karimi, Felix Sheng-Ho Chang, and also to Afsaneh Fazly, Reza Azimi, Golnaz Tajeddin, Iman Hajizadeh, Afra Alishahi, and my other Iranian friends in Waterloo and Toronto.

*to Solmaz,*

*to Mom and Dad,*

*and to my granduncle who recently passed away.*

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

In software engineering, *reuse* is not a new concept. Ideas, abstractions, and processes have been reused by programmers since the very early days of software development. But in those early days the approaches to reuse were ad-hoc, meaning that there was no systematic way of reusing software [87]. Moreover, the invention of subroutines for reuse was mostly because every byte of memory was precious at the time, and subroutines could substantially conserve memory. In fact, the earlier versions of software reuse were basically to serve the computers and their mechanical resources [28].

When the limitations on physical resources started to diminish, software engineers began to invent reuse approaches to save human resources as well. In addition, as the size and complexity of software systems constantly grow, in order to develop them in a timely and cost-effective fashion existing organized and systematic methods of reuse are vital. Nowadays, considering the widely increasing expectations from software systems, reusing ideas, abstractions and pieces of programs would not be sufficient, and reuse in larger scales is necessary. That is why new technologies and approaches for software development emerged in the last two or three decades. From this point of view, the *object-oriented paradigm* was the beginning of a new era in software development, which then led to *component-based software development (CBSD)*. Clements describes CBSD as follows [28]:

> CBSD is changing the way large software systems are developed. CBSD embodies the "buy, don't build" philosophy [21]. In the same way that early subroutines liberated the programmer from thinking about details, CBSD shifts the emphasis from programming software to composing software systems. Implementation has given way to integration as the

> focus. At its foundation is the assumption that there is sufficient commonality in many large software systems to justify developing reusable components to exploit and satisfy that commonality.

CBSD can be simply defined as development of systems that use components. A software component then can be defined as a non-trivial, nearly independent, and replaceable part of a system that, by interacting with other components, fulfills a clear function in the context of a well-defined architecture [52, 65]. A software component can be deployed independently and is subject to composition by a third party [31].

Components provide their functionalities through their interfaces, which are separate from their implementation. Interfaces are actually connectors that connect components together. Inputs and outputs in the form of data are transferred from and to components through these connectors. Moreover, users also use interfaces to interact with components. This means an interface is how a consumer of a component views that component [74].

The focus of this thesis is on software components as building blocks of today's software systems. We consider components as software black boxes whose specification and external behavior are known. Their specification could contain their physical location and the instructions on how to invoke them. The information on their external behavior may include their supported interfaces and operations, where for each, the inputs, outputs, preconditions, effects and nonfunctional attributes are specified. We assume that this information can somehow be extracted for each deployed software component. Web services, described in Section 2.1, are a good example of such components.

One assumption in this thesis is that there are available components to be reused. Therefore, we do not worry about how to build components eligible for reuse. In fact, this activity belongs to another process called *domain engineering*. "Domain engineering is about finding commonalities among systems to identify components that can be applied to many systems, and to identify program families that are positioned to take fullest advantage of those components"[1] [87]. This thesis does not further discuss the domain engineering process.

[1] Quotation by Paul Clements.

## 1.1 The Composition Problem

As mentioned earlier in this chapter, the most important advantage of software components is that they can be reused repeatedly in building different software systems. Reuse in this sense introduces challenging problems. Since we assume that there are developed and available domain-engineered software components to be reused, the first and basic assumption would be the availability of a searchable collection of software components along with their external behavioral specifications, called the *repository*.

When we need a specific software component, we first check for the existence of a repository component with the same or similar functionality. Therefore, the first problem, the *matching problem*, is finding a matching component from the repository for a given component request. Different versions of the matching problem have been addressed in the literature under various titles such as *component matching* [64, 82], *specification matching of software components* [58, 113], *signature matching of software components* [111, 112], and more recently, *(web) service discovery* [13, 63, 92] and *(web) service matching* [39, 70, 109].

The composition aspect of component-based development leads us to the second and more interesting problem, which is called the *composition problem*. The composition problem could be considered as the extended version of the matching problem. In component matching, a request for a desired component is given and we are interested in finding a *single* component from the repository to match the request. It is quite possible for the matching problem to have no solution, especially when the request is too specific, e.g., it comes with several syntactic or semantic constraints.

**Example 1.1** *Consider the repository component* `WeatherForecast`*, which receives the name of a Canadian city (*`city_CAN`*) and returns its current temperature in Celsius (*`temperature_C`*), and assume that there is no other component related to weather forecast in the repository. If a request is submitted against this repository for a component that receives a Canadian city (*`city_CAN`*) and returns its current temperature in Fahrenheit (*`temperature_F`*), there would be no match for it in the repository, as* `WeatherForecast` *does not return an output in Fahrenheit. However, if the repository contains a temperature conversion component* `TemperatureConvertor` *to convert a temperature in Celsius to its equivalent Fahrenheit value, we could find a combination of repository components to answer the given request successfully.* □

Compared to finding a single match, it is much more likely that a combination of repository components would satisfy the given request. The composition problem expands the matching problem by also checking the combinations of repository components. More specifically, the composition problem is defined as follows:

**Problem 1.1** *Given a repository of available software components and their behavioral specifications, and the specification of a component to be built, is it possible to build the request as a composition of some repository components? If so, how could the request lead to a working software component?* □

We simply observe that the solutions to a composition problem would be a superset of the solutions to a matching problem. This confirms the fact that the composition problem is more difficult to solve.

Obviously, when there are a large number of components in the repository, manually finding a solution to the composition problem would require considerable time and human effort, which could make this search practically impossible or unwieldy. However, if this search could be performed automatically, by saving significant amount of development time, cost, and human effort, it would be a huge step forward in component-based software development. In this thesis, we concentrate on finding an automatic solution to the composition problem.

Regarding the applications of solving this problem, in a small scale, we could think of software developers that build numerous software components. For their future developments they look forward to reusing their earlier products as often as possible. In a larger scale, web services would be the best examples of software components that fit into the composition problem, as they are currently the center of attention in research activities on the World Wide Web.

Automatic web service composition is one of the key challenges of service-oriented computing today. In general, *service composition* can be defined as creating a composite service obtained by combining available component services. It is highly effective mainly in situations where the client request cannot be satisfied by available services, except by combining some of them [67].

As mentioned earlier in this chapter, the repository contains information on the external behavior of available software components. The external behavior is the main attribute based on which the composition of two software components is determined. For instance, in Example 1.1 when composing two components *WeatherForecast* and *TemperatureConvertor*, since the output type of

*WeatherForecast* matches the input type of *TemperatureConvertor*, the composition of the two components is possible and satisfies the request. In similar cases where the output of one component is consumed by another component, another criterion that affects the composability of two components is that the effects of the former must not violate the preconditions of the latter. Moreover, nonfunctional requirements must be satisfied. For instance, in Example 1.1 if the response times of two components *WeatherForecast* and *TemperatureConvertor* are 125 and 35 milliseconds respectively, and the request specifies the maximum response time of 150 milliseconds, the given composition would be rejected, because it leads to the response time of 160 milliseconds which is higher than what is asked in the request.

There are two main aspects to the composition of software components in finding a solution to the composition problem [15, 67]:

- *Composition planning* (or *synthesis*) refers to studying how to generate a composition plan based on repository components to provide the desired behavior. The composition plan can be obtained either automatically, semi-automatically or manually, where in the first two cases finding the plan is mostly done using appropriate composition algorithms.
- *Orchestration* refers to appropriate control and data flow coordination among the involved components in executing the composition plan in the real world.

Since invoking a component could trigger a very complex process involving many other components, it becomes necessary to have a formalism to reason about the temporal aspects of a composition [2], i.e., the order of execution of the involved components. Therefore, to solve the composition problem we need to formally define

- all the possible types of composition in combining repository components, such as sequential and conditional composition, and
- the specification of the component resulting from composing two repository components using each of these composition types.

This formalism is normally based on different types of modeling languages and tools for representing behavior. A few examples of such formalisms would be process algebras [8], statecharts [46] and Petri nets [85, 86].

## 1.2 Main Contributions of the Thesis

In this thesis we provide two different automatic approaches to solving the composition planning (synthesis) in a specific version of the composition problem. More precisely, we assume that the repository contains only *stateless* components.

**Definition 1.1** *A software component is* stateless *if and only if in every one of its execution scenarios it receives all its necessary inputs before returning any of its produced outputs, and also, if it receives/returns all its inputs/outputs without any specific order (e.g., at the same time).* □

The concept of execution scenario is defined below.

**Definition 1.2** *Each time a component is completely executed it performs a specific behavior in terms of the inputs it receives, the outputs it returns, and the execution order of those inputs and outputs. Each such behaviors is called an* execution path *or an* execution scenario *(simply, a* scenario*) of that component.* □

In general, we assume that the request also describes a stateless component. We do not consider preconditions, effects and nonfunctional properties of software components. In other words, we try to find compositions that signature-wise satisfy the request.

To solve the composition planning problem for stateless components, we provide two different formal models for their behavioral representation. These two models are described in Chapter 4. We propose extensions to *interface automata* which was proposed by de Alfaro and Henzinger [34]. These extensions basically include new types of composition. The *composition algebra* is a process algebraic model that we tailored specifically for solving the composition problem. As well as formalizing the behavior of components and their compositions, these two formalisms are used to validate the solutions returned by our composition approaches.

In one of the composition approaches, we take advantage of graphs to model the repository. Graph search algorithms, along with a few composition-specific algorithms, are then used to find solutions for given component requests. In the other approach, we extend a logical reasoning algorithm for Horn clauses to produce algorithms for solving the composition planning problem. In both approaches we provide algorithms for finding the possibility of a composition, as well as finding the composition itself.

We also compare the two composition approaches and show why each of them is suitable for specific types of the problem according to the repository attributes. We then evaluate the performance of the reasoning-based approach and provide some experimental results. In these experiments, we study how different attributes of the repository components could impact the performance of the reasoning-based approach in composition planning.

## 1.3 Structure of the Thesis

In Chapter 2 we provide some background information on web service composition, which is the potential target for the automatic component composition. We also discuss the role of semantic matching in component composition. Then we propose a general architecture for solving the generic composition problem.

In Chapter 3 we review the literature on component and web service composition. We do this review separately for different subproblems of the composition problem which are introduced in Chapter 2.

In Chapter 4 we explain two representation models for the behavior of software components. In Section 4.1 we review interface automata and extend it to capture more composition types. In Section 4.2 we introduce composition algebra and, through some axioms, show how different types of composition are modeled in this algebra. We also explain why composition algebra, compared to interface automata, is in general a better model for behavioral representation.

In Chapter 5 we describe our graph-based solution to the composition planning problem. We start the chapter by solving a simple version of this problem. Then we improve the solution to solve the generic version for stateless components.

In Chapter 6 a reasoning-based approach for component composition is explained, which is based on the forward chaining reasoning approach for Horn clauses. We first study a simple version of the problem and then extend the solution to solve the generic case.

Chapter 7 contains a comparison of the two approaches based on different parameters of the composition planning. It is followed by a performance evaluation of the reasoning approach, which studies the running time of the proposed approach against the involved parameters. We also briefly discuss how other available tools would perform in solving the composition problem, compared to the composition approaches of Chapters 5 and 6. At the end of this chapter we briefly explain how

the proposed composition planning approaches can be applied to web services. To do so we study the structure of WSDL documents.

We conclude the thesis in Chapter 8 by a summary of the contributions, and suggestions on the research directions that could follow this work.

# Chapter 2

# Background

In the previous chapter we discussed the need for behavioral specification of components as one basic requirement of the composition problem. We also mentioned that web services are a good example of software components for the composition problem we study in this thesis. The reason is that there are specification languages for web services that provide this behavioral information. Since we study a specific composition problem in which we only need to know the inputs and outputs of involved components, WSDL specifications [106] would be adequate as they use elements such as types and operations to specify the behavior of web services. Other web service specification languages might be used for extracting the behavior, such as BPEL4WS [54] and OWL-S [32], which also specify the internal behavior using some control constructs.

In this chapter, in Section 2.1 we first provide some background information about web services and their composition. In Section 2.2 we study the importance of semantics and semantic matching in solving the composition problem. Finally, in Section 2.3 we propose a high-level architecture of a generic component composition engine. This composition engine provides the necessary features in solving the composition problem (Problem 1.1).

## 2.1 Web Services Composition

The World Wide Web Consortium (W3C) defines a web service as a software system designed to support interoperable machine-to-machine interaction over a network, and comes with an interface described in a machine-processable format. Other systems interact with the web service in a manner prescribed by its description

using messages that are typically conveyed using HTTP in conjunction with other web-related standards [104]. In simpler terms, a web service, or simply service, is a web application which has the potential of being reused by both human and other web applications using different internet protocols.

The communications between services are supported by a structure called *Service-Oriented Architecture* (SOA). SOA is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. In SOA, services are the mechanism by which requirements and capabilities are brought together. In the world of distributed computing, people and organizations create services to solve a solution for their own problems. But it is reasonable to think of one service developer's requirements being met by services offered by someone else. This is not necessarily a one-to-one correlation, as a given requirement may require combining numerous services [80].

SOA defines three basic roles and three basic operations for a web service. As depicted in Figure 2.1, these three roles are the *service provider*, the *service requester*, and the *service registry*. The objects involved in this architecture are the service and the service description, while the operations performed on these objects are *publish*, *find*, and *bind*. A service provider creates a web service and its description and then publishes the service in a service registry. Once a web service is published, a service requester may find the service by searching the registry, which provides the service requester with a service description and a URL pointing to the service itself. The service requester may then use this information to bind to the service and invoke it [43].

Compared to software components that are developed according to earlier technologies, web services are more loosely coupled and more abstract. Therefore, their composition is more practical since their low-level technical details, such as the operating system, communication protocol and programming language, can be ignored [89].

The development of composite web services is currently a manual task in most cases, which is time-consuming and requires considerable effort on low-level programming. Obviously, this approach does not scale well as the number of published web services increases [12]. That is why finding approaches for automatic composition of web services is now one of the main research topics in the SOA community.

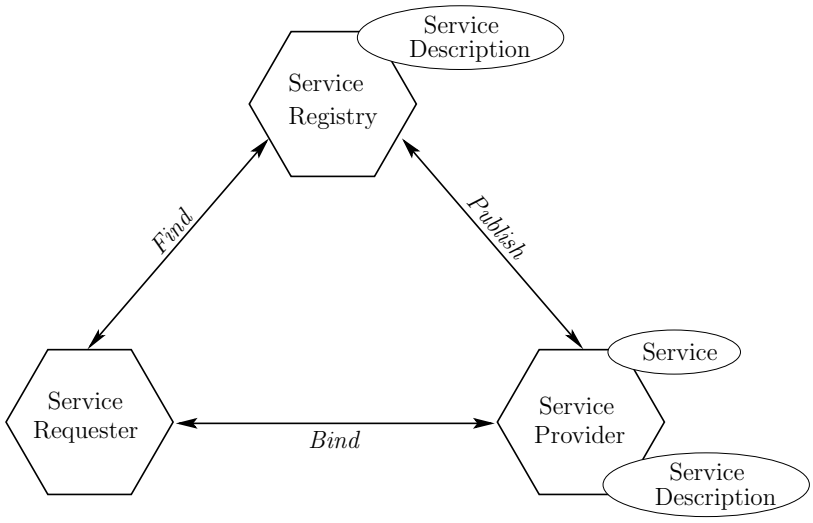



Figure 2.1: The basic service-oriented architecture including service actors, objects and operations [43].

## 2.2 The Role of Semantics

As the world of web services grows and the number of published web services increases, the languages and terms used for their specification also expand. These languages and terms adopted by different web service developers are not always compatible with each other. Different terms in these languages may be used to address the same concept; and similar terms may be used to address very different notions. That is how the *babelization problem* [19] appears. For example, a web service may use `publication_author` to indicate an author, while there may be other web services that use `author`, `document_creator` or `doc_author` for the same concept.

In general, meanings of the terms used by different web services may be the same, similar or different. Problems occur in one of the following cases [19]:

- When different terms (e.g., `publication_author` and `author`) are used to address the same concept: In this case a rule can be used to explicitly specify that two (or more) terms are equivalent.
- When the same terms (e.g., `weight` in kilograms and `weight` in pounds) address inconsistent meanings: In this case XML namespaces [101] are used to distinguish the two terms (e.g., `metric:weight` and `imperial:weight`). To make each data type in web service specification documents uniquely iden-

tify a single concept, their names are preceded by an XML namespace. A namespace actually represents a single vocabulary, in which each concept is clearly understood. Therefore, each pair of namespace and type name represents a unique concept. A well-known example of XML namespaces is the Dublin Core Metadata Initiative®[1].

- When different terms (e.g., *vehicle* and *car*) address related meanings: In this case the question is how different terms may be related to each other (*car* is a *subtype* of *vehicle*).

Another problem with adopting different languages and terms by different web services is that data is developed and administered by each web service separately and independent of other services. This problem, which is a result of the babelization problem, makes it hard for web services to address other web services data. The basic reason is that the format and semantics of data in two web services do not necessarily match.

In order to provide necessary means to solve the above problems, documents should use common vocabularies, unambiguous names and common data model to express information, all in machine processable formats. These requirements are the same requirements that lead to the *semantic web* [93]. The semantic web can be considered as an improved version of the current web in which meaning is much more important and is machine processable. It is a more useful web which can be used not only by human users but also by machines. By making it easier to find, share and combine information all over the web, the semantic web is a big step towards automating web applications [19].

In the semantic web, *ontologies* [102] are used to express common vocabularies, URIs [18] are used to specify unambiguous names, and the RDF language [103] is used as the common data model (meta-data) to express information. Ontologies, as machine processable vocabularies, are used to describe concepts and their relationships. For example, the definition of the terms *car* and *vehicle* and the relationship "*car* is a *subtype* of *vehicle*" can be part of an ontology. There are several ontologies and ontology development tools that have been developed. As an example of each, *cs-dept-ontology*[2] is a computer science department ontology developed at the University of Maryland, and *ezOWL* [27] is a visual semantic web ontology editor [19]. A URI (Uniform Resource Identifier) is a compact string of

[1] `http://purl.org/dc/elements/1.1/`

[2] `http://www.cs.umd.edu/projects/plus/SHOE/onts/cs.html`

characters for identifying an abstract or physical resource. It provides a simple and extensible means for identifying a resource. For example, URL (Uniform Resource Locator) is a subset of URI that is used to represent resources based on their network locations [18].

## 2.3 Architecture of a Component Composition Engine

In Figure 2.2 we propose a generic architecture for *component composition engines*, which is used to solve the composition problem given in Chapter 1. There are four main components in this architecture:

- *Component specification extractor*: Every time a new component is created and introduced to the component composition engine, the component specification extractor extracts its behavioral information, and sends it to the repository for storage. Here the assumption is that each component comes with a specification document, or its specification is given to the system manually, e.g., by the component developer.

- *Ontology matching engine and repository*: The repository contains the behavioral specification of all available components. The way this information is stored in the repository, which could partly depend on the composition approach, should speed up the future searches as much as possible. An ontology matching engine might be attached to this repository to improve the capabilities of the composition approach by carrying out a semantic matching among different involved vocabularies or ontologies.

- *Component composition planner*: The composition planner is responsible for the first side of the component composition, i.e., finding the composition plan. Given a request submitted by the client the composition planner looks into the repository and, using specific search algorithms, tries to find components that could participate in a composition to satisfy the request. It then generates a plan that describes the temporal order of execution of those components along with necessary interactions among them that leads to a composition fulfilling the given request. This planner might be equipped with a quality of service controller, which would be responsible for selecting the optimal plan whenever there are multiple plans satisfying the request.

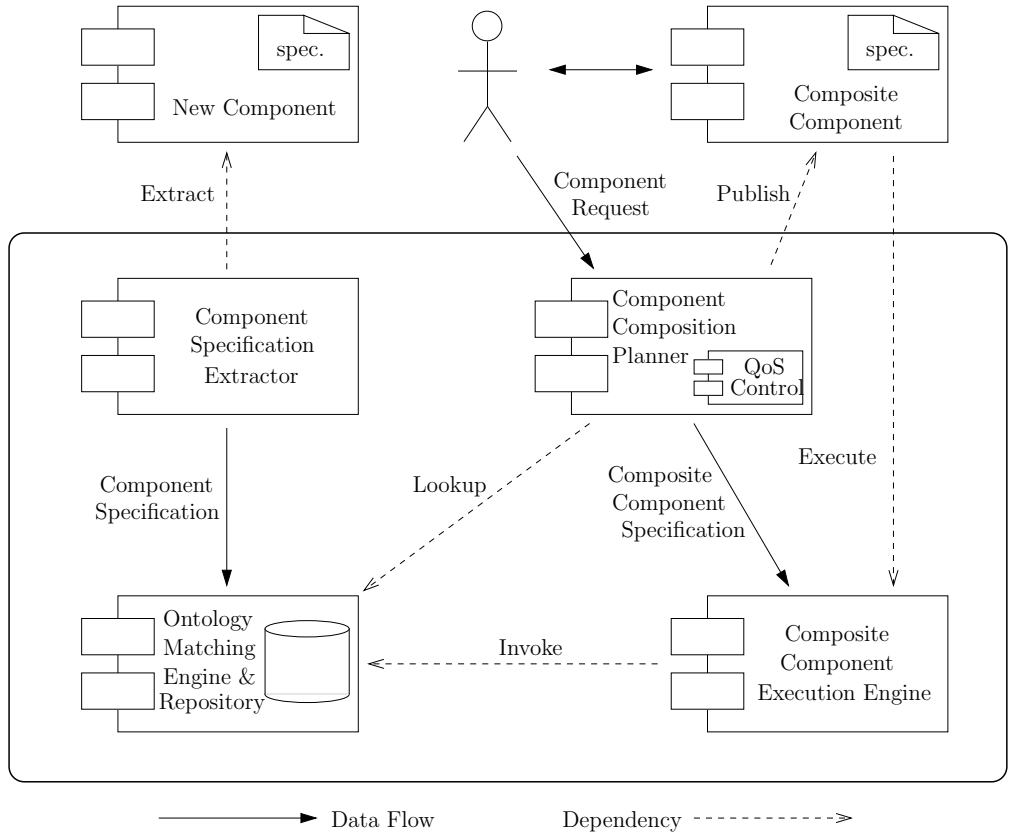



Figure 2.2: Typical architecture of an automatic component composition engine.

- *Component composition execution engine*: The execution engine is responsible for the second side of the component composition, i.e., orchestration. It receives the composition plan and, using the specification of involved components, executes the composite component by making participating components interact in an appropriate way. The orchestration could be managed by a third party in a centralized fashion, or by the involved components themselves.

In this thesis we focus on studying two approaches for implementing the component composition planner, without any quality of service controller. The research on implementing a quality of service controller will be left for future work.

# Chapter 3

# Related Work

Reusing software components and their (automatic) composition has received a great deal of attention, especially since the emergence of web services. Web services gather all the necessary requirements for the automated reuse and composition, and therefore, researchers in this area have become involved in studying how this automation is possible in both theory and practice.

The research on the subproblems of automatic service composition, as introduced in the generic architecture of Figure 2.2, has been going on in the past few years. However, there is no accepted solution yet for any of those subproblems. In this chapter, we briefly review some of the related research studies on these subproblems. These works mostly address web service composition, instead of the general component composition. In Section 3.1 some of the formal models proposed to capture behavioral compositions of components are presented. Section 3.2 describes some of the techniques offered for web service discovery and matching. Some of the suggested approaches to automatic component composition, including finding the composition plan and the orchestration, are discussed in Section 3.3. The efforts on techniques for semantic matching and ontology matching are partly covered in Section 3.4. Finally, Section 3.5 cites a few approaches to quality of service control in component composition. Although in some works it is argued that integrated approaches to the composition problem are better than separating the issues [66], we believe otherwise; and the reason is the complexity of the whole problem and the fact that its subproblems are self-contained and can be solved independently.

## 3.1 Behavior Representation

Berardi et al. [16] propose a formal model based on Situation Calculus [73] and Basic Action Theory [91] for service composition. They represent the sequences of possible invocations to a service by execution trees, and show how a composite service can be calculated based on its constituent component services. One of their interesting results is a theorem that states that checking the existence of a service composition can be done in EXPTIME. We come to this result later in the thesis when studying the worst-case running time complexity of our composition algorithms. Berardi et al., in another work [17], present a formal technique for automatic composition synthesis when the behavior of available services is nondeterministic, and partially controllable by the orchestrator. They prove that their technique is sound, complete and terminating.

Hamadi and Benatallah [45] present a Petri net-based algebra to formally model different types of web services compositions. In this work, each web service is modeled as a Petri net. The authors explain how compositions of web services can also be captured by Petri nets. Sequence, choice, unordered sequence and parallel with communication are some of the composition types studied in the article.

Narayanan and McIlraith [79] define the semantics for a subset of OWL-S (formerly known as DAML-S) [32] in terms of Situation Calculus [73] predicates. This semantics is then used to encode web service descriptions in a Petri net formalism and to provide procedures for web service simulation, verification and composition. All these procedures are abstractly explained in the paper without any specific algorithm. For example, regarding the automated composition of web services, which is addressed in the title of their work, only a short description is given which considers only the pipeline execution of component services.

Kazhamiakin et al. [59] describe an approach for the verification of web service compositions defined by sets of BPEL4WS [54] processes. They develop a technique to associate with a web service composition an adequate communication model, which is the simplest model sufficient to capture all the behaviors of the composition. Their composition model is based on the parametric definition of the communication infrastructure which results from changing the number of queues, and allowing more asynchrony.

Bultan et al. [22] introduce a framework for modeling and specifying the global behavior of service compositions. Under this framework, individual services communicate through asynchronous messages and each service maintains a queue for

incoming messages. A global watcher keeps track of messages as they occur. They propose and study a central notion of a *conversation*, which is a sequence of messages observed by the watcher. They consider the case where services are represented by finite state machines. They also propose conversation specifications as a formalism to define the conversations allowed by a service composition.

## 3.2 Component and Web Service Matching

Agarwal et al. [2, 3] represent a matching technique based on $\pi$-calculus [78] and description logic [7] for finding web services that (partially) match a given request. Temporal constraints, simulation relations, semantic constraints, and security constraints are among those covered by their matchmaking algorithm.

Shen and Su [94] represent a matching approach for finding components of a composite stateful web service from a repository of stateless services. The behavior of a stateful web service is modeled by an automaton, in which each state represents an activity that is performed by the composite service. Edges and their labels represent data flow and the preconditions and effects of activities. Although the authors claim their approach to be a composition approach, it actually seems to be a web service matching one.

Medjahed et al. [75] define an ontology-based framework for automatically finding partial matches for a composite web service. For this purpose, they introduce a composability model for comparing syntactic and semantic features of web services to find out if selected web services can actually interact with each other. They also propose a technique to generate composite service descriptions, which takes as input a high-level description of the desired composite service. Then using a matchmaking algorithm, they find repository services matching different operations of the composite service given in the description. Although the authors call their approach an automatic web service composition approach, it is simply a collection of matchmaking attempts for a web service with different operations. They perform an experimental evaluation of their approach as well.

Grigori et al. [44] propose a solution for service matching based on behavioral specification. They first argue about the need to retrieve services based on their workflow model. By using a graph representation formalism for services, they propose an approximate matching algorithm. Starting from the classical graph edit distance, they suggest two new graph edit operations to take into account the difference of granularity levels that could appear in two models. The authors

exemplified the approach for behavior matching of workflow protocols expressed using WSCL [10] and developed a prototype that is available as a web service.

Paolucci et al. [83] present an algorithm for semantic matching of web service. They compare the requested service against the repository components and decide if a component matches the request using four different matching levels: *exact*, *plug-in* (when the repository service is more general), *subsume* (when the target service is more general), and *fail*.

Dong et al. [37] propose a web service search engine, called Woogle, which supports similarity search for services in addition to simple keyword searches. Different types of search are supported by Woogle, such as searching for operations with similar functionality, with similar inputs/outputs, or composable with another operation. The key ingredient of their search engine is a clustering algorithm that groups names of operation parameters into semantically meaningful concepts, which are used for similarity searches.

## 3.3 Component and Service Composition

One of the advantages of service composition is to distribute the workload of servers. The overall behavior, in this case, would be performed by multiple services residing at different physical locations [67]. In this section, we review some of the approaches to component and web service composition, in both composition planning and orchestration areas. A more detailed study of web service composition approaches can be found in the corresponding survey articles [38, 53, 76, 90, 98].

### 3.3.1 Composition Planning

Current proposed solutions for composition planning usually take advantage of graph search algorithms or logic-based planning. In this part, we first present some of the graph-based approaches, and then address some of the logic-based ones.

#### Graph-Based Approaches

The composition planning approaches that use graphs usually model input/output data types of available service and/or services themselves, as graph nodes. This way the composition planning problem is converted to finding graph paths from the

input data types in the request to its output types. In the generic case, the graph paths that are found have to be somehow combined so that a unique composition plan can be produced as the solution.

Zhang et al. [115] represent a graph-based approach to web services composition. In their approach each graph node represents a web service and edges represent the possibility of data passing between two services; i.e., when the output of one service is semantically similar to the input of another service. Weights are assigned to graph edges based on the semantic similarity of outputs and inputs at both ends, as well as some quality metrics, such as execution time. Then, the Bellman-Ford algorithm [30] is used to find a shortest weighted graph paths from the inputs to the outputs specified in the requested service. The authors claim an $O(N^3)$ running time, where $N$ is the number of services involved. Since they would need to use a modified version of the Bellman-Ford algorithm which involves more computation, it is not clear why the running time complexity would remain intact. Also, the case where more than one instance of the same data type are involved is not discussed, which clearly adds to the complexity of the problem. Their proposed composition algorithm returns only the first solution found, and does not look for other possible solutions in case, for some reason, the first solution is not a good candidate.

Aydoğan and Zırtıloğlu [6] propose a graph-based approach for finding composition plans. In their graph, nodes represent both data types and services, while edges represent dependencies, from output nodes to service nodes, and from service nodes to input nodes. They claim that, for finding the composition plan, it suffices to find paths from request outputs to its inputs. They assign weights to service nodes based on their quality values (reliability, accessibility, ...), and when there are multiple choices, their approach picks the service with highest weight value. Their proposed solution is quite abstract and does not carefully discuss some aspects of the problem. Specifically, it fails to find the best quality solution because it picks the best quality service locally, instead of picking the one which leads to the overall maximum quality. In finding paths they do not explain, in enough details, the effect of multiple input/output data types, and multiple instances of the same data type. They do not discuss the complexity of their algorithm as well.

Shin and Lee [95] present a graph-based approach to finding composite information web services to satisfy a given request by considering the functional signature of the web services along with their functional semantics. They base their approach on one of our earlier works [48] and extend it to find more precise compositions. We discuss their approach in more details in Section 5.3.

Fujii and Suda [41] represent a semantic-based dynamic service composition system which integrates the semantic information and the functional information of a component into a single semantic graph representation, and generates the execution path of the requested service and checks the semantics of the path against the user request.

### Logic-Based Approaches

In logic-based approaches, some form of AI planning is applied in order to find the composition plan. These approaches should convert the problem into a planning problem, solve the planning problem, and then convert the result into a solution for the original composition planning problem.

Peer [84] introduces an AI planning technique for web service composition based on PDDL [42], which is a language for expressing planning problems. In this work, a mapping between WSDL specification and PDDL constructs are created through some semantic annotations which results in descriptions of service behaviors in PDDL. Each service request then is converted to an AI planning problem which is handed to the appropriate planner. In this approach it is not explained how the semantic markup is automatically created for each service.

Rao et al. [89] present a Linear Logic-based attempt to web services composition. They convert the functional and non-functional specification of available web services and the requested service into Linear Logic axioms and take advantage of theorem provers to prove the requested service axiom using the axioms of available services. As a result, a proof tree is created which can show which repository services have to be used. However, the authors do not explain in this paper how this proof tree can be converted into a composition plan. Also, since the logical axioms must be used by automatic theorem provers, the necessary translations into a format acceptable by the provers is not discussed in their work.

Oh et al. [81] represent a forward-chaining approach for finding whether a repository is able to satisfy a web service request. The main matching idea is that when the request inputs is a superset of a candidate service inputs and the request outputs is a subset of the candidate service outputs, the candidate service matches the request. This way, a chain of candidate services might be found to match the request when there is no single service to do so. Then, using a simple search algorithm the feasibility of the composition can be determined. The presented approach is rather simplistic because a randomized data structure is used which might provide false

positives. Also, there is no discussion on how to find the composition plan in case the algorithm returns successfully. Although the authors do not explicitly discuss the complexity of their algorithm, it seems to have an exponential time worst-case complexity.

Laukkanen and Helin [68] propose an approach for finding semantically similar web services to a specific one. This similarity search includes searching for a web service or a set of web services with similar inputs, outputs, preconditions and effects. Also, part of their research is dedicated to making relationships between different ontologies in order to increase the chance of finding a solution. BPEL4WS [54] is used for describing the functionality of the desired service. Then, repository services that semantically match the identified functionality are found, according to four different matching levels: *exact match*, *plug-in* (the repository service is more general), *subsumption* (the target service is more general), and *fail*. Finally, a workflow is created and executed which provides the required functionality. One disadvantage of their approach is that since functionality is captured by logical expressions, the cardinality of inputs/outputs cannot be captured.

Tang et al. [97] introduce an automatic web service composition method based on logical inference of Horn clauses in Petri net models. Available services and the request are translated into a set of Horn clauses, and then modeled using Petri nets. The T-invariant method of Petri nets is used to determine the existence of composite web services fulfilling the request. The authors consider a subset of available components, i.e., components that receive/return only one instance of each involved data type. This assumption is too restrictive, and the authors do not explicitly discuss the complexity of their approach. However, we show in a similar approach in Chapter 6 that this simplified version of the composition problem can be solved in linear time in the number of available components.

Kona et al. [61] formally define a web service discovery and composition approach based on constraint logic programming. It is simply based on the fact that for two services to be sequentially composed, the outputs and effects of the first must match the inputs and preconditions of the second. They propose an algorithm which simply checks if the outputs and effects of the requested service are reachable from its inputs and preconditions using the repository components. Their approach seems to return only a 'yes' or 'no' answer based on the possibility of a composition, and there is no specific procedure to find the appropriate composition plan.

Among other logic-based approaches, Aiello et al. [5] represent a request language for specifying requested services, and an approach for composing web services

based on planning under uncertainty and constraint satisfaction techniques. Traverso and Pistore [99] propose a planning technique for the automated composition of web services which deals with nondeterminism, partial observability and complex goals. A system is partially observable if its internal status and variables are hidden from other systems. Limthanmaphon and Zhang [69] provide a model for web service composition based on case-base reasoning techniques.

As an example of a composition planning approach that does not use graphs and logic as the above approaches, Berardi et al. [15] present an automaton-based framework for describing the expected behavior of web services in terms of their possible executions (execution trees). They use this setting to analyze the complexity of finding a composition plan for a given request. They also propose an approach for finding composition plans, although they do not provide any concrete algorithm. They show that their approach runs in exponential time with respect to the size of the given automaton.

### 3.3.2 Composition Orchestration

Benatallah et al. [12, 14] propose a declarative language for composite web services and a dynamic peer-to-peer paradigm for their execution. In their framework, which is called SELF-SERV, web services are composed and the resulting composite services are executed in a decentralized way. They take advantage of a declarative language, to specify the statechart model of the composite service, service communities, as containers of alternative services with similar behavior, and also a peer-to-peer execution model. In this execution model the responsibility of coordinating the execution of a composite service is distributed across several peer software components called coordinators. These coordinators are, in fact, attached to each component service, and are in charge of initiating, controlling, and monitoring their associated services, and collaborating with their peers.

Maamar et al. [71] introduce an approach for composite service execution based on three main concepts, i.e., software agents, contexts and conversations. A software agent is a program that acts on behalf of the user, and does conceptually similar to what a coordinator does in [12, 14]. Context is the information relevant to the interactions between the user and the environment. Conversations are the messages passed among the participants to achieve a specific purpose. The authors define different types of agents and contexts.

Yildiz and Godart [108] present a methodology that derives cooperating distrib-

uted processes of a centralized specification with respect to their information flow policies. This methodology is used to provide a systematic approach to manage the information flow between composed services.

Casati and Shan [25] propose eFlow as a platform for specifying, enacting, and monitoring composite services. In eFlow composite services are modeled as business processes enacted by a service process engine. Their platform supports service process specification and management including a simple service composition language, events and exception handling, ACID service level transactions, and security management.

Korhonen et al. [62] present a way to automatically compose web service workflows. The web services workflows are described using a transactional workflow ontology which can be used to describe both component web service workflows and composite web service workflows. They have also implemented a workflow engine that runs the workflow instances.

## 3.4 Ontology Matching

There are different approaches to ontology and schema matching, such as schema and instance-based, element and structure-based, linguistic-based, constraint-based, and cardinality-based. Rahm and Bernstein [88] survey these approaches by describing different domains in which schema matching might be required, along with discussing the *Match* operator which is used to compare two schemas. They also cover and compare some of the schema matching works in the literature on this topic.

In another work, Madhavan et al. [72] combine some of the above matching techniques to find a new match algorithm. Specifically, they take advantage of linguistic-based, element-based and structure-based matching in order to provide a powerful algorithm.

Tamani and Evripidou [96] present an approach to facilitate web service discovery. They add some XML meta-data to web service requests and offers that exposes additional information about the service, such as the identity of web service provider/requester, the purpose of the offer/request and the inputs/outputs involved. Then, a matching process is followed using XPath queries to compare the request against the available offers.

Yeh et al. [107] propose an approach for finding semantic mismatches between

two different representations of the same ontology. Since ontologies are normally expressive enough and representations are usually large and built by different people, multiple encodings of the same knowledge do not necessarily match. Therefore, syntactic and semantic matching become essential to make relations between the two. Since a good matching approach would find mismatches that provide valuable information on comparing two representations, the authors make an effort to address mismatches as well.

## 3.5 Quality of Service Control

Zeng et al. [114] propose a quality-driven approach to optimally and dynamically select component web services in executing a composite service. They introduce a multi-dimensional web service quality model to indicate nonfunctional properties of web services, such as execution price and execution duration. Then, based on this model, they propose a global planning method, formulated as an optimization problem with a linear programming approach, to find the best execution plan for a composite web service. The strength of their approach is that the quality of the composition as a whole is being optimized, instead of optimizing the quality of each component service.

Canfora et al. [23] propose a genetic algorithm-based approach for quality of service aware service composition, which determines a set of concrete services to be bound to abstract services in an orchestration to meet a set of constraints and to optimize a fitness criterion on quality of service attributes. Compared with linear integer programming, genetic algorithms allow dealing with quality of service attributes having nonlinear aggregation functions.

Aggarwal et al. [4] present an approach for achieving constraint driven web service composition by adding an abstract process designer, a constraint analyzer, an optimizer and a binder module. They extend the workflow quality of service model in [24] to allow global optimization and composition of web processes.

Yu and Lin [110] study the web service quality of service constraint issue using a quality of service broker which is responsible for coordinating individual service components to meet the requested quality constraint. The service selection problem is modeled as a multiple choice knapsack problem and its solution and performance is studied using different algorithms.

There are also other publications that try to consider quality of service, very abstractly, in finding composition plans [6, 89].

# Chapter 4

# Behavior Formalization

In this chapter we explain two representation models for the behavior of software components. One of these models is *interface automata* [34] which is a way of describing involved methods/actions in a component along with their temporal order. The next model is *composition algebra* [49] which is an alternative way of representing the similar information about each component. Using appropriate concurrency rules, these two formalisms are capable of representing the behavior of compositions of software components as well.

## 4.1 Interface Automata

Interface automata take advantage of the ordering implied by states and transitions of the automaton model to capture the temporal order of methods and actions in a software component. This model can be used in design and documentation, as well as in validation and model checking [34]. As one of its powerful features, it formally defines the interface automaton resulting from the synchronization of two interface automata. For a component to be represented with an interface automaton, the methods that it provides to the environment and the inputs it receives are modeled as its input actions, while the methods that it invokes (from other interface automata) and the outputs it returns are modeled as its output actions. An example of interface automata is given in Section 4.1.1.

### 4.1.1 Formalism

In this section we represent the formal definition of interface automata and the corresponding concepts related to the scope of this thesis. The materials of this section are mainly taken from de Alfaro and Henzinger [34].

**Definition 4.1** *An interface automaton $P$ is formally defined by the sextuplet $(V_P, V_P^{init}, \mathcal{A}_P^I, \mathcal{A}_P^O, \mathcal{A}_P^H, \mathcal{T}_P)$, where*

- *$V_P$ is a set of states,*
- *$V_P^{init} \subseteq V_P$ is a set of initial states ($V_P^{init} \neq \emptyset$),*
- *$\mathcal{A}_P^I, \mathcal{A}_P^O, \mathcal{A}_P^H$ are mutually disjoint sets of input, output and internal actions ($\mathcal{A}_P = \mathcal{A}_P^I \cup \mathcal{A}_P^O \cup \mathcal{A}_P^H$),*
- *$\mathcal{T}_P \subseteq V_P \times \mathcal{A}_P \times V_P$ is a set of steps.* □

A step $(v, a, v')$, in which $a \in \mathcal{A}_P^I$, is called an *input step*. Alternatively, it is called an *output* or an *internal step* if it belongs to $\mathcal{A}_P^O$ or $\mathcal{A}_P^H$, respectively. An action $a$ is said to be *enabled* at a state $v \in V_P$ if there is a step $(v, a, v') \in \mathcal{T}_P$. Input, output and internal actions enabled at a state $v \in V_P$ are shown by $\mathcal{A}_P^I(v)$, $\mathcal{A}_P^O(v)$ and $\mathcal{A}_P^H(v)$, respectively. The set of all actions enabled at $v$ is shown by $\mathcal{A}_P(v)$, and $\mathcal{A}_P(v) = \mathcal{A}_P^I(v) \cup \mathcal{A}_P^O(v) \cup \mathcal{A}_P^H(v)$. The set $\mathcal{A}_P^I \backslash \mathcal{A}_P^I(v)$ contains *illegal inputs* at $v$. The *size* of an interface automaton $P$ is defined by $|P| = |V_P| + |\mathcal{T}_P|$.

**Example 4.1** *Figure 4.1 depicts the interface automaton of a message transmission component. This component, called* `Comp`*, has a* `msg` *method for sending messages which returns either* `ok` *or* `fail` *as a result. This component performs this functionality through a method* `send` *to a communication channel with* `ack` *and* `nack` *outputs for successful and unsuccessful transmissions, respectively. For this component, we have*

- $V_{Comp} = \{0, 1, 2, 3, 4\}$
- $V_{Comp}^{init} = \{0\}$
- $\mathcal{A}_{Comp}^I = \{\texttt{msg}, \texttt{ack}, \texttt{nack}\}$
- $\mathcal{A}_{Comp}^O = \{\texttt{send}, \texttt{ok}, \texttt{fail}\}$

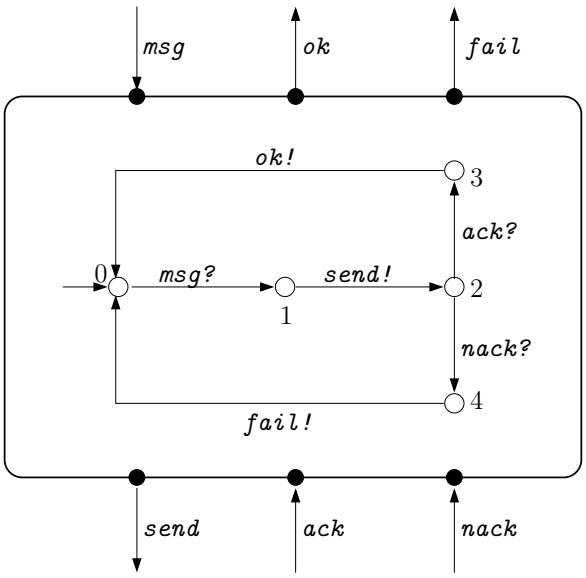



Figure 4.1: Interface automaton of the *Comp* component in Example 4.1.

- $\mathcal{A}^H_{Comp} = \emptyset$
- $\mathcal{T}_{Comp} = \{(0, \texttt{msg}, 1), (1, \texttt{send}, 2), (2, \texttt{ack}, 3), (2, \texttt{nack}, 4), (3, \texttt{ok}, 0), (4, \texttt{fail}, 0)\}$

*The size of this interface automaton is 11.* □

Note that input, output and internal actions in interface automata are affixed with `?`, `!` and `;` symbols, respectively. Interface automata is equipped with a formalism for calculating the composition of two interface automata, especially when they are synchronized on some action. The corresponding operator in interface automata is the *composition* operator shown by $||$ . Before formally defining this operator, we define the concepts of *composability* and *illegal states*.

**Definition 4.2** *Two interface automata $P$ and $Q$ are composable, if and only if $\mathcal{A}^H_P \cap \mathcal{A}_Q = \emptyset$, $\mathcal{A}^H_Q \cap \mathcal{A}_P = \emptyset$ (internal actions of one interface automaton cannot be among the actions of the other automaton), $\mathcal{A}^I_P \cap \mathcal{A}^I_Q = \emptyset$ (their input actions are disjoint), and $\mathcal{A}^O_P \cap \mathcal{A}^O_Q = \emptyset$ (their output actions are disjoint). We let $shared(P, Q) = \mathcal{A}_P \cap \mathcal{A}_Q$. If $P$ and $Q$ are composable, $shared(P, Q) = (\mathcal{A}^O_P \cap \mathcal{A}^I_Q) \cup (\mathcal{A}^I_P \cap \mathcal{A}^O_Q)$.* □

**Definition 4.3** *The illegal states of $P||Q$ is defined by*

$$Illegal(P,Q) = \left\{ (v,u) \in V_P \times V_Q \;\middle|\; \exists a \in shared(P,Q) \,.\, \begin{pmatrix} a \in \mathcal{A}^O_P(v) \wedge a \notin \mathcal{A}^I_Q(u) \\ \vee \\ a \in \mathcal{A}^O_Q(u) \wedge a \notin \mathcal{A}^I_P(v) \end{pmatrix} \right\},$$

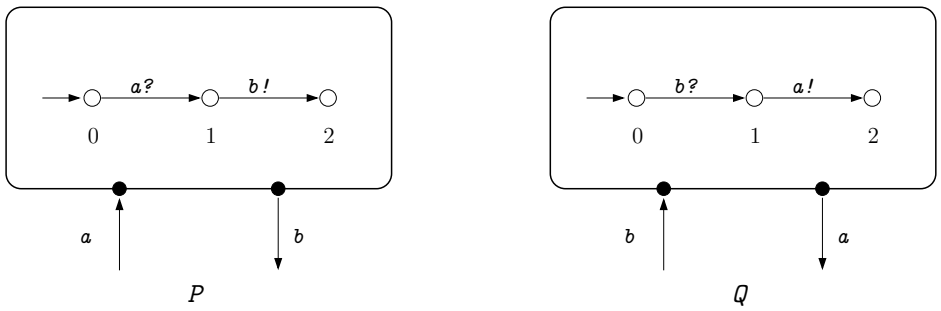



Figure 4.2: Two interface automata $P$ and $Q$ for which $\mathit{Illegal}(P,Q) \neq \emptyset$.

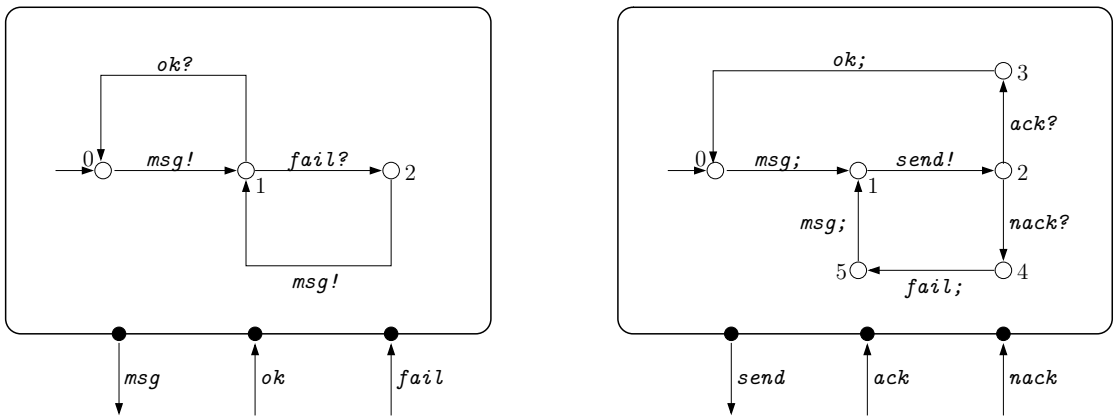
*where $P||Q$ indicates the composition of interface automata $P$ and $Q$.* □

**Example 4.2** *Figure 4.2 depicts two interface automata $P$ and $Q$. According to Definition 4.3, $\mathit{Illegal}(P,Q) \neq \emptyset$. For example state $(0,0)$ is an illegal state, at which one automaton can only receive $a$, and the other can only receive $b$, while both $a$ and $b$ belong to the shared actions of $P$ and $Q$. In case we need to remove all the illegal states, we would have to rename at least one of the shared actions in one automaton. For example, if we rename the output action $b$ in $P$ to $b'$, then there would be no illegal states left.* □

Now, we can formally define the composition of two interface automata.

**Definition 4.4** *The composition $P||Q$ of two composable interface automata $P$ and $Q$, where $\mathit{Illegal}(P,Q) = \emptyset$, is the interface automaton defined by*

- $V_{P||Q} = V_P \times V_Q$
- $V^{init}_{P||Q} = V^{init}_P \times V^{init}_Q$
- $\mathcal{A}^I_{P||Q} = (\mathcal{A}^I_P \cup \mathcal{A}^I_Q) \backslash \mathit{shared}(P,Q)$
- $\mathcal{A}^O_{P||Q} = (\mathcal{A}^O_P \cup \mathcal{A}^O_Q) \backslash \mathit{shared}(P,Q)$
- $\mathcal{A}^H_{P||Q} = \mathcal{A}^H_P \cup \mathcal{A}^H_Q \cup \mathit{shared}(P,Q)$
- $$\begin{aligned} \mathcal{T}_{P||Q} \;=\; & \{((v,u),a,(v',u)) \mid (v,a,v') \in \mathcal{T}_P \wedge a \notin \mathit{shared}(P,Q) \wedge u \in V_Q\} \\ \cup\; & \{((v,u),a,(v,u')) \mid (u,a,u') \in \mathcal{T}_Q \wedge a \notin \mathit{shared}(P,Q) \wedge v \in V_P\} \\ \cup\; & \{((v,u),a,(v',u')) \mid (v,a,v') \in \mathcal{T}_P \wedge (u,a,u') \in \mathcal{T}_Q \wedge a \in \mathit{shared}(P,Q)\} \end{aligned}$$




(a) (b)

Figure 4.3: An example of the interface automata composition. (a) Interface automaton of the component `User` in Example 4.3. (b) Interface automaton of the composition `User||Comp`.

*The states that remain unreachable from $V^{init}_{P||Q}$ are removed from $V_{P||Q}$ after the above calculations. The special case $Illegal(P,Q) = \emptyset$ suffices our needs in this thesis. Discussion on the composition operation in general is beyond the scope of this thesis.* □

**Example 4.3** *Consider another component, called* `User`*, that uses the component* `Comp` *of Example 4.1. This component could be a human user or another component that sends messages through* `Comp` *and receives* `ok` *or* `fail` *responses. We assume that* `User` *sends its message over and over again until it receives an* `ok` *response. The interface automaton of this component is shown in Figure 4.3-(a). To check the composability of components* `Comp` *and* `User`*, we follow Definition 4.2,*

- $\mathcal{A}^H_{Comp} \cap \mathcal{A}_{User} = \emptyset$
- $\mathcal{A}^H_{User} \cap \mathcal{A}_{Comp} = \emptyset$
- $\mathcal{A}^I_{Comp} \cap \mathcal{A}^I_{User} = \emptyset$
- $\mathcal{A}^O_{Comp} \cap \mathcal{A}^O_{User} = \emptyset$,

*which confirms that they are composable. Consequently, $shared(\texttt{Comp}, \texttt{User}) = \{\texttt{msg}, \texttt{ok}, \texttt{fail}\}$, and $Illegal(\texttt{Comp}, \texttt{User}) = \emptyset$. To find the composition of these two interface automata we can follow Definition 4.4, which leads us to the interface automaton of Figure 4.3-(b).* □

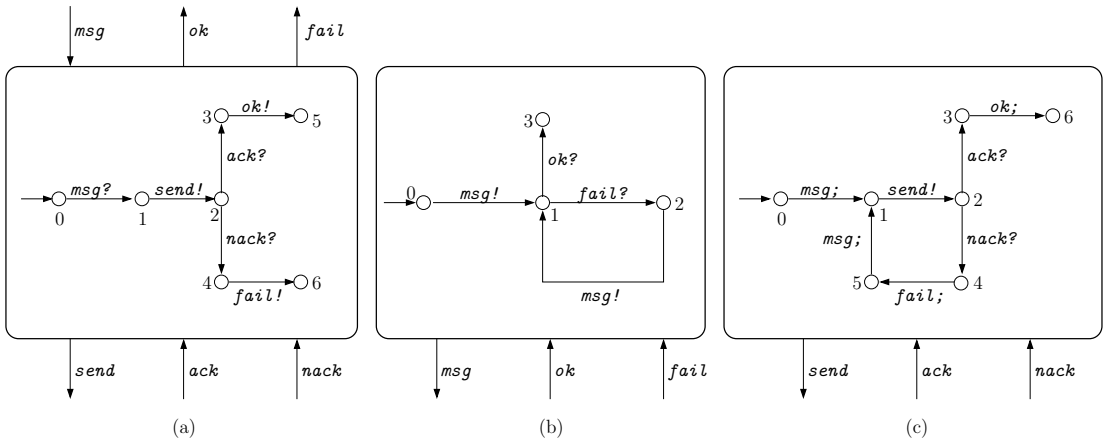



Figure 4.4: Examples of the new representation for interface automata. (a) *Comp*. (b) *User*. (c) *User*||*Comp*.

The composition operator in interface automata is both commutative and associative.

### 4.1.2 Application

Interface automata capture the behavior of components in a general loop, at each execution of which the component is executed once. We can see this in Figures 4.1 and 4.3, where the state 0 is both the initial and the final state. We make a change in this representation by separating the initial and final states. The final states then would be the states from which no step is initiated. The final states of an interface automaton $P$ are shown as $V_P^{fin}$.

This little change would not affect Definition 4.4, and the same formalism works for calculating the composition in this new form of interface automata. The new representations of components *Comp*, *User* and *User*||*Comp* are shown in Figure 4.4. Note that, by definition, an interface automaton could have multiple initial states and final states. However, we prove the following lemma to use a generic type of interface automata for our future discussions.

**Lemma 4.1** *For every interface automaton with multiple initial and final states there is a normalized interface automaton with a single initial state and a single final state which represents the exact same behavior.*

**Proof.** Consider the interface automaton $P$ with $m$ initial states and $n$ final states, i.e., $V_P^{init} = \{i_1, i_2, \cdots, i_m\}$ and $V_P^{fin} = \{f_1, f_2, \cdots, f_n\}$. The normalized equivalent

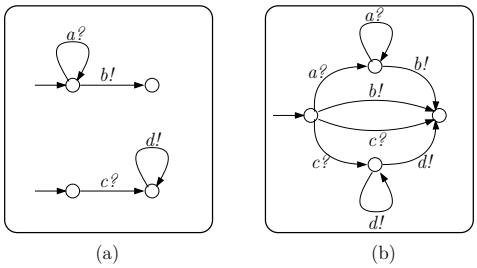
interface automaton $N$ would have a single initial state $i$ and a single final state $f$, with the following attributes:

- $V_N = (V_P \cup \{i, f\}) \backslash (V_P^{init} \cup V_P^{fin})$
- $V_N^{init} = \{i\}$
- $V_N^{fin} = \{f\}$
- $\mathcal{A}_N^I = \mathcal{A}_P^I$
- $\mathcal{A}_N^O = \mathcal{A}_P^O$
- $\mathcal{A}_N^H = \mathcal{A}_P^H$
- $\mathcal{T}_N = \mathcal{T}_P$, except that
  - each $(i_k, a, v) \in \mathcal{T}_P$, in which $1 \leqslant k \leqslant m$, $a \in \mathcal{A}_P$ and $v \in V_P \backslash V_P^{init}$, is replaced by $(i, a, v)$,
  - each $(v, a, f_k) \in \mathcal{T}_P$, in which $1 \leqslant k \leqslant n$, $a \in \mathcal{A}_P$ and $v \in V_P \backslash V_P^{fin}$, is replaced by $(v, a, f)$.[1]

Two interface automata $P$ and $N$ would represent the same behavior, because every execution path in one is an execution path in the other.[2] □

**Example 4.4** *Interface automata of components* `User` *and* `User||Comp` *in Figure 4.4 are normalized. However, the interface automaton of component* `Comp` *is not. If states 5 and 6 of this automaton are merged into a single state, the automaton becomes normalized.* □

[1] In case there is a loop on an initial state or a final state in $P$, the conversion is somewhat different, but can still be simply done. The following is an example of such a case, where the interface automaton in part (b) is the normalized form of the one in part (a).




[2] An *execution path* in an interface automaton $P$ is an alternating sequence $v_0, a_1, v_1, \cdots, a_k, v_k$, in which $v_0 \in V_P^{init}$, $v_k \in V_P^{fin}$, and $(v_{j-1}, a_j, v_j) \in \mathcal{T}_P$ $(1 \leqslant j \leqslant k)$.

The composition operator in interface automata is only one of several ways two components can be composed. In fact, the composition operator indicates how two components can be synchronized on some shared action between them. There are other ways two components can be combined into a more complex or composite component.

- Sequential execution: One component is executed right after the other one finishes its execution.
- Conditional execution: Only one of the two components is executed each time. The choice between the two can be made either deterministically or nondeterministically.
- Parallel execution: Two components are executed without any specific order with respect to each other, and there is no synchronization between them.

Based on the above alternative ways of combining two components, we define the concept of *composition* to refer to all the four types of combining two components, and change the name of interface automata composition operator to the *synchronization* operator and show it by the $\odot$ symbol. We use *sequence* ($\cdot$), *choice* ($\oplus$) and *parallel* ( $||$ ) operators for the above sequential, conditional and parallel execution, respectively. These three composition alternatives can also be modeled using interface automata. We discuss these four operators in the rest of this section.

**Sequence**

To represent the sequential execution of two interface automata $P$ and $Q$, shown as $P \cdot Q$, it suffices to merge the final state of $P$ and the initial state of $Q$. This way, when the execution of $P$ is finished, $Q$ starts its execution. Figure 4.5-(b) shows how to obtain the sequential execution of two interface automata $P$ and $Q$ of Figure 4.5-(a). Formally speaking, the interface automaton of $P \cdot Q$ is defined based on the normalized interface automata of $P$ and $Q$ as follows. We assume that $V_P^{fin} = \{f_P\}$ and $V_Q^{init} = \{i_Q\}$.

- $V_{P \cdot Q} = (V_P \cup V_Q) \backslash \{i_Q\}$
- $V_{P \cdot Q}^{init} = V_P^{init}$
- $V_{P \cdot Q}^{fin} = V_Q^{fin}$

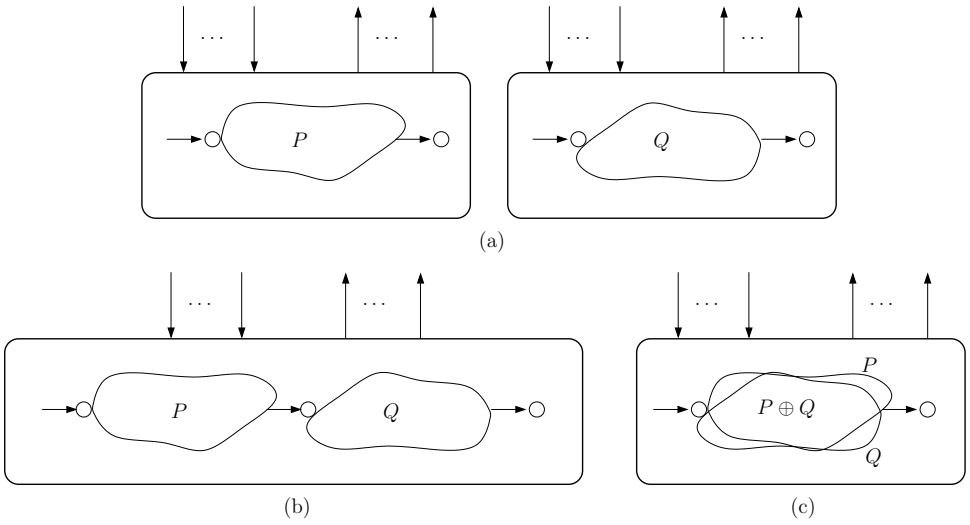



Figure 4.5: How sequence and choice operations are modeled in interface automata. (a) Two normalized interface automata $P$ and $Q$. (b) Interface automata representation for $P \cdot Q$. (c) Interface automata representation for $P \oplus Q$.

- $\mathcal{A}^I_{P \cdot Q} = \mathcal{A}^I_P \cup \mathcal{A}^I_Q$
- $\mathcal{A}^O_{P \cdot Q} = \mathcal{A}^O_P \cup \mathcal{A}^O_Q$
- $\mathcal{A}^H_{P \cdot Q} = \mathcal{A}^H_P \cup \mathcal{A}^H_Q$
- $\mathcal{T}_{P \cdot Q} = \mathcal{T}_P \cup \mathcal{T}_Q$, except that each $(i_Q, a, v) \in \mathcal{T}_Q$, in which $a \in \mathcal{A}_Q$ and $v \in V_Q \backslash \{i_Q\}$, is replaced by $(f_P, a, v)$.

Clearly, the interface automaton of $P \cdot Q$ will be normalized as well. Moreover, we can easily that the sequence operator in interface automata is associative, but not commutative. In other words, $P \cdot Q \neq Q \cdot P$ and $(P \cdot Q) \cdot R = P \cdot (Q \cdot R)$.

### Choice

The interface automaton of the conditional execution of two interface automata $P$ and $Q$, shown as $P \oplus Q$, only combines the initial states of the two. This means that at the initial state the execution could start with $P$ or with $Q$, and once the execution is started, it would stay on the same automaton and would never jump to the other. Figure 4.5-(c) shows how the interface automaton of $P \oplus Q$ would look. The interface automaton of $P \oplus Q$ is formally defined as follows. Again we assume that interface automata of $P$ and $Q$ are normalized, $V_P^{init} = \{i_P\}$, $V_P^{fin} = \{f_P\}$, $V_Q^{init} = \{i_Q\}$ and $V_Q^{fin} = \{f_Q\}$.

- $V_{P \oplus Q} = (V_P \cup V_Q) \backslash \{i_Q, f_Q\}$
- $V^{init}_{P \oplus Q} = V^{init}_P$
- $V^{fin}_{P \oplus Q} = V^{fin}_P$
- $\mathcal{A}^I_{P \oplus Q} = \mathcal{A}^I_P \cup \mathcal{A}^I_Q$
- $\mathcal{A}^O_{P \oplus Q} = \mathcal{A}^O_P \cup \mathcal{A}^O_Q$
- $\mathcal{A}^H_{P \oplus Q} = \mathcal{A}^H_P \cup \mathcal{A}^H_Q$
- $\mathcal{T}_{P \oplus Q} = \mathcal{T}_P \cup \mathcal{T}_Q$, except that
  - each $(i_Q, a, v) \in \mathcal{T}_Q$, in which $a \in \mathcal{A}_Q$ and $v \in V_Q \backslash \{i_Q\}$, is replaced by $(i_P, a, v)$,
  - each $(v, a, f_Q) \in \mathcal{T}_Q$, in which $a \in \mathcal{A}_Q$ and $v \in V_Q \backslash \{f_Q\}$, is replaced by $(v, a, f_P)$.

Consequently, the resulting interface automaton will be normalized as well. Also, this definition of the choice operator in interface automata implies that it is both commutative and associative. Therefore, $P \oplus Q = Q \oplus P$ and $(P \oplus Q) \oplus R = P \oplus (Q \oplus R)$.

### Parallel

Finding the interface automaton of the parallel execution of two interface automata $P$ and $Q$ is more complex. The thing to do is walk through the automaton of each component at the same time having the option to choose the next action from each of the two. It is like running the two component at the same time in an unordered fashion. Figure 4.6 shows how the parallel execution of two simple components is modeled in interface automata. This interface automaton is obtained by applying a formalism rather similar to Definition 4.4. Specifically,

- $V_{P \parallel Q} = V_P \times V_Q$
- $V^{init}_{P \parallel Q} = V^{init}_P \times V^{init}_Q$
- $V^{fin}_{P \parallel Q} = V^{fin}_P \times V^{fin}_Q$
- $\mathcal{A}^I_{P \parallel Q} = \mathcal{A}^I_P \cup \mathcal{A}^I_Q$

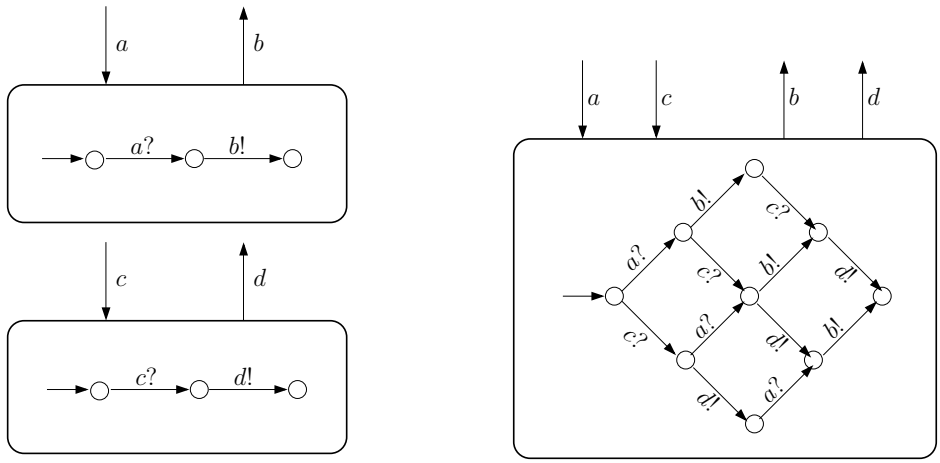



Figure 4.6: An example of a parallel execution in an interface automaton. (a) Simple interface automaton for components $P$ and $Q$. (b) The interface automaton of $P \parallel Q$.

- $\mathcal{A}^O_{P \parallel Q} = \mathcal{A}^O_P \cup \mathcal{A}^O_Q$
- $\mathcal{A}^H_{P \parallel Q} = \mathcal{A}^H_P \cup \mathcal{A}^H_Q$
- $$\begin{aligned}\mathcal{T}_{P \parallel Q} \;=\; & \{((v,u),a,(v',u)) \mid (v,a,v') \in \mathcal{T}_P \wedge u \in V_Q\} \\ \cup\; & \{((v,u),a,(v,u')) \mid (u,a,u') \in \mathcal{T}_Q \wedge v \in V_P\}\end{aligned}$$

According to this formalism, if interface automata of $P$ and $Q$ is normalized, so will be the interface automaton of $P \parallel Q$. Moreover, since the parallel execution is a special case of the original interface automata composition, in which there is no shared action between the two automata, the parallel operator inherits its commutativity and associativity. In other words, $P \parallel Q = Q \parallel P$ and $(P \parallel Q) \parallel R = P \parallel (Q \parallel R)$.

### Synchronization

We discussed this operator, as the composition operator, in Section 4.1.1. We mentioned that it works based on the notion of shared actions between two components, i.e., actions that are inputs in one and outputs in the other. We call this type of shared actions *complementary actions*.

**Definition 4.5** Complementary actions *of two components $P$ and $Q$, defined as complementary$(P,Q)$, are defined as the set of actions which are inputs in one and*

*outputs in the other. Considering their interface automata, complementary actions of $P$ and $Q$ would be* $(\mathcal{A}_P^O \cap \mathcal{A}_Q^I) \cup (\mathcal{A}_Q^O \cap \mathcal{A}_P^I)$. □

The interface automata synchronization based on complementary actions is both commutative and associative [34].

Here, we add an alternative to shared actions, i.e., *shared inputs* [33]. We claim that two components can be synchronized on their shared inputs as well, because when they are expecting an input of the same type, one instance of that data type could be given to both, and there is no need to prepare two instances, i.e., one instance for each component.

**Definition 4.6** Shared inputs *of two components $P$ and $Q$, sharedinputs$(P, Q)$, are the set of input actions common to the two of them. Considering their interface automata, the set of shared actions of $P$ and $Q$ would be* $(\mathcal{A}_P^I \cap \mathcal{A}_Q^I)$. □

As a result, shared actions of two components $P$ and $Q$, *shared*$(P, Q)$, would be the union of their complementary actions and shared inputs. Formally speaking, *shared*$(P, Q)$ = *complementary*$(P, Q)$ ∪ *sharedinputs*$(P, Q)$. Based on this new definition, the synchronization of two interface automata $P$ and $Q$, when *Illegal*$(P, Q) = \emptyset$, is defined as follows.

- $V_{P \odot Q} = V_P \times V_Q$
- $V_{P \odot Q}^{init} = V_P^{init} \times V_Q^{init}$
- $\mathcal{A}_{P \odot Q}^I = (\mathcal{A}_P^I \cup \mathcal{A}_Q^I) \backslash complementary(P, Q)$
- $\mathcal{A}_{P \odot Q}^O = (\mathcal{A}_P^O \cup \mathcal{A}_Q^O) \backslash complementary(P, Q)$
- $\mathcal{A}_{P \odot Q}^H = \mathcal{A}_P^H \cup \mathcal{A}_Q^H \cup complementary(P, Q)$
- $$\begin{aligned} \mathcal{T}_{P \odot Q} \;=\; & \{((v,u), a, (v',u)) \mid (v, a, v') \in \mathcal{T}_P \wedge a \notin shared(P,Q) \wedge u \in V_Q\} \\ \cup\; & \{((v,u), a, (v,u')) \mid (u, a, u') \in \mathcal{T}_Q \wedge a \notin shared(P,Q) \wedge v \in V_P\} \\ \cup\; & \{((v,u), a, (v',u')) \mid (v, a, v') \in \mathcal{T}_P \wedge (u, a, u') \in \mathcal{T}_Q \wedge a \in shared(P,Q)\} \end{aligned}$$

The interface automata synchronization based on shared inputs has been discussed by de Alfaro et al. [33], and addressed as a commutative and associative operator.

Now that we are familiar with the interface automata operators, we define the concepts of equality and equivalence for interface automata.

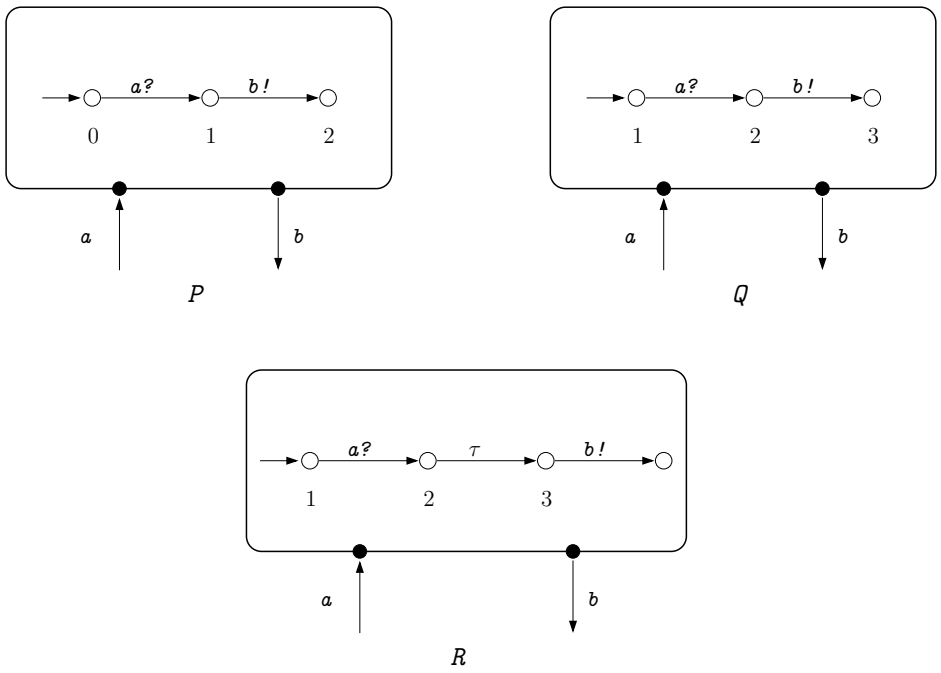


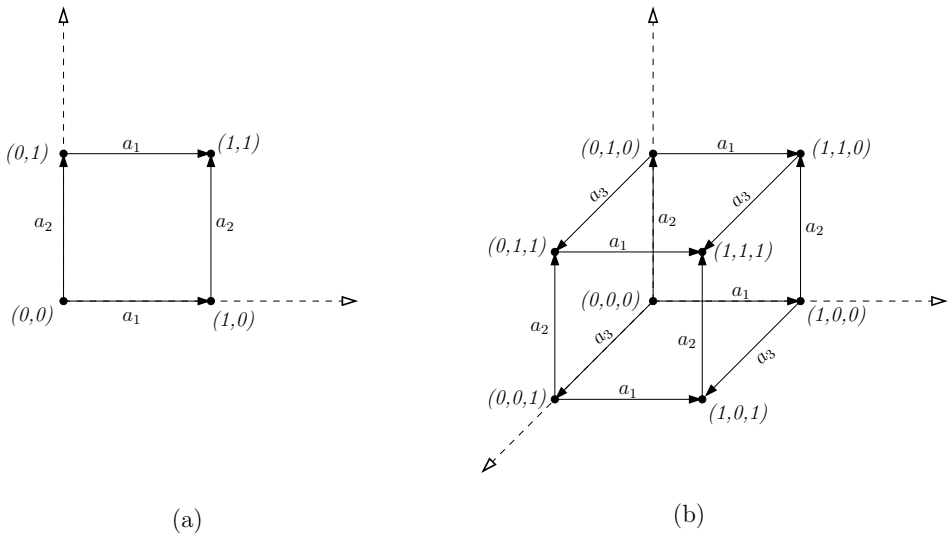

Figure 4.7: Examples of equal ($P$ and $Q$) and equivalent ($P$ and $R$) interface automata.

**Definition 4.7** *Two interface automata $P$ and $Q$ are* equal *($P = Q$), if and only if there are one-to-one correspondences between elements of $V_P$ and $V_Q$, $V_P^{init}$ and $V_Q^{init}$, $\mathcal{T}_P$ and $\mathcal{T}_Q$, and moreover $\mathcal{A}_P^I = \mathcal{A}_Q^I$, $\mathcal{A}_P^O = \mathcal{A}_Q^O$, and $\mathcal{A}_P^H = \mathcal{A}_Q^H$. They are* equivalent *($P \equiv Q$), if and only if they represent the same externally visible behavior. We define $\partial$-*trace equivalence [100] *as the equivalence relation for interface automata.* □

For $\partial$-trace equivalence, two processes are equivalent if each execution path in one has a similar execution path in the other. Note that all internal actions in an interface automaton are invisible to the outside viewer, and would be equivalent to the silent action $\tau$.

**Example 4.5** *Figure 4.7 depicts three interface automata $P$, $Q$, and $R$. According to Definition 4.7, $P = Q$ and $P \equiv R$.* □

The modified version of interface automata we studied in this section can be used to represent the behavior of components and their compositions. We see examples of its direct application in Chapter 5.




(a) (b)

Figure 4.8: Unordered execution of actions can be seen as moving from the point $(0, 0, \cdots, 0)$ to the point $(1, 1, \cdots, 1)$ in the $n$-dimensional space. Figures (a) and (b) represent the cases where two and three actions are involved, respectively.

## 4.2 Composition Algebra

To formally represent stateless components, using interface automata is not the best choice. The simple reason is that when the number of actions grows it is not easy enough to represent and understand an interface automaton. For example, consider a stateless component which receives 5 inputs in parallel and returns only one output. An interface automaton would need 32 states just to represent the unordered execution of the inputs of this component. The following lemma formally describes this observation.

**Lemma 4.2** *In order to represent the unordered execution of $n$ actions in an interface automaton $2^n$ states and $n \times 2^{n-1}$ steps are required.*

**Proof.** Assume that the interface automaton $P$ represents the unordered execution of $n$ actions $a_1$, $a_2$, ..., $a_n$. This unordered execution could be seen as moving from the point $(0, 0, \cdots, 0)$ to the point $(1, 1, \cdots, 1)$ in the $n$-dimensional space (see Figure 4.8), where each dimension corresponds to one of the actions involved. Therefore, each move in the $k$-th dimension would represent executing action $a_k$. We can see that for such an unordered execution, there is a one-to-one correspondence between the relative interface automaton nodes and all the points $\{(x_1, x_2, \cdots, x_n) \mid x_i \in \{0, 1\}\ (1 \leqslant i \leqslant n)\}$ in this $n$-dimensional space, where

the points $(0, 0, \cdots, 0)$ and $(1, 1, \cdots, 1)$ correspond to the initial and final states of the automaton, respectively. For instance, arriving at point $(0, 1, 0, \cdots, 0, 1)$ indicates that actions $a_2$ and $a_n$ have been executed and all other actions are yet to be executed. This is enough to show that the number of required nodes in $P$ is $2^n$.

Similarly, we notice that there is also a one-to-one correspondence between the steps of $P$ and the edges in this $n$-dimensional space. Each point $(x_1, x_2, \cdots, x_n)$, $x_i \in \{0, 1\}$, is connected to $n$ other point through $n$ edges, and therefore, the total number of edges in the $n$-dimensional space, and hence in the interface automaton $P$, would be $\frac{n \times 2^n}{2} = n \times 2^{n-1}$. Note that the denominator 2 is used because each edge is shared between two nodes. □

Due to the complexities that come with representing behaviors in interface automata, we use an algebraic model as an alternative and easier representation for the same information. This model, which is called *composition algebra*, gives an algebraic representation to interface automata operators and axioms. At the same time, it can be compared to well-known process algebras, such as CSP (Communicating Sequential Processes) [51] and CCS (Calculus of Communicating Systems) [77].

### 4.2.1 Formalism

Similar to the well-known process algebras [51, 77] proposed for behavior modeling, the composition algebra also captures inputs, outputs, and the temporal order of actions in a process in all its possible executions, according to its underlying interface automata representation.

We assume that a process consists of one or more actions (input or output) or simpler processes which are executed as parts of a specific workflow. In composition algebra process names start with an uppercase and action names start with a lowercase letter. Each single action could also be seen as a process with only one action. Each action is considered to be atomic and indivisible, and therefore, would be located at the lowest levels of this hierarchy as a leaf node. In order to distinguish between input and output actions, outputs in composition algebra are identified with a macron.

**Example 4.6** *Assume a component* `CANAreaCodes` *which receives a city and a province name in Canada and returns the corresponding area code. Then* `city`, `province` *and* $\overline{\texttt{areaCode}}$ *would be three actions of this process. The first two actions are input actions and the last one is an output.* □

Composition algebra captures from each process only actions visible to the outside world. For example, the above *CANAreaCodes* component, after receiving the two inputs, probably contacts its underlying database to find the appropriate area code. However, this database connection cannot be seen by the users and, therefore, is not of our interest. One could say that the composition algebra models only the external behavior of processes. Based on this observation, we use the *silent* action $\tau$ [77] to represent one or more back to back internal actions invisible to the outside world. In fact, to the outside viewer there is no difference between occurring no action and occurring one or more consecutive internal actions, because the difference cannot be observed. Therefore, we can represent them all by the silent action. Unlike interface automata, composition algebra treats all the internal actions the same way by assigning them the unique label $\tau$.

The concept of *equivalence* in composition algebra refers to behavioral equivalence. Two processes are equivalent in composition algebra if the external behaviors represented by their underlying interface automata is the same. As mentioned earlier, we use the concept of *∂-trace equivalence* [100] for this purpose. We designate the equivalence of two processes $P$ and $Q$ by $P \equiv Q$. We use the familiar symbol $=$ to indicate that two algebraic expressions are exactly the same. Therefore, $P = Q$ implies $P \equiv Q$, while the reverse is not necessarily true.

Now that we are familiar with actions and processes as the building blocks of the composition algebra and also the idea behind process equivalence, we introduce the algebraic operators. These operators are semantically the same as those of interface automata. As we mentioned earlier, since each action can be seen as a process, these operators also apply to actions. Below, the composition algebraic operators are introduced along with a simple comparison to their counterparts in CSP and CCS.

### Sequence

The sequence operator, represented by $\cdot$, is used to show that a process is executed right after another process is terminated. In other words, $P \cdot Q$ specifies a process in which the process $P$ is first executed and after its execution, the process $Q$ is executed. The same description is true for actions. For example, $P = a \cdot \overline{b}$ means that during the process $P$, an input of type $a$ is received and, after that, an output of type $b$ is returned. In fact, it is the composition algebraic representation of the interface automaton $P$ in Figure 4.2. This sequence operator is more general

compared to the sequence operators of CSP and CCS, which only allow a process to follow an atomic action.

### Choice

The choice or conditional operator, represented by $\oplus$, is used to represent different paths of execution when the control flow is determined based on a specific condition or decision, or even nondeterministically. Therefore, $P \oplus Q$ represents a process that behaves like either $P$ or $Q$ (not both) during each execution. This operator can be used on actions too. As an example, the process $P = (a \oplus b) \cdot \overline{c}$ receives an input of type $a$ or $b$ (not both) and returns an output of type $c$. As another example, the process $msg{\cdot}\overline{send}{\cdot}((ack{\cdot}\overline{ok}) \oplus (nack{\cdot}\overline{fail}))$ is the composition algebraic representation of the interface automaton of Figure 4.4-(a). The similar choice operator exists in CCS, while CSP is equipped with three different choice operators.

### Parallel

In composition algebra, parallel or unordered execution of two processes, represented by $||$ , means that there is no synchronization point between them during their execution. So one would expect that each process performs its workflow independently. Although there is a conceptual difference between the unordered execution and the parallel execution, we believe that the importance of the parallel execution is to be experienced in practice, and on the paper it is quite similar to the execution of two processes without any specific order. Therefore, we simulate the parallel execution of two processes by interleaving their workflows. It is like assuming that processes, as collections of atomic actions, are executed by a single processor. For example, the process $(a \cdot \overline{b}) \;||\; (c \cdot \overline{d})$ is an alternative representation for the interface automaton of Figure 4.6-(b).

The parallel operator is not a primary operator in our process algebra as it can be simulated using a combination of sequence and choice operators. This is how the parallel operator is also defined in the main process algebraic references [9, 51, 77]. In general, the simulated form of a process containing some parallel operators can be calculated using the following equations [9, 51]:

- $P \;||\; \tau \equiv \tau \;||\; P \equiv P$ (4.1)
- $P \;||\; (Q \oplus R) \equiv (P \;||\; Q) \oplus (P \;||\; R)$, (4.2)
  $(Q \oplus R) \;||\; P \equiv (Q \;||\; P) \oplus (R \;||\; P)$ (4.3)

- $(a \cdot P) \mid\mid (b \cdot Q) \equiv (a \cdot (P \mid\mid (b \cdot Q))) \oplus (b \cdot ((a \cdot P) \mid\mid Q))$ (4.4)

There is a composition operator in CCS and a parallel operator in CSP which partly behave like the above parallel operator. CSP also contains an interleaving operator with a similar semantics.

### Synchronization

As the last type of composition operators, represented by $\odot$, the synchronization operator specifies a situation in which two processes synchronize their execution because they have specific similarities between one or more of their actions. There are two types of synchronization:

- Input-output consumption: When the output of one process is used by another process as an input a handshake between the two processes happens, which makes them synchronized on those input-output actions. We refer to these input and output actions as *complementary actions*. As a result, the two complementary actions become invisible to the outside world. That is why we represent their synchronization by the silent action ($\tau$). For instance, for two processes $P = a \cdot \overline{b}$ and $Q = b \cdot \overline{c}$, we would have $P \odot Q \equiv a \cdot \tau \cdot \overline{c} \equiv a \cdot \overline{c}$. We described in Section 4.1.1 how to find the result of synchronizing two interface automata based on their complementary actions. We follow those formulas in calculating the result of such a synchronization in the composition algebra as well.

  There might be more than one pair of complementary actions in a synchronization. In this situation, the important requirement is that complementary actions must occur in the same order in both processes, if there is any specific temporal order involved. For example, $P = a \cdot \overline{b} \cdot \overline{c}$ and $Q = c \cdot b \cdot \overline{d}$ cannot be synchronized because of the different order of actions $b$ and $c$ in them.[3]

- Shared inputs: When two processes expect the same input action, they can be synchronized on that action. This synchronization type has been addressed by others [33, 51]. The reason this synchronization is valid is that if two processes $P$ and $Q$ need an input $a$, only a single instance of $a$ can be provided and fed to both processes. The input action $a$ in this example is a synchronizer, but

[3]In fact, this synchronization is not possible because their underlying interface automata composition comes with some illegal states. According to Definition 4.4, the synchronization of $P$ and $Q$ is undefined whenever $\mathit{Illegal}(P, Q) \neq \emptyset$.

it is different from the previous case. Here, the result of synchronizing two same input actions is one input action of the same type; while in the previous case the result is the silent action. As an example, if $P = a \cdot \overline{b}$ and $Q = a \cdot \overline{c}$, then $P \odot Q \equiv a \cdot (\overline{b} \mid\mid \overline{c})$. We explained in Section 4.1.2 how the result of such a synchronization can be calculated.

In the case where there is no synchronizer action in the two processes, this operator acts as a parallel operator, because when there is no synchronization between two running processes, they may be executed in any possible order.

The composition operator in CCS and the parallel operator in CSP merge the semantics of both above parallel and synchronization operators in one operator. The composition operator in CCS does the synchronization only when complementary actions exist. On the other hand, in CSP the parallel operator synchronizes two processes only when there is a pair of complementary actions involved. Therefore, we could claim that the synchronization operator in composition algebra is more general.

We assume that complementary actions and shared inputs have the same name. Therefore, we use a renaming expression for the case the two *synchronizing actions*, i.e., complementary actions or shared inputs, have different names. The renaming expression $P[a'/a, b'/b, \cdots]$ means that action names $a$, $b$, ... in component $P$ are substituted by action names $a'$, $b'$, ..., respectively. CCS and CSP are equipped with a similar expression too.

In some situations, we might need to hide some of the actions in an algebraic expression. We can use the *hiding* operation of CSP and CCS for this purpose. The expression $P \backslash S$, in which $S$ is a set of action names, specifies a process $P$ in which all its actions which are in $S$ are removed from its expression.

Regarding the binding power of the above operators, the synchronization operator is the strongest; then sequence, parallel and choice in that order. For example, $P \mid\mid Q \odot R \oplus S \cdot T \equiv (P \mid\mid (Q \odot R)) \oplus (S \cdot T)$. Finally, we assume that there is no direct or indirect recursion allowed in process expressions. This restriction avoids processes to be defined based on themselves.

We can easily see that each interface automaton can be converted to an equivalent composition algebraic expression; and vice versa. Note that sequence and choice are the basic operators in both models. By decomposing an interface automaton into its separate execution scenarios, we can simply find a corresponding composition algebraic expression for each scenario. Then by conditionally composing those scenarios we would find an equivalent algebraic expression for the original

interface automaton. The same can be done in the opposite direction to find an equivalent interface automaton for a given composition algebraic expression.

### 4.2.2 Algebraic Rules

There are several rules related to the operators of composition algebra, which can be concluded from the interface automata operators introduced earlier in this chapter.

- Closure: The composition of two components using any of the above operators is also a component.
- Commutativity: According to their definition, choice, parallel and synchronization operators are commutative [34, 51, 77]; i.e.,
  - $P \cdot Q \not\equiv Q \cdot P \quad (P \not\equiv Q \wedge P \not\equiv \tau \wedge Q \not\equiv \tau)$ (4.5)
  - $P \oplus Q \equiv Q \oplus P$ (4.6)
  - $P \mid\mid Q \equiv Q \mid\mid P$ (4.7)
  - $P \odot Q \equiv Q \odot P$ (4.8)
- Associativity: All algebraic operators except the synchronization are associative [34, 51, 77]; i.e.,
  - $P \cdot (Q \cdot R) \equiv (P \cdot Q) \cdot R \equiv P \cdot Q \cdot R$ (4.9)
  - $P \oplus (Q \oplus R) \equiv (P \oplus Q) \oplus R \equiv P \oplus Q \oplus R$ (4.10)
  - $P \mid\mid (Q \mid\mid R) \equiv (P \mid\mid Q) \mid\mid R \equiv P \mid\mid Q \mid\mid R$ (4.11)
  - $P \odot (Q \odot R) \not\equiv (P \odot Q) \odot R \quad (\exists P, Q, R)$ (4.12)

  Regarding the associativity for the synchronization operator, when there is no shared input involved, the synchronization operator is associative [34]. Similarly, when there is no pair of complementary actions involved, the synchronization is again associative [51]. The non-associativity arises when both shared inputs and complementary actions exist in a process. For example, we could easily see that the two processes $(a \odot a) \odot \overline{a}$ and $a \odot (a \odot \overline{a})$ are not equivalent; as the first one has no external behavior ($\tau$), while the second one is equivalent to $a$. Therefore, excluding this case, we could consider the synchronization operator to be associative as well.
- Identity: The silent process is the identity element for sequence, parallel and synchronization operators; i.e.,

- $P \cdot \tau \equiv \tau \cdot P \equiv P$ (4.13)
- $P \oplus \tau \equiv \tau \oplus P \not\equiv P \quad (P \not\equiv \tau)$ (4.14)
- $P \mid\mid \tau \equiv \tau \mid\mid P \equiv P$ (4.1)
- $P \odot \tau \equiv \tau \odot P \equiv P$ (4.15)

The expression $P \oplus \tau$ refers to a process that, in each execution, either acts like $P$ or does not have any externally visible behavior. Apparently, it is not equivalent to the process $P$ [9].

- Inverse: Each process has a single inverse process under the synchronization operator. The actions and their order in both processes are the same, except that each input in one is an output in the other; and vice versa. The silent process is its own inverse. The inverse process is identified by a macron; and
  - $P \odot \overline{P} \equiv \overline{P} \odot P \equiv \tau$ (4.16)
- Distributivity: Not every operator is distributive over the other ones. According to the notion of $\partial$-trace equivalence only the followings hold [9]
  - $P \cdot (Q \oplus R) \equiv (P \cdot Q) \oplus (P \cdot R)$, (4.17)
    $(Q \oplus R) \cdot P \equiv (Q \cdot P) \oplus (R \cdot P)$ (4.18)
  - $P \mid\mid (Q \oplus R) \equiv (P \mid\mid Q) \oplus (P \mid\mid R)$, (4.2)
    $(Q \oplus R) \mid\mid P \equiv (Q \mid\mid P) \oplus (R \mid\mid P)$ (4.3)
  - $P \odot (Q \oplus R) \equiv (P \odot Q) \oplus (P \odot R)$, (4.19)
    $(Q \oplus R) \odot P \equiv (Q \odot P) \oplus (R \odot P)$ (4.20)
- Synchronization: Consider two processes $P = P_1 \cdot x \cdot P_2$ and $Q = Q_1 \cdot y \cdot Q_2$, in which $x$ and $y$ are the first synchronizing actions appearing in the two processes. If $x$ and $y$ are complementary actions ($x = \overline{y}$) then
  - $P \odot Q \equiv (P_1 \mid\mid Q_1) \cdot (P_2 \odot Q_2)$. (4.21)

  Similarly, if $x$ and $y$ are shared inputs ($x = y$, and they are both input actions) then
  - $P \odot Q \equiv (P_1 \mid\mid Q_1) \cdot x \cdot (P_2 \odot Q_2)$. (4.22)

  Processes $P_1$ and $Q_1$ are executed in parallel because we assumed that $x$ and $y$ are the first synchronizing actions in $P$ and $Q$. The same equations are used to calculate $P_2 \odot Q_2$ if there is some synchronizing actions in $P_2$ and $Q_2$; otherwise $P_2$ and $Q_2$ are executed in parallel too.

The reason the above equivalences hold is quite simple. If two processes can be synchronized, they would have some synchronizing actions. For every one of these actions, the parts before the action in two processes and also the parts after are executed without any specific relative order; i.e., in parallel.

**Some Results**

Based on the definition of the composition algebraic operations and rules, we can conclude a number of corollaries:

- $P \cdot Q \equiv P \cdot R \implies Q \equiv R$ (4.23)
- $P \oplus P \equiv P$ (4.24)
- $P \oplus Q \equiv P \oplus R \implies Q \equiv R$ (4.25)
- $P \mid\mid Q \equiv P \mid\mid R \implies Q \equiv R$ (4.26)
- $P \odot Q \equiv \tau \implies Q \equiv \overline{P}$ (4.27)
- $\overline{\overline{P}} = P$ (4.28)

## 4.3 Summary

In this section we reviewed the interface automata as a behavioral model for software components. Then we extended this model so that it captures more composition types. We also discussed why interface automata are not the best choice for representing the behavior, especially when the behavior of components become more complex. This why we introduced composition algebra, as a n algebraic model for the behavior of components and their composition. This algebra is completely based on interface automata and follows the same axioms and rules in composing the behaviors of components. It could be considered as an alternative way of representing interface automata. In the next chapters we explain how this algebra helps in validating the results returned by our automatic composition planning approaches.

# Chapter 5

# A Graph-Based Approach to Component Composition

In this chapter we explain how graphs can be used to address and solve the composition planning problem. We use a graph structure called *dependency graph* to model the repository of available components. Then we use graph search algorithms to find solutions for a given component request. We start this chapter by a simple version of the composition planning problem and provide a solution for it. For this simple version, we use interface automata to model the behavior of components and their composition. We discuss the shortcomings of the given solution, refine the problem, and improve the solution to resolve those shortcomings. We use composition algebra in this second part for representing the behavior of components and their composition.

## 5.1 The Simple Composition Planning Problem

The following is the composition planning problem that is studied in the first part of this chapter:

**Problem 5.1**

**Given:**

- *a repository $R$ of components that, in each execution scenario*[1]*, receive only one input and then return only one output.*

[1]Definition 1.2 describes the concept of an execution scenario.

- *a request $G$ which is a component defined by the execution scenarios it provides.*

**Goal:**

- *Yes/No, based on whether there is a composition of some repository components that behaves like $G$.*
- *an appropriate composition, if the above answer is Yes.* □

Before discussing the solution to the simple version of the composition planning problem, we need to formalize the behavior of components. This formalization, which helps us understand the behavior exposed by compositions of repository components, is provided using interface automata. We saw in Chapter 4 how interface automata capture the behavior of software components, and also how they represent different types of behavioral composition. In this section we use this feature of interface automata to formally illustrate and validate component compositions that are found using the composition approach presented later in the section.

### 5.1.1 Required Formalization

To solve this version of the problem, each repository component $C$ is defined by three attributes:

- Inputs ($I_C$): the set of inputs received by $C$.
- Outputs ($O_C$): the set of outputs returned by $C$.
- Dependencies ($\kappa_C$): dependencies that hold between the inputs in $I_C$ and the outputs in $O_C$ (*input-output dependencies*), or refer to temporal order of inputs and outputs in each execution of $C$ (*temporal dependencies*).

The first two attributes are quite straightforward. Any input that the component can receive will be part of its input set, and any output that it can produce will be part of its output set. The two types of dependencies are formally defined as follows.

**Definition 5.1** *If a component receives the set of inputs $I$ and returns the set of outputs $O$ in one of its execution scenarios, we say that, by default, all the outputs in $O$ are dependent on all the inputs in $I$. This is called an* input-output dependency *and is shown by $I \rightarrow O$. If the default case does not apply to a specific component, the corresponding input-output dependencies must be given explicitly.* □

In other words, each component acts as a function which takes some inputs and returns some outputs based on those given inputs. The symbol $\kappa$ is taken from [35], where it is referred to as "the I/O dependency relation" representing the similar concept.

A component might have different execution scenarios, but in each such scenario the component uses the given inputs to generate the corresponding outputs. Therefore, there is a dependency relation between the consumed inputs and the produced outputs in each scenario. As mentioned above, each input-output dependency is of the form $I \rightarrow O$, in which $I$ is the set of inputs and $O$ is the set of outputs. The set $\kappa_C$ then includes subsets $\kappa_C^1$, $\kappa_C^2$, ..., $\kappa_C^n$, where $n$ is the number of execution scenarios in $C$, and $\kappa_C^i$ represents the set of dependencies for the $i$-th execution scenario.

**Example 5.1** *Component* `TemperatureConvertor` *converts Celsius and Fahrenheit temperature values (*`temperature_C` *and* `temperature_F`*, respectively) into each other. The triplet* $(I_{TemperatureConvertor}, O_{TemperatureConvertor}, \kappa_{TemperatureConvertor})$ *describes this component, and*

- $I_{TemperatureConvertor} = \{$`temperature_C`, `temperature_F`$\}$,
- $O_{TemperatureConvertor} = \{$`temperature_F`, `temperature_C`$\}$,
- $\kappa_{TemperatureConvertor} = \{\kappa^1_{TemperatureConvertor}, \kappa^2_{TemperatureConvertor}\}$, *where*
  - $\kappa^1_{TemperatureConvertor} = \{$`temperature_C` $\rightarrow$ `temperature_F`$\}$[2],
  - $\kappa^2_{TemperatureConvertor} = \{$`temperature_F` $\rightarrow$ `temperature_C`$\}$.

*This component has two different execution scenarios. In one, it receives a Celsius temperature and returns a Fahrenheit temperature, and in the other it does the opposite.* □

**Definition 5.2** *A component may receive/produce its inputs/outputs in a predefined order. If these temporal orders are not captured by the input-output dependencies, they must be provided explicitly. In this case, they are called* temporal dependencies *and are shown similar to input-output dependencies. Often, a temporal dependency holds among some inputs or some outputs to imply the temporal order based on which inputs are given to the component or outputs are returned by the component.* □

[2]Although, by definition, each side of a dependency is a set, the brackets are removed when no ambiguity arises.

The three attributes of the components can be used to obtain their corresponding interface automaton. If $\mathcal{C}$ is the interface automaton of the component $C$, then $\mathcal{A}^I_{\mathcal{C}} = I_C$ and $\mathcal{A}^O_{\mathcal{C}} = O_C$. Moreover, the internal automaton of $\mathcal{C}$ can be found through $\kappa_C$. Note that the assumption we make here is that every input/output of the component $C$ would appear in $\kappa_C$, and further, each dependency in $\kappa_C$ appears explicitly in $\mathcal{C}$. In order to satisfy the latter assumption, we allow temporal dependencies to be added to the dependency set $\kappa_C$. The following example clarifies this assumption.

**Example 5.2** *Component* `WeatherForecast` *provides the weather information for Canadian cities. It provides two basic functionalities. In one, it receives a city name and returns the current temperature for that city. In the other, it receives a city name and a date and returns the average temperature of that city on the given date. Then, we would have the following attributes for this component:*

- $I_{\texttt{WeatherForecast}} = \{\texttt{city\_CAN}, \texttt{date}\}$,
- $O_{\texttt{WeatherForecast}} = \{\texttt{temperature\_C}\}$,
- $\kappa_{\texttt{WeatherForecast}} = \{\kappa^1_{\texttt{WeatherForecast}}, \kappa^2_{\texttt{WeatherForecast}}\}$, *where*
  - $\kappa^1_{\texttt{WeatherForecast}} = \{\texttt{city\_CAN} \rightarrow \texttt{temperature\_C}\}$,
  - $\kappa^2_{\texttt{WeatherForecast}} = \{\{\texttt{city\_CAN}, \texttt{date}\} \rightarrow \texttt{temperature\_C}\}$.

*The first dependency set is straightforward. In the second set we see that the output* `temperature_C` *is explicitly dependent on two inputs* `city_CAN` *and* `date`*. To represent this dependency in an interface automaton, the only way is to assume that the two inputs can be given in any order. That is why the corresponding interface automaton for this component is the one given in Figure 5.1-(a). However, if the* `WeatherForecast` *component took the input* `city_CAN` *before the input* `date`*,* $\kappa^2_{\texttt{WeatherForecast}}$ *would have been somewhat different, i.e.,* $\kappa^2_{\texttt{WeatherForecast}} = \{\{\texttt{city\_CAN}, \texttt{date}\} \rightarrow \texttt{temperature\_C}, \texttt{city\_CAN} \rightarrow \texttt{date}\}$*, where the first dependency is an input-output dependency and the second one is a temporal dependency. In this case, the interface automaton of Figure 5.1-(b) would have represented the underlying behavior.* □

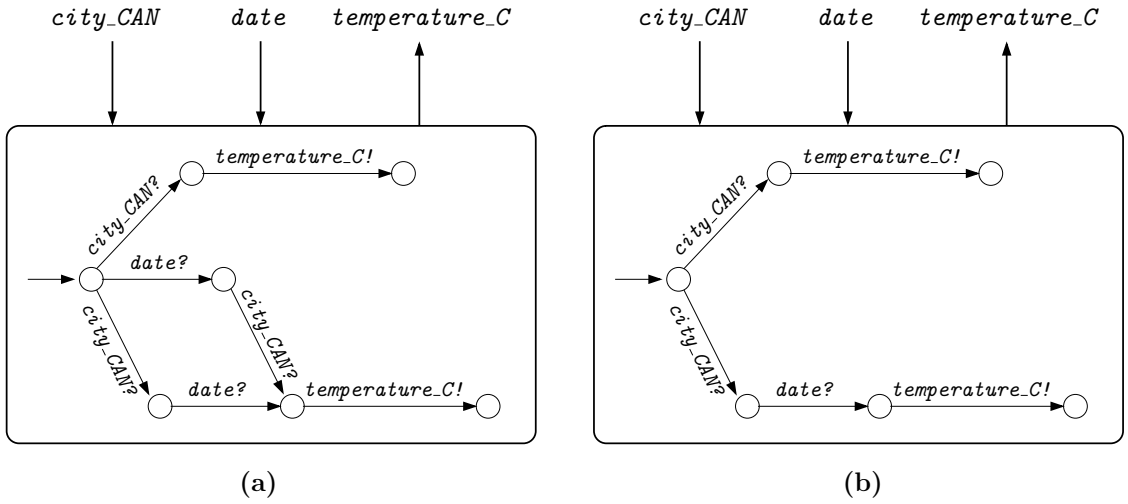



Figure 5.1: Two interface automata of the component *WeatherForecast*. (a) There is no temporal dependency between the two inputs in the second scenario. (b) In the second scenario, one of the inputs is supposed to be provided before the other one.

### 5.1.2 Dependency Graph

Along with the information stored for each component, and in order to design a mechanism by which we can find a solution for the composition planning problem, we store the repository as a graph defined below.

**Definition 5.3** Dependency graph $DG = (V, E)$ *contains information about the existing components in the repository. The set* $V$ *of nodes represents input/output data types appearing in at least one* $I_C$ *or* $O_C$*. There is a directed edge from the node* $v_x$ *to the node* $v_y$ *in the graph* ($v_x, v_y \in V$)*, if and only if there is a dependency* $v_x \rightarrow v_y$ *in at least one dependency set of at least one* $\kappa_C$*. There is also a set attached to each edge in* $E$ *that contains all components in the repository which have the corresponding dependency in one of their dependency sets.*

Based on the assumption we made in Problem 5.1, the following restrictions would hold on this dependency graph:

- Each dependency set $\kappa_C^j$ would contain only one dependency of the form $i \rightarrow o$, in which $i$ is the only input and $o$ is the only output of the component $C$ in its $j$-th scenario.

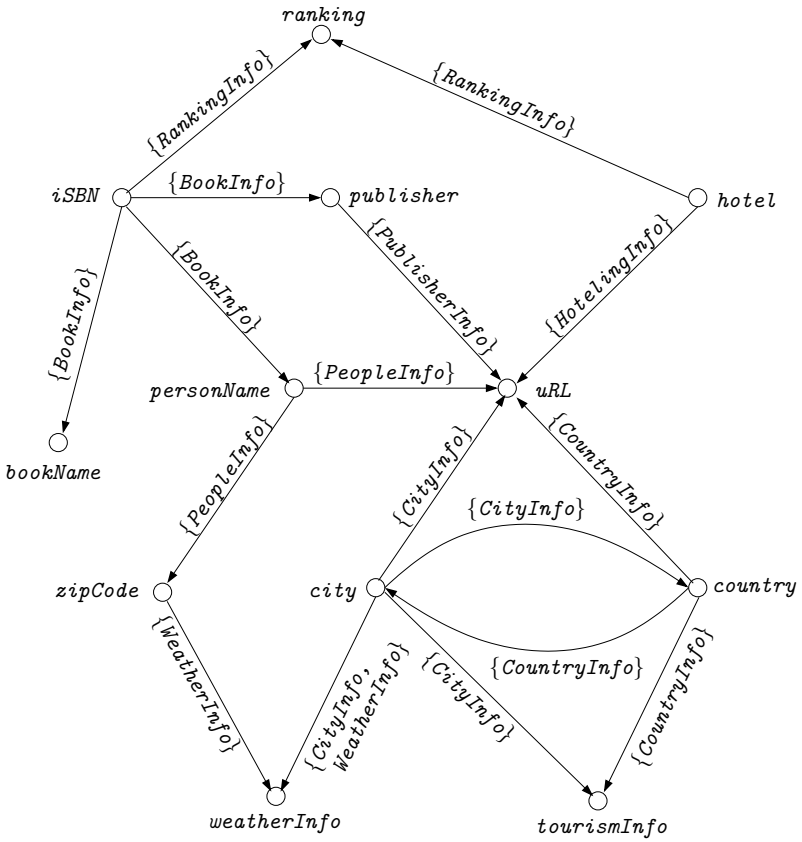



Figure 5.2: A simple dependency graph in which all components receive one input and return one output in each of their scenarios.

- There would be no temporal dependency in the dependency sets. This can also be concluded from the above restriction.

**Example 5.3** *A very simple dependency graph, taken from [47], that follows our simplifying assumption of this section is shown in Figure 5.2. Here is the specification of the components captured by this dependency graph:*

- *BookInfo*
  - $I_{BookInfo} = \{iSBN\}$
  - $O_{BookInfo} = \{bookName, personName, publisher\}$
  - $\kappa_{BookInfo} = \{\kappa^1_{BookInfo}, \kappa^2_{BookInfo}, \kappa^3_{BookInfo}\}$
    - $\kappa^1_{BookInfo} = \{iSBN \rightarrow bookName\}$
    - $\kappa^2_{BookInfo} = \{iSBN \rightarrow personName\}$

    * $\kappa^3_{BookInfo} = \{iSBN \rightarrow publisher\}$

- $CityInfo$
  - $I_{CityInfo} = \{city\}$
  - $O_{CityInfo} = \{uRL, country, tourismInfo, weatherInfo\}$
  - $\kappa_{CityInfo} = \{\kappa^1_{CityInfo}, \kappa^2_{CityInfo}, \kappa^3_{CityInfo}, \kappa^4_{CityInfo}\}$
    * $\kappa^1_{CityInfo} = \{city \rightarrow uRL\}$
    * $\kappa^2_{CityInfo} = \{city \rightarrow country\}$
    * $\kappa^3_{CityInfo} = \{city \rightarrow tourismInfo\}$
    * $\kappa^4_{CityInfo} = \{city \rightarrow weatherInfo\}$
- $CountryInfo$
  - $I_{CountryInfo} = \{country\}$
  - $O_{CountryInfo} = \{uRL, city, tourismInfo\}$
  - $\kappa_{CountryInfo} = \{\kappa^1_{CountryInfo}, \kappa^2_{CountryInfo}, \kappa^3_{CountryInfo}\}$
    * $\kappa^1_{CountryInfo} = \{country \rightarrow uRL\}$
    * $\kappa^2_{CountryInfo} = \{country \rightarrow city\}$
    * $\kappa^3_{CountryInfo} = \{country \rightarrow tourismInfo\}$
- $HotelInfo$
  - $I_{HotelInfo} = \{hotel\}$
  - $O_{HotelInfo} = \{uRL\}$
  - $\kappa_{HotelInfo} = \{\kappa^1_{HotelInfo}\}$
    * $\kappa^1_{HotelInfo} = \{hotel \rightarrow uRL\}$
- $PeopleInfo$
  - $I_{PeopleInfo} = \{personName\}$
  - $O_{PeopleInfo} = \{uRL, zipCode\}$
  - $\kappa_{PeopleInfo} = \{\kappa^1_{PeopleInfo}, \kappa^2_{PeopleInfo}\}$
    * $\kappa^1_{PeopleInfo} = \{personName \rightarrow uRL\}$
    * $\kappa^2_{PeopleInfo} = \{personName \rightarrow zipCode\}$
- $PublisherInfo$

- $I_{PublisherInfo} = \{publisher\}$
- $O_{PublisherInfo} = \{uRL\}$
- $\kappa_{PublisherInfo} = \{\kappa^1_{PublisherInfo}\}$
  - $\kappa^1_{PublisherInfo} = \{publisher \rightarrow uRL\}$

- *RankingInfo*
  - $I_{RankingInfo} = \{iSBN, hotel\}$
  - $O_{RankingInfo} = \{ranking\}$
  - $\kappa_{RankingInfo} = \{\kappa^1_{RankingInfo}, \kappa^2_{RankingInfo}\}$
    - $\kappa^1_{RankingInfo} = \{iSBN \rightarrow ranking\}$
    - $\kappa^2_{RankingInfo} = \{hotel \rightarrow ranking\}$ □

We notice that edges in the dependency graph represent dependencies that are enforced by repository components. That is why this graph is called the dependency graph.

Note that a component with multiple scenarios can be simply seen as multiple components each with one single scenario. Therefore, without loss of generality, hereafter in this section we assume that each component has only one execution scenario. As a result, we do not need to keep different dependency sets $\kappa^i_C$ for each component $C$ and can put all its dependencies in one single dependency set $\kappa_C$.

**Example 5.4** *The component* ***RankingInfo*** *of Example 5.3 has two execution scenarios. We can break this component into two components* ***BookRankingInfo*** *and* ***HotelRankingInfo*** *each with only one scenario. Their specification then would be much simpler as below:*

- *BookRankingInfo*
  - $I_{BookRankingInfo} = \{iSBN\}$
  - $O_{BookRankingInfo} = \{ranking\}$
  - $\kappa_{BookRankingInfo} = \{iSBN \rightarrow ranking\}$
- *HotelRankingInfo*
  - $I_{HotelRankingInfo} = \{hotel\}$
  - $O_{HotelRankingInfo} = \{ranking\}$

– $\kappa_{HotelRankingInfo} = \{hotel \rightarrow ranking\}$

*We can similarly break down other multi-scenario components as follows*

- *BookInfo into BookName, BookAuthor and BookPublisher,*
- *CityInfo into four components CityWebsite, CountryOfCity, CityTourism and CityWeatherInfo,*
- *CountryInfo into CountryWebsite, CountryCapital and CountryTourism,*
- *PeopleInfo into PeopleWebsite and PeopleAddress,*
- *WeatherInfo into ZipCodeWeather and CityWeatherInfo2.*

*The updated dependency graph according to the above separation of functionality is shown in Figure 5.3.* □

To simplify representing repository components and their behavior, and also for readability purposes from now on we assume that each repository component has only one execution scenario.

Considering the dependency graph of Figure 5.3 we notice that two components *CityWeatherInfo* and *CityWeatherInfo2* provide the same input-output pairing. So in case we need to use a component that receives an input of type *city* and returns an output of type *weatherInfo*, either of these components can be used. In order to enrich the dependency graph to provide better performance we introduce the concept of *component community* as follows.

**Definition 5.4** *A group of components that provide the same functionality is called a* component community. □

Therefore, instead of putting component names on the edge labels of the dependency graph, we could categorize the repository components into component communities and use component community names on the edge labels. This way, whenever we need to find a component with a specific behavior it suffices to find the corresponding component community and pick one component from that community.

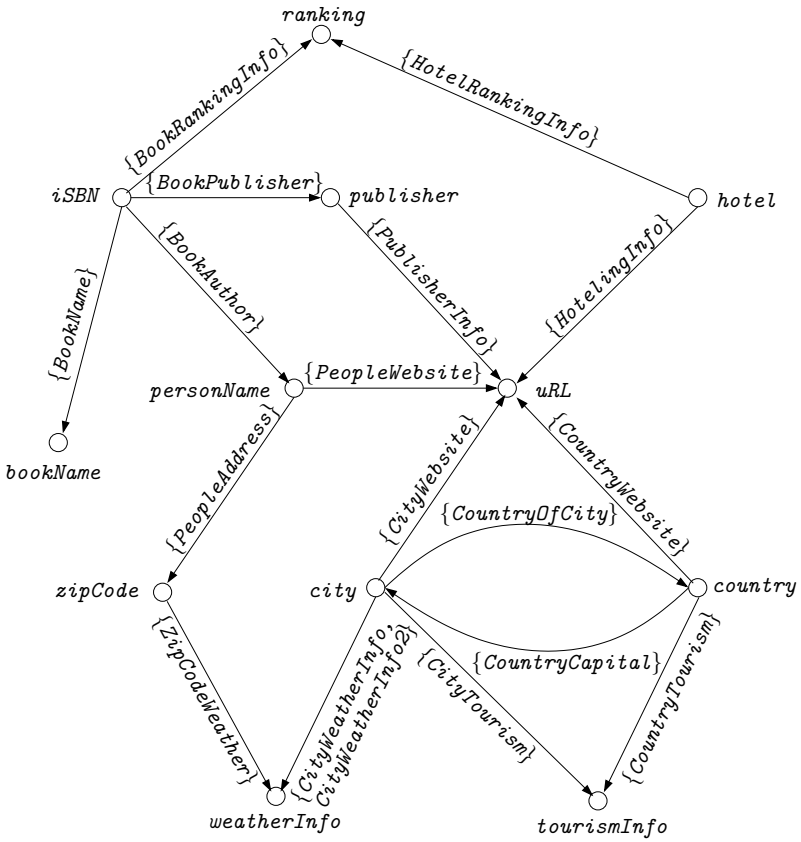



Figure 5.3: The simplified version of the dependency graph of Figure 5.2, in which each component has only one scenario.

**Example 5.5** *If the component community* `CityWeatherInfoCommunity` *contains two components* `CityWeatherInfo` *and* `CityWeatherInfo2`*, the dependency graph would have the label* `CityWeatherInfoCommunity` *on its* `city` $\rightarrow$ `weatherInfo` *edge, instead of* $\{$`CityWeatherInfo`, `CityWeatherInfo2`$\}$*. If we need to use a component to receive an input of type* `city` *and return an output of type* `weatherInfo`*, we find the component community* `CityWeatherInfoCommunity` *in the corresponding dependency graph edge label, and then look into this community and pick one of its components.* □

The concept of component community is taken from [12], where it is referred to as *service community*. Using component communities is particularly important for quality of service purposes; e.g., when we are interested to find the cheapest or the fastest composite component that satisfies a given request. Since the discussion about the quality of service is not in the scope of this thesis, without loss of generality, we assume that each repository component provides a unique functionality,

and hence, there is no need to use component communities.

So far, we provided the necessary formalism for solving the simple version of the composition planning problem. In the rest of this section we take advantage of graph search algorithms to provide a solution to the problem. We do the composition planning in two main steps:

1. We first find those components from the repository that should participate in the composition.
2. Then we find the composition plan based on the components found in the previous step.

We explain how each of these two tasks are performed in the next two subsections.

### 5.1.3 Finding Potential Components

We mentioned earlier that the dependency graph represents dependencies imposed by repository components through its edges. In other words, if there is an edge in the graph from node $v_i$ to $v_j$, there is a repository component which receives $v_i$ as its only input and returns $v_j$ as its only output. Our goal here is to check whether this collection of dependencies satisfy the dependencies requested in $\kappa_G$.

We assume that the goal component is represented similar to repository components, i.e., $G = (I_G, O_G, \kappa_G)$, where every input/output in $I_G/O_G$ appears in at least one dependency in $\kappa_G$. In case this does not hold for the given component request, we could use the default assumption that every output is dependent on all the inputs and add necessary dependencies to $\kappa_G$. There might be different execution scenarios in $G$, i.e., $\kappa_G = \{\kappa_G^1, \kappa_G^2, \cdots\}$, where each $\kappa_G^i$ $(1 \leqslant i \leqslant |\kappa_G|)$ contains at least one dependency. The problem, in terms of the dependency graph, is as follows:

**Problem 5.2** *Given a dependency graph $DG = (V, E)$ that models the repository of components $C_1$, $C_2$, $\cdots$, $C_N$ and a goal $G = (I_G, O_G, \kappa_G)$, is there a group of repository components that satisfies all the dependencies in $\kappa_G$?*

One optimal way to find if a dependency in $\kappa_G$ is satisfied by the dependency graph is running the BFS (Breadth First Search) algorithm [30] on the nodes appearing on its left-hand side. Given a graph node $v$ the BFS algorithm returns the

set of nodes reachable from $v$. For a dependency $i \rightarrow o$, we can run this algorithm on node $i$ and stop the algorithm once we notice that $o$ is reachable from $i$. If $o$ is not reachable from $i$ the algorithm terminates and returns a set not including $o$ meaning that there is no path in the graph from $i$ to $o$, or in other words, the given dependency cannot be satisfied by the repository components.

Since the BFS algorithm keeps track of the visited nodes, the path from $i$ to $o$ and also the components appearing on the edge labels of this path could be extracted, if such a path exists. This can be done using the PRINT-PATH algorithm [30], which returns the corresponding path after the BFS search has been successful.[3] If there is a graph path for every given dependency in $\kappa_G$, the problem has a solution.

**Example 5.6** *The repository of Figure 5.3 and the following goal $G$ are given.*

- $I_G = \{\texttt{iSBN}, \texttt{country}, \texttt{city}\}$
- $O_G = \{\texttt{uRL}, \texttt{personName}, \texttt{ranking}, \texttt{tourismInfo}, \texttt{weatherInfo}\}$
- $\kappa_G = \{\kappa_G^1, \kappa_G^2, \kappa_G^3\}$, *where*
  - $\kappa_G^1 = \{\texttt{iSBN} \rightarrow \texttt{uRL}\}$,
  - $\kappa_G^2 = \{\texttt{iSBN} \rightarrow \{\texttt{personName}, \texttt{ranking}\}\}$,
  - $\kappa_G^3 = \{\texttt{country} \rightarrow \texttt{tourismInfo}, \texttt{city} \rightarrow \texttt{weatherInfo}\}$. *Here, there are two inputs and outputs involved; however,* `tourismInfo` *depends only on* `country`*, while* `weatherInfo` *depends only on* `city`*.*

*In order to check if there are repository components satisfying this request, we use the* BFS *and* PRINT-PATH *algorithms on the inputs in $I_G$ to see if there is any dependency graph path to the appropriate nodes according to the dependencies in $\kappa_G$. The following is the result of applying these two algorithms on this example:*

- `iSBN`*: According to the dependencies in $\kappa_G$ we need to check if nodes* `uRL`*,* `personName` *and* `ranking` *are reachable from* `iSBN`*. By applying the* BFS *algorithm, we find out that* `personName` *and* `ranking` *are reachable in one step (using the components* `BookAuthor` *and* `BookRankingInfo`*, respectively) and* `uRL` *is reachable in two steps (using the components* `BookPublisher` *and* `PublisherInfo`*, or the components* `BookAuthor` *and* `PeopleWebsite`*).*

[3] BFS and PRINT-PATH algorithms do not consider any label on the graph edges. A very small change in the PRINT-PATH algorithm and also the data structure they use would allow us to have the edge labels in the result as well.

| Dependency | Path |
|---|---|
| $iSBN \rightarrow uRL$ | $iSBN \xrightarrow{BookPublisher} publisher \xrightarrow{PublisherInfo} uRL$, $iSBN \xrightarrow{BookAuthor} personName \xrightarrow{PeopleWebsite} uRL$ |
| $iSBN \rightarrow personName$ | $iSBN \xrightarrow{BookAuthor} personName$ |
| $iSBN \rightarrow ranking$ | $iSBN \xrightarrow{BookRankingInfo} ranking$ |
| $country \rightarrow tourismInfo$ | $country \xrightarrow{CountryTourism} tourismInfo$ |
| $city \rightarrow weatherInfo$ | $city \xrightarrow{CityWeatherInfo} weatherInfo$ |

Table 5.1: Dependency graph paths for the dependencies of Example 5.6.

- `country`: *The node* `tourismInfo` *is reachable from* `country` *in one step (using the component* `CountryTourism`*).*
- `city`: *The node* `weatherInfo` *is reachable from* `city` *in only one step (using the component* `CityWeatherInfo`*). Note that we assumed earlier that each functionality is provided by a unique component. We have ignored the component* `CityWeatherInfo2` *based on this assumption.*

*The results are summarized in Table 5.1. Since all the dependencies are satisfied by the repository components, we conclude that there exists a solution for the given request $G$.* □

Now that we have the components that should participate in the composition, we need to find out the appropriate composition plan for the requested behavior. We discuss this matter in the next part.

### 5.1.4 Finding the Composition Plan

After finding out appropriate graph paths for every single dependency in $\kappa_G$, we need to find a composition plan which is in fact a plan on how to execute the selected components so that the desired behavior is achieved. In general, two components can be composed in one of the following four ways:

- They can be synchronized according to the synchronization operation explained in Section 4.
- They can be executed sequentially.

- They can be executed conditionally.
- They can be executed in parallel.

We can find out how two involved components must be composed by considering the paths returned in the last step and also the goal triplet $(I_G, O_G, \kappa_G)$. We build the requested service by incremental composition of the involved components in the returned paths.

We first process sets of dependencies in $\kappa_G$ one by one. For a particular set $\kappa_G^i$, if there is only one dependency in it and the path returned for it is of length one, it means that there is a component in the repository that satisfies this dependency and no composition is required. Otherwise, when the path length is greater than one, we need to synchronize the components appearing on the edge labels to satisfy the dependency. This simply is because each component on this path creates an output that must be fed to the next component on the path as its input. This makes all the intermediate nodes in the path become internalized.

If there are more than one dependency in the set, it means that there are more than one input-output pairs that are of interest. For each of these dependencies we do the above compositions, if required, and then the resulting compositions are executed sequentially or in parallel in such a way that the execution does not violate any dependency in the set. The only exception is when there are some dependencies in the set having the same input on their left-hand side. For this type of dependencies the two or more outputs must appear right after that common input, and therefore, the parallel execution is the only way the request can be satisfied. This is achieved using the shared input synchronization.

After processing the dependencies in each dependency set $\kappa_G^i$, the resulting compositions for sets $\kappa_G^i$ are composed conditionally to represent different scenarios of execution. Since all composition operators are binary operations, the whole setup can be shown using a binary expression tree.

**Example 5.7** *Let us continue Example 5.6 by finding the appropriate composition. We assume that components $G_1$, $G_2$ and $G_3$ satisfy dependency sets $\kappa_G^1$, $\kappa_G^2$, and $\kappa_G^3$, respectively. Then, based on the discussion above, the proper composition for the goal $G$ would be $(G_1 \oplus G_2) \oplus G_3$ or $G_1 \oplus (G_2 \oplus G_3)$.*[4] *Now we find the composition plan for the components $G_1$, $G_2$ and $G_3$ separately. We use the results presented in Table 5.1 for this purpose.*

[4]Note that the conditional operator is both commutative and associative.

- *Since the path returned for $\kappa_G^1$ is of length 2, we need to perform a synchronization. There are two options that both satisfy this dependency:*

  - $\texttt{BookPublisher} \odot \texttt{PublisherInfo}$
  - $\texttt{BookAuthor} \odot \texttt{PeopleWebsite}$

  *There is no preference between these two solutions. At this point the user could become involved to make the decision based on the functional semantics of the request and that of the given possible solutions. But in general, the solution would be* $G_1 \equiv (\texttt{BookPublisher} \odot \texttt{PublisherInfo}) \oplus (\texttt{BookAuthor} \odot \texttt{PeopleWebsite})$.

- *Since both paths returned for $\kappa_G^2$ are of length 1, there is a component in the repository that satisfies each. These components are* `BookAuthor` *and* `BookRankingInfo`*. However, since there are multiple outputs on the right-hand side of the dependency, based on our earlier discussion we need to synchronize these two components on their shared input to satisfy the explicit dependency of both outputs to the single input. Therefore, the solution for this dependency set would be* $G_2 \equiv \texttt{BookAuthor} \odot \texttt{BookRankingInfo}$.

- *For the dependency set $\kappa_G^3$, the situation is somewhat similar to $\kappa_G^2$ with the exception that there is no common left-hand side input in the two dependencies. Therefore, the sequential or parallel execution of the components* `CountryTourism` *and* `CityWeatherInfo` *would be the solution. Since there is no restriction on their execution order, each of them may be executed ahead of the other. This suggests their parallel execution, which leads to the solution* $G_3 \equiv \texttt{CountryTourism} \,||\, \texttt{CityWeatherInfo}$.

*A binary expression tree for the composite component G is shown in Figure 5.4. The composition results can be represented using the interface automata compositions explained in Section 4.1.2. For example, the corresponding interface automaton for the above composite component G is the one shown in Figure 5.5. This interface automaton can be used to validate the result found by the composition algorithm against the given request.* □

### 5.1.5 Complexity

We used the BFS and PRINT-PATH algorithms [30] to find appropriate dependency graph paths and repository components corresponding to the given request. For

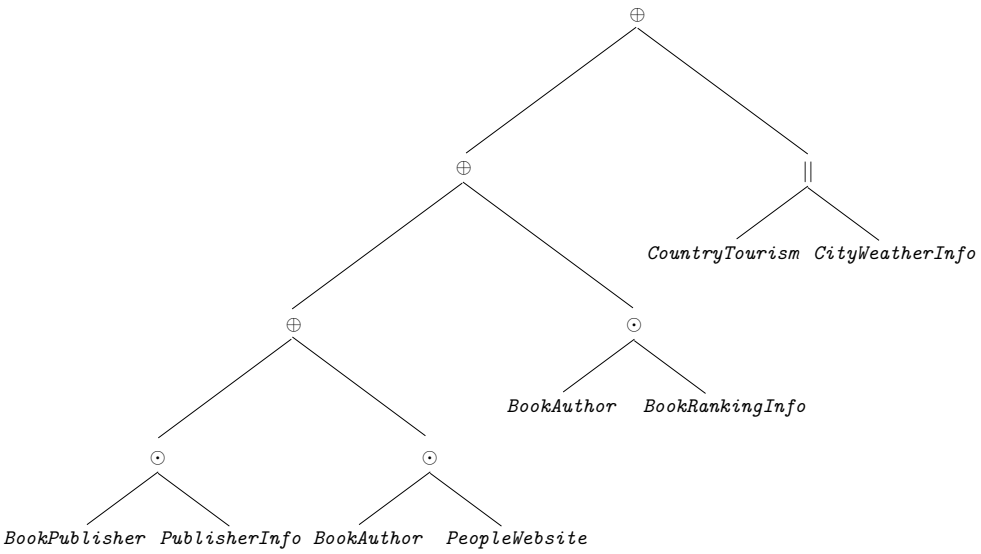



Figure 5.4: The binary expression tree representation of the plan for a component satisfying the request $G$ in Example 5.7.

every distinct dependency in every dependency set $\kappa_G^i$ we need to run the BFS algorithm to make sure that all the right-hand side nodes are reachable from the left-hand side node. Assuming that all the dependencies of the form $i \rightarrow \{o_1, o_2, \cdots\}$ are written in the separate form $i \rightarrow o_1$, $i \rightarrow o_2$, ..., the complexity of running the BFS algorithm on the set $\kappa_G$ would be $O(\|\kappa_G\| \cdot (|V| + |E|))$, in which $\|\kappa_G\|$ is the number of all these simple form dependencies, i.e., $\|\kappa_G\| = \sum_{i=1}^{|\kappa_G|} |\kappa_G^i|$.

Moreover, the complexity of the PRINT-PATH algorithm is linear in the number of nodes in the returned path, i.e., $O(|E|)$. Then the complexity of running the PRINT-PATH algorithm on all the dependencies in $\kappa_G$ would be $O(\|\kappa_G\| \cdot |E|)$.

In order to find the composition plan for the given request, we need to study all the returned paths one by one. The complexity of this process on each path would be $O(|E|)$, and again, we would have the overall $O(\|\kappa_G\| \cdot |E|)$ complexity for finding the composition plan.

By comparing the three complexity measures for different parts of the solution we conclude that the complexity of the composition approach for the simple version of the composition planning problem is $O(\|\kappa_G\| \cdot (|V| + |E|))$, i.e., linear in the size of the graph for each dependency.

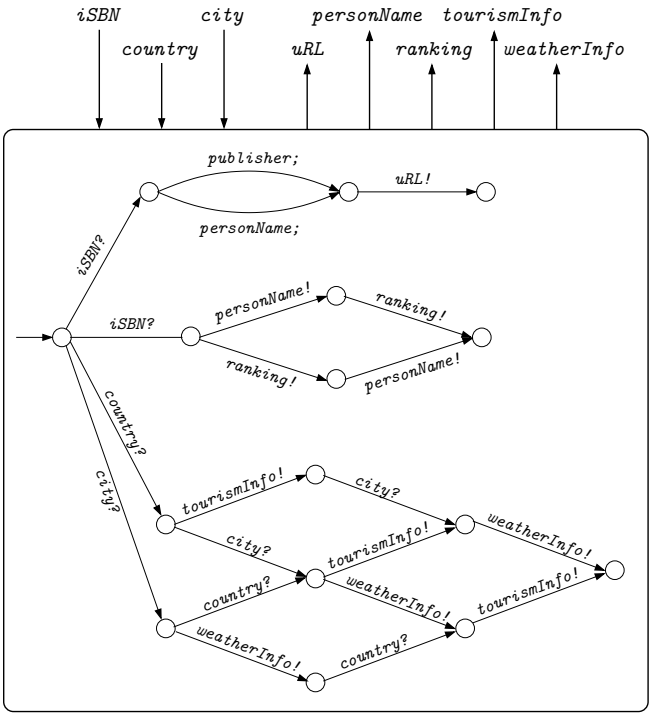



Figure 5.5: Interface automata representation of the composite component $G$. The top, middle and bottom branches correspond to $\kappa_G^1$, $\kappa_G^2$ and $\kappa_G^3$, respectively.

## 5.2 The Generic Composition Planning Problem

In the rest of this chapter, which is based on [48], we focus on a more generic form of the composition planning problem. We still assume that there is a repository of available components and there is a target component that we would like to build. However, the main difference from the simplified version of the problem is that repository components and also the request may receive/return any number of inputs/outputs, as long as they are stateless.

**Example 5.8** *We saw earlier in this chapter (Figure 5.1) two different versions of the* `WeatherForecast` *component. Based on the above definition of stateless components, the interface automaton on the left (Figure 5.1-(a)) represents a stateless component, while the component whose interface automaton is the one on the right (Figure 5.1-(b)) is not stateless. We see that in part (b) the component must receive the input* `city_CAN` *before the input* `date` *which violates the requirement for statelessness.* □

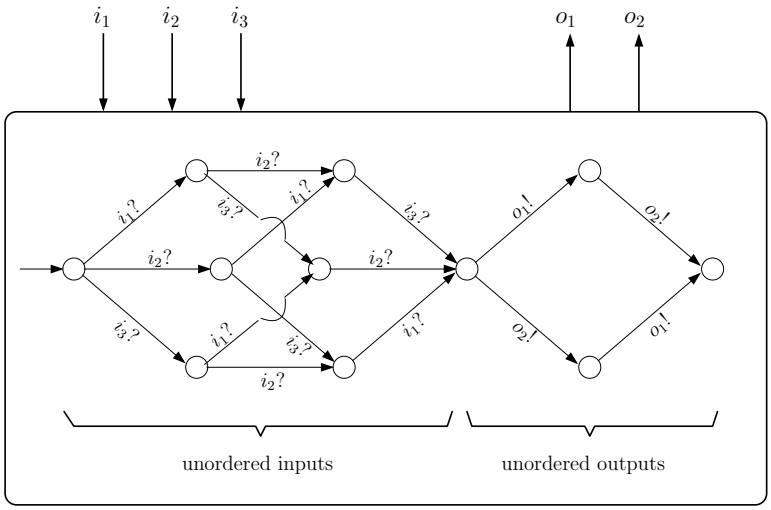



Figure 5.6: Interface automaton of a stateless component with three inputs and two outputs.

According to Definition 1.1, every execution path in the interface automaton of a stateless component must start with an unordered execution of its inputs and then finish with an unordered execution of its outputs. For example, the interface automaton of a single-scenario component with three inputs and two outputs should look like the one represented in Figure 5.6.

In terms of the composition algebra, each scenario of a stateless component $C$ is of the form $C = I \cdot O$, in which $I = (i_1 \mid\mid i_2 \mid\mid \cdots \mid\mid i_m)$ and $O = (\overline{o_1} \mid\mid \overline{o_2} \mid\mid \cdots \mid\mid \overline{o_n})$, where $m$ and $n$ are the number of inputs and outputs in that scenario, respectively. The statelessness plays an important role in the generic version of the problem, which is defined as following.

**Problem 5.3** *A repository of stateless components and a request for a target stateless component are given, where all components are described by their inputs and outputs in each execution scenario. We would like to know if the target component can be built from some of the repository components. In case the answer is YES, the appropriate composition plan is also required.*

In the next subsection we provide the necessary background for solving the generic composition planning problem. This background includes the formalism by which the behavior of components is represented, and also the improved version of the dependency graph which is used as a model of the repository to facilitate solving the problem.

### 5.2.1 Required Formalization

We use the composition algebra to describe stateless components. Although interface automata would be another option for doing this, we choose the composition algebra because, as discussed in Section 4.2, it is more understandable in general.

Again, without loss of generality, we assume that each repository component has a single execution scenario. Therefore, each repository component can be sufficiently identified by its input and output types. Note that it is possible for a component to receive/return two or more instances of a specific type. Hence, we need to use multisets[5], or bags, to represent these input and output types.

**Example 5.9** *The stateless component* `CANCityDistance` *receives two Canadian city names and returns their distance in kilometers. Therefore, we would have*

- $I_{\texttt{CANCityDistance}} = \{\texttt{city\_CAN}^2\}$
- $O_{\texttt{CANCityDistance}} = \{\texttt{distance\_Km}\}$

*Note that the superscript* 2 *in* $\texttt{city\_CAN}^2$ *indicates that two inputs of type* `city_CAN` *are involved in this component. In composition algebra, this behavior would be represented as* $\texttt{CANCityDistance} = (\texttt{city\_CAN} \,||\, \texttt{city\_CAN}) \cdot \overline{\texttt{distance\_Km}}$. □

The target component, however, could have different execution scenarios. Therefore, it is described as the triplet $G = (I_G, O_G, \kappa_G)$, where $\kappa_G = \{\kappa_G^1, \kappa_G^2, \cdots\}$.

We further assume that in each repository and goal component every output is, by default, dependent on all the inputs because otherwise the functionality can be decomposed.

**Example 5.10** *Consider a goal component* $G$ *that receives inputs* `country` *and* `zipCode`*, and returns outputs* `capital` *and* `address`*. If we intend* $G$ *to return the capital of a given country, and the address associated with a zip code, we would have to decompose it into two components; one receiving* `country` *and returning* `capital`*, and the other receiving* `zipCode` *and returning* `address`*.* □

[5] A multiset is a set in which the cardinality of elements matters.

### 5.2.2 Improved Dependency Graph

If we look closely at the dependency graph of Section 5.1.2 and consider the generic version of the problem, we notice that the graph cannot fully capture the information about repository components. In particular, the first version of the dependency graph has the following shortcomings:

- It fails to properly model components with two or more inputs/outputs. For example, the component *WeatherForecast* of Figure 5.1-(a) in its second scenario receives an input of type *city_CAN* and another input of type *date* and returns an output of type *temperature_C*. The most the earlier dependency graph can do is to add edges $city\_CAN \xrightarrow{\{WeatherForecast\}} temperature\_C$ and $date \xrightarrow{\{WeatherForecast\}} temperature\_C$. However, for this component, these two edges could be interpreted as two different scenarios, where in one the (current) temperature of a given city, and in the other, the temperature (of a default location) in a given date is returned.

- It fails to represent the cardinality of inputs/outputs when more than one input/output of a specific type are involved. For instance, the component *CANCityDistance* receives two inputs of type *city_CAN* and returns an output of type *distance_Km*. For this component, the earlier dependency graph would add the edge $city\_CAN \xrightarrow{\{CANCityDistance\}} distance\_Km$, which would be interpreted as a functionality, in which some distance value for a Canadian city is returned.

Therefore, we need to improve the dependency graph so that it can capture the above attributes properly. In the new version, nodes and edges represent similar information, i.e., input/output types and input-output dependencies, respectively. However, we make some changes in the format of edge labels in order to overcome the above shortcomings:

- To represent multiple input/output data types involved in each execution scenario, we add the scenario name to the edge labels. This scenario name is normally the name of the corresponding component operation. For example, if we assume that the second execution scenario of the *WeatherForecast* component corresponds to its *ForecastForDate* operation, then the edge label $\{(WeatherForecast, ForecastForDate)\}$ would appear on $city\_CAN \rightarrow temperature\_C$ and $date \rightarrow temperature\_C$ edges. In the special case that

each repository component has only one scenario, this improvement could be ignored. In the rest of this thesis, without loss of generality, we assume that this special case applies to the component repository unless otherwise is explicitly mentioned. Therefore, we skip the operation names on the edge labels assuming that each repository component comes with only one scenario.

- To represent the cardinality of inputs and outputs, we add a pair of numbers to the labels to show the cardinality of data types at both ends of the edge that are involved in the corresponding scenario. For example, the updated label for the edge *city_CAN* $\rightarrow$ *distance_Km* according to the component *CANCityDistance* would be $\{$(*CANCityDistance, 2, 1*)$\}$, indicating that two instances of type *city_CAN* are given and one instance of type *distance_Km* is returned.

So far, each edge in the dependency graph exists because of some repository component that relates the two data types at the two ends of that edge. Other than this dependency relation, there are relations that could relate two or more data types to each other.

**Example 5.11** *The relation between the data types* `capital` *and* `city` *likely would not be captured by any component in the repository. Nonetheless, we know that there is a relationship between the two concepts, i.e.,* `capital` *is a subtype of* `city`. □

**Example 5.12** *The* `address_CAN` *data type is a composite data type which includes component data types* `streetAddr`, `city_CAN`, `province_CAN`, `zipCode_CAN`. *Each repository component that receives/returns Canadian address information, would probably work either with the composite form or the component form, but not both. Therefore, the relationship between the two forms would not be captured by the dependency edges.* □

In order to represent different types of relationships between data types, we categorize graph edges into three groups:

- Dependency edges: These edges are represented by arrows of the form $\rightarrow$ and are the normal dependency graph edges we have seen so far. They connect nodes corresponding to the inputs of components to the nodes corresponding to their outputs. As mentioned earlier, dependency edge labels are sets of triplets of the form $(C, card_i, card_o)$, where $C$ is the component name, and $card_i$ and $card_o$ are the number of involved inputs and outputs of the specific data type, respectively. Dependency edges are unidirectional.

- Generalization-Specialization (GenSpec) edges: These edges, represented by arrows of the form $\twoheadrightarrow$, are used to relate supertypes and subtypes to each other. For example, to show that *capital* is a subtype of *city* (Example 5.11) the edge *capital* $\twoheadrightarrow$ *city* is added to the graph. These edges do not have any labels, as they do not represent any component and only indicate that an instance of the subtype can be considered as an instance of the supertype. When considering a graph path which includes some edge $u \twoheadrightarrow v$ we can ignore this edge by combining the nodes $u$ and $v$. GenSpec edges are unidirectional.

- Composition-Decomposition (CD) edges: These edges are shown by arrows of the form $\diamond\!\!\rightarrow$ and relate composite types and their component types to each other. The diamond end of the edge points to the composite type, while the arrow end points to the component type. CD edges are bidirectional. A single number appears on these edges that identifies the number of component type instances used in the composite type. For example, for the composite type *address_CAN* of Example 5.12, edges *address_CAN* $\overset{1}{\diamond\!\!\rightarrow}$ *streetAddr*, *address_CAN* $\overset{1}{\diamond\!\!\rightarrow}$ *city_CAN*, *address_CAN* $\overset{1}{\diamond\!\!\rightarrow}$ *province_CAN*, and *address_CAN* $\overset{1}{\diamond\!\!\rightarrow}$ *zipCode_CAN* are added to the graph. The conversion between the two formats can be done using auxiliary components. These auxiliary components either decompose the composite types into its component types, or do the opposite. This information, which includes the pair of composer and decomposer components, is attached to the composite type node in the graph. When considering a graph path which includes some edge $u \overset{k}{\diamond\!\!\rightarrow} v$ each involved instance of type $u$ would create $k$ instances of type $v$. Alternatively, for a graph path which includes some edge $v \overset{k}{\leftarrow\!\!\diamond} u$, $k$ instances of type $v$ are required, though not necessarily sufficient, to produce an instance of type $u$.

**Example 5.13** *Figure 5.7 shows an example of the improved dependency graph for the following component repository. We emphasize here that we are studying stateless components, and also following the assumption that each component has a single execution scenario.*

- *PhoneNoLocation*
  - $I_{PhoneNoLocation} = \{$*phoneNo_CAN*$\}$
  - $O_{PhoneNoLocation} = \{$*address_CAN*$\}$

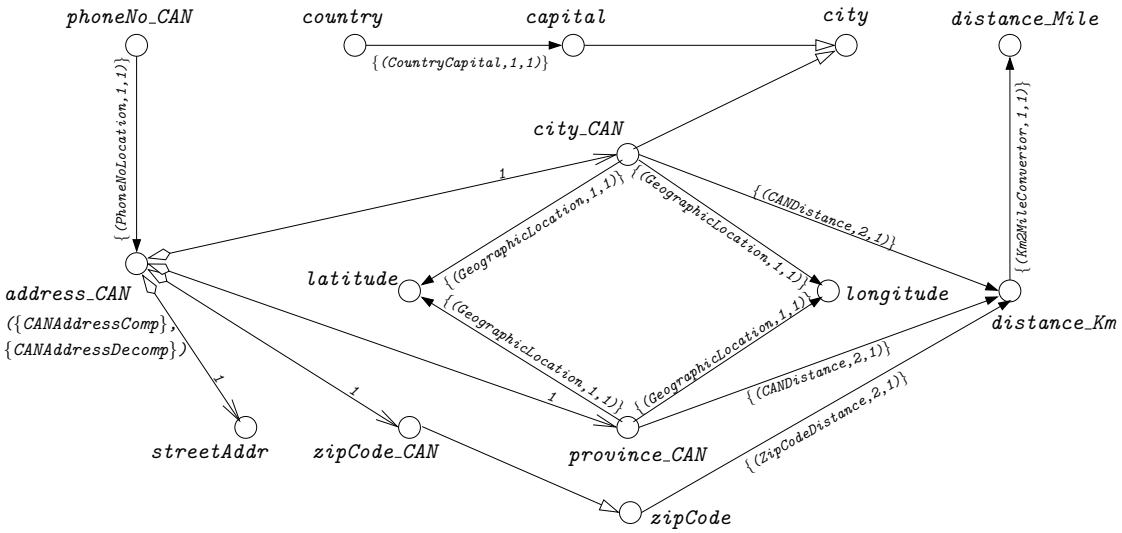



Figure 5.7: An example of the improved dependency graph.

- *GeographicLocation*
  - $I_{GeographicLocation} = \{city\_CAN, province\_CAN\}$
  - $O_{GeographicLocation} = \{latitude, longitude\}$
- *CANCityDistance*
  - $I_{CANDistance} = \{city\_CAN^2, province\_CAN^2\}$
  - $O_{CANDistance} = \{distance\_Km\}$
- *ZipCodeDistance*
  - $I_{ZipCodeDistance} = \{zipCode^2\}$
  - $O_{ZipCodeDistance} = \{distance\_Km\}$
- *Km2MileConvertor*
  - $I_{Km2MileConvertor} = \{distance\_Km\}$
  - $O_{Km2MileConvertor} = \{distance\_Mile\}$
- *CountryCapital*
  - $I_{CountryCapital} = \{country\}$
  - $O_{CountryCapital} = \{capital\}$

- *CANAddressComp (an auxiliary composer)*
  - $I_{CANAddressComp} = \{streetAddr, city\_CAN, province\_CAN, zipCode\_CAN\}$
  - $O_{CANAddressComp} = \{address\_CAN\}$
- *CANAddressDecomp (an auxiliary decomposer)*
  - $I_{CANAddressDecomp} = \{address\_CAN\}$
  - $O_{CANAddressDecomp} = \{streetAddr, city\_CAN, province\_CAN, zipCode\_CAN\}$

*There are also three supertype-subtype relations among the present repository data types, i.e.,* $capital \subseteq city$, $city\_CAN \subseteq city$ *and* $zipCode\_CAN \subseteq zipCode$. □

Note that

- Attached to each composite data type node there is a pair $(C, D)$ where $C$ is the set of auxiliary composers and $D$ is the set of auxiliary decomposers for that data type.
- For two semantically equivalent data types $t_1$ and $t_2$, the graph would contain both edges $t_1 \rightarrowtriangle t_2$ and $t_1 \leftarrowtriangle t_2$.

**Observation 5.1** *CD edges can be simulated by dependency edges. Figure 5.8 represents this simulation. Each CD edge is simulated by two dependency edges with opposite directions, where one edge identifies composer components and the other identifies decomposer ones.* □

We have already seen, in Section 5.1.3, how dependency edges help in solving the problem. We explain the advantage of having CD and GenSpec edges in the dependency graph through the following examples.

**Example 5.14** *Consider the dependency graph of Figure 5.7 and the goal G as*

- $I_G = \{zipCode\_CAN^2\}$
- $O_G = \{distance\_Km\}$
- $\kappa_G = \{\kappa_G^1\}$, *where* $\kappa_G^1 = \{zipCode\_CAN^2 \rightarrow distance\_Km\}$

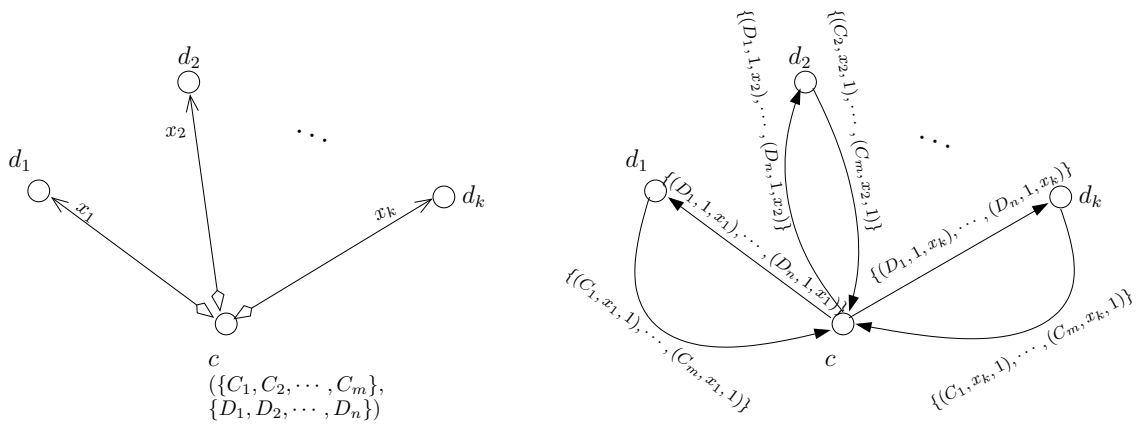



Figure 5.8: CD edges can be simulated by dependency edges.

*It is obvious that Canadian zip codes are zip codes in general. Therefore, they can be fed to the component* `ZipCodeDistance` *as inputs to produce the required output* `distance_Km`*. In fact, this plan works because the path* `zipCode_CAN` $\twoheadrightarrow$ `zipCode` $\rightarrow$ `distance_Km` *exists in the graph.* □

**Example 5.15** *The goal $G$ is described as*

- $I_G = \{$`phoneNo_CAN`$\}$
- $O_G = \{$`zipCode_CAN`$\}$
- $\kappa_G = \{\kappa_G^1\}$, *where* $\kappa_G^1 = \{$`phoneNo_CAN` $\rightarrow$ `zipCode_CAN`$\}$

*Again, we can easily see that a Canadian phone number can be converted to a Canadian address using the component* `PhoneNoLocation`*. Then the corresponding Canadian zip code can be found by decomposing this address using the component* `CANAddressDecomp`*. The graph path* `phoneNo_CAN` $\rightarrow$ `address_CAN` $\diamond\!\!\rightarrow$ `zipCode_CAN` *gives us the chance to find this plan.* □

The discussion above leads us to the following observation:

**Observation 5.2** *For every scenario of the goal component, if there is no path in the dependency graph from each input to each output, the composition planning problem has no solution.* □

After the discussion on the requirements for solving the generic version of the problem (Problem 5.3), in the rest of this section we explain how those requirements, i.e., the composition algebra and the dependency graph, are actually used for this purpose. Similar to the solution to the simple version, we solve the generic version in two steps.

1. Finding proper repository components that could participate as part of the composition plan.
2. Finding the composition plan.

### 5.2.3 Finding Potential Components

Assuming that the target component has a single scenario, based on the above observation, the first condition to be checked is whether there is a graph path from each input in $I_G$ to each output in $O_G$. If there is no path for any pair $(i, o)$, where $i \in I_G$ and $o \in O_G$, the 'NO' answer will be returned. This condition is captured by the following lemma.

**Lemma 5.1** *Consider an execution scenario $\kappa_G^i$ of the goal $G$ with inputs $I_G^i$ and outputs $O_G^i$. In order for the composition planning problem to have a solution for this scenario, for every $i \in I_G^i$ and every $o \in O_G^i$, there must be a graph path from $i$ to $o$, their corresponding dependency graph nodes.*

**Proof.** For any such pair $(i, o)$, if either $i$ or $o$ does not have a corresponding node in the graph, there would not be any repository component that receives/returns them, meaning that there would be no solution. If they do have corresponding graph nodes, but there is no path from the node $i$ to the node $o$, $o$ would be unreachable from $i$. This means that there would not be any sequence of repository components that produces $o$ from $i$. □

**Example 5.16** *The following goal $G$ is given against the repository of Figure 5.7:*

- $I_G = \{\texttt{phoneNo\_CAN}^2\}$
- $O_G = \{\texttt{distance\_Mile}\}$
- $\kappa_G = \{\kappa_G^1\}$, *where* $\kappa_G^1 = \{\texttt{phoneNo\_CAN}^2 \rightarrow \texttt{distance\_Mile}\}$

*Based on the above lemma, since there exists a path from the node* ***phoneNo_CAN*** *to the node* ***distance_Mile****, the goal G could have a solution. In fact, there are three such graph paths:*

- $phoneNo\_CAN \rightarrow address\_CAN \diamond\!\!\rightarrow city\_CAN \rightarrow distance\_Km \rightarrow distance\_Mile$
- $phoneNo\_CAN \rightarrow address\_CAN \diamond\!\!\rightarrow province\_CAN \rightarrow distance\_Km \rightarrow distance\_Mile$
- $phoneNo\_CAN \rightarrow address\_CAN \diamond\!\!\rightarrow zipCode\_CAN \twoheadrightarrow zipCode \rightarrow distance\_Km \rightarrow distance\_Mile$ □

But this constraint is not enough in finding potential components. In other words, there might be graph paths satisfying the given dependency set, while there is no possible composition plan.

**Example 5.17** *Although there is a graph path from the given input to the given output for both of the following requests, no composition plan exists for them according to the graph of Figure 5.7.*

- $G_1$
  - $I_{G_1} = \{city\_CAN\}$
  - $O_{G_1} = \{longitude\}$
  - $\kappa_{G_1} = \{\{city\_CAN \rightarrow longitude\}\}$

  *The only component that comes into play here is* `GeographicLocation`*. However, it needs an input of type* ***province_CAN*** *as well to create a* ***longitude*** *value.*

- $G_2$
  - $I_{G_2} = \{zipCode\}$
  - $O_{G_2} = \{distance\_Km\}$
  - $\kappa_{G_2} = \{\{zipCode \rightarrow distance\_Km\}\}$

  *Here, the component* `ZipCodeDistance` *could be useful. However, it fails because it needs two instances of* ***zipCode****.* □

We convert the problem of finding potential components to the problem of finding appropriate graph paths. Once these paths are found, their edge labels would identify the potential components. We explain the approach for a single-scenario target component. It is trivial that for a multi-scenario request, we need to find a solution for each scenario and then, compose them all conditionally to find a solution for the whole request. Therefore, the problem is narrowed down to the following:

**Problem 5.4** *How the improved dependency graph can be used to find potential components for the following single-scenario goal $G$?*

- $I_G = \{i_1^{k_1}, i_2^{k_2}, \cdots, i_m^{k_m}\}$
- $O_G = \{o_1^{l_1}, o_2^{l_2}, \cdots, o_n^{l_n}\}$
- $\kappa_G = \{\kappa_G^1\}$, *where* $\kappa_G^1 = \{I_G \rightarrow O_G\}$ □

According to Lemma 5.1, the first thing to do is to check for a graph path from each graph node whose label is in $I_G$ to each graph node whose label is in $O_G$. If there is no graph node for some element in $I_G$ or $O_G$, or if there is no path for some input-output pair from $I_G$ and $O_G$, the goal $G$ cannot be satisfied by the given repository.

Now, let us assume that all the necessary graph paths mentioned in Lemma 5.1 exist for $G$. Now some investigation needs to be performed on their edge labels in order to find out if they are eligible.

**Definition 5.5** *For a dependency $d$ in the dependency set $\kappa_G^1$, like $I_G \rightarrow O_G$ in Problem 5.4, each dependency part $p_d^j = i_x^{k_x} \rightarrow o_y^{l_y}$ $(1 \leqslant x \leqslant m, 1 \leqslant y \leqslant n)$ is called a* partial dependency *for $d$. The set of all partial dependencies of $d$ is shown as $\mathcal{P}(d)$.* □

**Definition 5.6** *Assume that we have already found a graph path for the partial dependency $p_d^j = i_x^{k_x} \rightarrow o_y^{l_y}$ $(1 \leqslant x \leqslant m, 1 \leqslant y \leqslant n)$. This path, which could contain all the three types of edges, in general looks like $i_x \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_{n-1} \rightarrow o_y$, where $\rightarrow \in \{\rightarrow, \rightarrowtriangle, \diamond\!\!\rightarrow, \leftarrow\!\!\diamond\}$. For each dependency edge in this path, only one of the triplets from the edge label must be selected. Such a path, which has only one triplet on its dependency edges, is called a* path instance. *This path instance is called an* eligible path instance *for the partial dependency $p_d^j$, if and only if $k_x$ and $l_y$ satisfy the cardinality constraints on the dependency and CD edges.*

*The satisfiability of cardinality constraints can be verified through the following process: We remove all the GenSpec edges by combining the nodes at both ends, and simulate each CD edge by a dependency edge. Note that, in general, each CD edge is simulated by two dependency edges; but here, we keep only the one that is in the same direction as the path instance. Therefore, the path instance after these changes would look like* $i_x \xrightarrow{(C_1,x_1,y_1)} u_1 \xrightarrow{(C_2,x_2,y_2)} u_2 \cdots u_{m-1} \xrightarrow{(C_m,x_m,y_m)} o_y$ *which, again, corresponds to the partial dependency* $p_d^j = i_x^{k_x} \rightarrow o_y^{l_y}$*. This path instance is eligible, if and only if the following set of equations has integer solutions for* $n_1$*,* $n_2$*, ...,* $n_m$*.*[6]

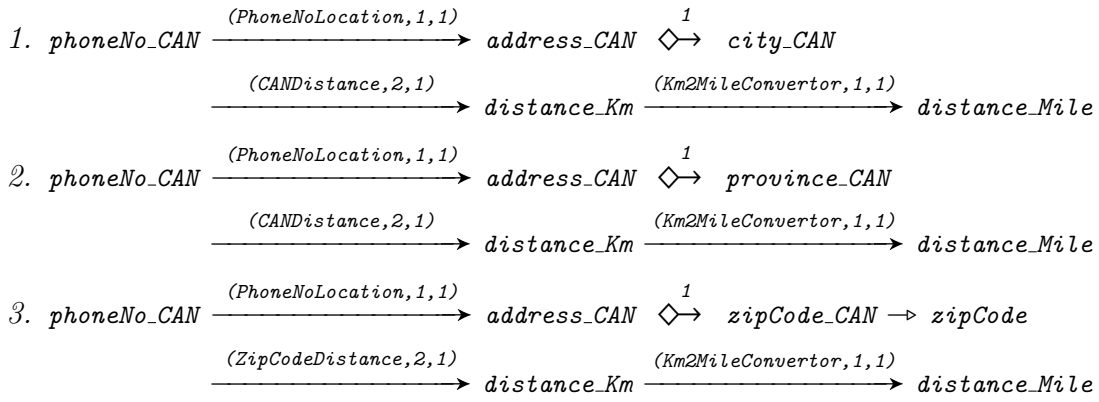
$$
\begin{aligned}
& k_x = n_1 \times x_1 \\
& n_1 \times y_1 = n_2 \times x_2 \\
& \cdots \\
& n_{m-1} \times y_{m-1} = n_m \times x_m \\
& n_m \times y_m = l_y \\
& GCD(k_x, n_1, n_2, \cdots, n_{m-1}, n_m, l_y) = 1
\end{aligned}
$$

*The last condition requires the greatest common divisor of the numbers* $k_x$*,* $n_1$*,* $n_2$*, ...,* $n_{m-1}$*,* $n_m$ *and* $l_y$ *to be 1. If the greatest common divisor of these numbers is greater than 1, say* $g$ *(*$g > 1$*), then the partial dependency* $i_x^{\frac{k_x}{g}} \rightarrow o_y^{\frac{l_y}{g}}$ *would also have an eligible path instance with all the above numbers divided by* $g$*. This means that each instance of* $o_y$ *would be dependent on only* $\frac{k_x}{g}$ *instances of* $i_x$*, not all the* $k_x$ *instances.*

*Each partial dependency might have more than one eligible path instance. The set of all eligible path instances of a partial dependency* $p$ *is called its* eligible path instance set *and is shown by* $\mathcal{E}(p)$*.* □

**Example 5.18** *Consider the single-scenario goal* $G$ *of Example 5.16 again. The dependency set* $\kappa_G^1$ *has only one dependency* $d =$ ***phoneNo_CAN***$^2 \rightarrow$ ***distance_Mile*** *with only one partial dependency* $p_d =$ ***phoneNo_CAN***$^2 \rightarrow$ ***distance_Mile****. Since it has a single input type and a single output type, we need to find only one set of eligible path instances, i.e., from the node* ***phoneNo_CAN*** *to the node* ***distance_Mile****. We saw that there are three different graph paths for this partial dependency. Each of these three paths shown in Example 5.16 has a single path instance because every dependency edge has a single triplet in its label. So there are three path instances that need to be checked for eligibility:*

[6]Note that if we accept more outputs than originally requested, we would have to solve a set of inequalities instead, which potentially could result in multiple solutions.




*The partial dependency $p_d$ involves two instances of* **phoneNo_CAN** *and one instance of* **distance_Mile**. *In order to check the eligibility of the first path against this partial dependency, we start with the two instances of* **phoneNo_CAN**. *Following the path instance, we get two instances of* **address_CAN** *(by applying* **PhoneNoLocation** *on each available* **phoneNo_CAN**), *then two instances of* **city_CAN**, *then one instance of* **distance_Km** *(note that* **CANDistance** *converts two instances of* **city_CAN** *into one instance of* **distance_Km**), *and finally, one instance of* **distance_Mile**, *which satisfies the one* **distance_Mile** *instance requested by $p_d$. Therefore, the first path above is an eligible path for the partial dependency $p_d$. Note that, in terms of the set of equations in Definition 5.6, we would have $n_1 = 2$, $n_2 = 2$, $n_3 = 1$ and $n_4 = 1$ as a valid solution.*

*Since the other two path instances have the same cardinality constraints, they are also eligible. As a result, the partial dependency $p_d$ has three eligible path instances ($|\mathcal{E}(p_d)| = 3$) according to the dependency graph of Figure 5.7.* □

**Example 5.19** *Consider the partial dependency $p =$* **zipCode**$^4 \rightarrow$ **distance_Km**$^2$ *and the dependency graph of Figure 5.7. The corresponding path instance for $p$ would be* **zipCode** $\xrightarrow{(ZipCodeDistance,2,1)}$ **distance_Km**. *Although the set of equations in Definition 5.6 would return $n_1 = 2$, since $GCD(4, 2, 2) = 2 \neq 1$ the answer is rejected and no eligible path instance would be returned for $p$. Note that $n_1 = 2$ means that the component* **ZipCodeDistance** *would be used twice to convert four instances of* **zipCode** *to two instances of* **distance_Km**, *but each of these two instances would be dependent on only two of the four* **zipCode** *instances, not all of them.* □

Although by defining the concept of eligible path instances we took one step forward in finding appropriate graph paths, and hence finding potential components, this eligibility alone is not enough.

**Example 5.20** *Consider the request $G_1$ in Example 5.17 one more time. This request has one dependency $d$ and one partial dependency $p_d$, where $d = p_d =$ `city_CAN` $\rightarrow$ `longitude`. This partial dependency has a single eligible path instance, i.e., $\mathcal{E}(p_d) = \{$`city_CAN` $\xrightarrow{(GeographicLocation,1,1)}$ `longitude`$\}$. Nonetheless, the request has no answer since the component `GeographicLocation` needs an input of type `province_CAN` as well.* □

The next step is taken by the following definition:

**Definition 5.7** *Consider the general request $G$ defined in Problem 5.4. A* consistent eligible path instance set *(or simply* consistent set*) $cs$ for a dependency $d \in \kappa_G^1$ is a set for which its size is at least the number of partial dependencies in $d$. For each partial dependency $p$ in $\mathcal{P}(d)$, this set has at least one eligible path instance from $\mathcal{E}(p)$. These two properties are formally represented as $|cs| \geqslant |\mathcal{P}(d)|$ and $\forall p \in \mathcal{P}(d) \centerdot |cs \cap \mathcal{E}(p)| \geqslant 1$. In fact, the $>$ case in these two formulas might occur only because of the following two restrictions on the consistent eligible path instance set $cs$:*

- *If $cs$ contains an eligible path instance $e$ which contains some edge $v_i \xrightarrow{(C,x,z)} v_j$, every other graph edge $v_k \xrightarrow{(C,y,z)} v_j$ $(k \neq i)$ must also belong to some eligible path instance in $cs$.*
- *If $cs$ contains an eligible path instance $e$ which contains some edge $v_i \overset{x}{\longleftarrow\!\diamond} v_j$, every other graph edge $v_k \overset{y}{\longleftarrow\!\diamond} v_j$ $(k \neq i)$ must also belong to some eligible path instance in $cs$.*

*These properties guarantee that all the components appearing on the eligible path instance edges in $cs$ could be actually used by making sure that all their required inputs are available. The set of all possible consistent sets for the dependency $d$ is shown as $\mathcal{CS}(d)$.* □

We show through some examples how these restrictions help in finding appropriate path instances and, as a result, potential components for the given request.

**Example 5.21** *The dependency graph of Figure 5.7 and the following goal $G$ are given.*

- $I_G = \{$`city_CAN`, `province_CAN`$\}$

- $O_G = \{\texttt{latitude}\}$
- $\kappa_G = \{\kappa_G^1\}$, $\kappa_G^1 = \{d = \{\texttt{city\_CAN}, \texttt{province\_CAN}\} \rightarrow \texttt{latitude}\}$

*Then we would have*

- $\mathcal{P}(d) = \{p_d^1 = \texttt{city\_CAN} \rightarrow \texttt{latitude},\ p_d^2 = \texttt{province\_CAN} \rightarrow \texttt{latitude}\}$
- $\mathcal{E}(p_d^1) = \{\texttt{city\_CAN} \xrightarrow{(\texttt{GeographicLocation},1,1)} \texttt{latitude}\}$
- $\mathcal{E}(p_d^2) = \{\texttt{province\_CAN} \xrightarrow{(\texttt{GeographicLocation},1,1)} \texttt{latitude}\}$

*Since each partial dependency has only one eligible path instance, and each consistent set cs for d must have at least one eligible path instance for each partial dependency, we put them both in cs. Therefore, the only possible solution would be*

$$cs = \{e_1 = \texttt{city\_CAN} \xrightarrow{(\texttt{GeographicLocation},1,1)} \texttt{latitude},\ e_2 = \texttt{province\_CAN} \xrightarrow{(\texttt{GeographicLocation},1,1)} \texttt{latitude}\}$$

*We also need to check the constraints given in the above definition,*

- *since* $e_1 = \texttt{city\_CAN} \xrightarrow{(\texttt{GeographicLocation},1,1)} \texttt{latitude}$ *is in cs, and there is another graph edge* $\texttt{province\_CAN} \rightarrow \texttt{latitude}$ *which has the same triplet in its label, the edge* $\texttt{province\_CAN} \xrightarrow{(\texttt{GeographicLocation},1,1)} \texttt{latitude}$ *must also belong to some eligible path instance in cs. We see that this edge already belongs to* $e_2 \in cs$.
- *since* $e_2 = \texttt{province\_CAN} \xrightarrow{(\texttt{GeographicLocation},1,1)} \texttt{latitude}$ *is in cs, and there is another graph edge* $\texttt{city\_CAN} \rightarrow \texttt{latitude}$ *which has the same triplet in its label, the edge* $\texttt{city\_CAN} \xrightarrow{(\texttt{GeographicLocation},1,1)} \texttt{latitude}$ *must also belong to some eligible path instance in cs. We see that this edge already belongs to* $e_1 \in cs$.

*Therefore, the above consistent set cs is a consistent eligible path instance set for the given dependency and the given request (since there is only one dependency in the request). Note that although these path instances indicate the "one* `city_CAN` *to one* `latitude`*" and "one* `province_CAN` *to one* `latitude`*" conversions separately, together they do not indicate the "one* `city_CAN` *and one* `province_CAN` *to two*

*latitude" conversion. The reason is the common component which appears on the edges* $\texttt{city\_CAN} \rightarrow \texttt{latitude}$ *and* $\texttt{province\_CAN} \rightarrow \texttt{latitude}$*, indicating that only one* `latitude` *would be created given one* `city_CAN` *and one* `province_CAN`*. If these components were different, the former indication would have been valid.* □

**Example 5.22** *Consider the following goal* $G$ *against the same dependency graph:*

- $I_G = \{\texttt{city\_CAN}^2\}$
- $O_G = \{\texttt{distance\_Km}\}$
- $\kappa_G = \{\kappa_G^1\}$, $\kappa_G^1 = \{d = \texttt{city\_CAN}^2 \rightarrow \texttt{distance\_Km}\}$

*As a result,*

- $\mathcal{P}(d) = \{p_d = \texttt{city\_CAN}^2 \rightarrow \texttt{distance\_Km}\}$
- $\mathcal{E}(p_d) = \{\texttt{city\_CAN} \xrightarrow{(\texttt{CANDistance,2,1})} \texttt{distance\_Km}\}$

*Potentially,* $cs = \{e_1 = \texttt{city\_CAN} \xrightarrow{(\texttt{CANDistance,2,1})} \texttt{distance\_Km}\}$ *is a consistent set for* $d$*. However, since there is another graph edge* $\texttt{province\_CAN} \rightarrow \texttt{distance\_Km}$ *with the same triplet in its label, and this edge does not belong to any path instance in* $cs$*, the potential solution would be rejected. Since there is no other alternative for* $cs$ *we conclude that this goal cannot be satisfied against the given graph.* □

**Example 5.23** *For the request given in Example 5.16, we would have*

- $d = \texttt{phoneNo\_CAN}^2 \rightarrow \texttt{distance\_Mile}$
- $\mathcal{P}(d) = \{p_d = \texttt{phoneNo\_CAN}^2 \rightarrow \texttt{distance\_Mile}\}$
- $\mathcal{E}(p_d) = \{e_1, e_2, e_3\}$*, where* $e_1$*,* $e_2$ *and* $e_3$ *were listed in Example 5.18, respectively.*

*Therefore, three possible consistent eligible path instance sets exist for* $d$*:*

- $cs_1 = \{e_1\}$*: In this case, because of the edge* $\texttt{city\_CAN} \xrightarrow{(\texttt{CANDistance,2,1})} \texttt{distance\_Km}$*, the edge* $\texttt{province\_CAN} \xrightarrow{(\texttt{CANDistance,2,1})} \texttt{distance\_Km}$ *must also belong to some path instance in* $cs_1$*. Therefore,* $e_2$ *would be added to the set, i.e.,* $cs_1 = \{e_1, e_2\}$*.*

**Algorithm 5.1**: *Converge*$(e_1, e_2)$: Returns the node at which path instances $e_1$ and $e_2$ converge.

```
input  : Two path instances e1 and e2
output : A graph node at which e1 and e2 converge; nil if they do not converge.
1  begin
2      if e1 =nil or e2 =nil then return nil
3      if the last node in e1 and e2 is the same (node x) then
4          y =nil
5          if labels of the last edge of e1 and e2 are the same then
6              y =Converge(e1.RemoveLast(), e2.RemoveLast())
               //RemoveLast() removes the last node and edge from a path instance
7          end
8          if y =nil then return x else return y
9      else
10         return nil
11     end
12 end
```

- $cs_2 = \{e_2\}$*: Similar to the previous one, this time we need to add* $e_1$ *to the set. Therefore,* $cs_2 = \{e_1, e_2\}$.
- $cs_3 = \{e_3\}$*: This set conforms to Definition 5.7.*

*Therefore, there are two possible consistent sets for the given request, i.e.,* $\mathcal{CS}(d) = \{\{e_1, e_2\}, \{e_3\}\}$. □

**Definition 5.8** *Two path instances* $e_1$ *and* $e_2$ *are said to* converge *at node* $v$*, if and only if the node* $v$ *is the first node that both* $e_1$ *and* $e_2$ *visit, and from which both path instances are the same. This means that from the node* $v$*, and not any previous node, the edge types and their labels are the same in both* $e_1$ *and* $e_2$*.* □

**Example 5.24** *Consider path instances 1-3 in Example 5.18. Paths 1 and 2 converge at node* `distance_Km`*. Also, paths 2 and 3 converge at node* `distance_Km`*, and so do paths 1 and 3.* □

Algorithm 5.1 describes, in a recursive manner, how it can be determined if two path instances converge at some graph node.

**Lemma 5.2** *If two eligible path instances* $e_1$ *and* $e_2$ *belonging to the consistent eligible path instance set cs converge at node* $v$*, either*

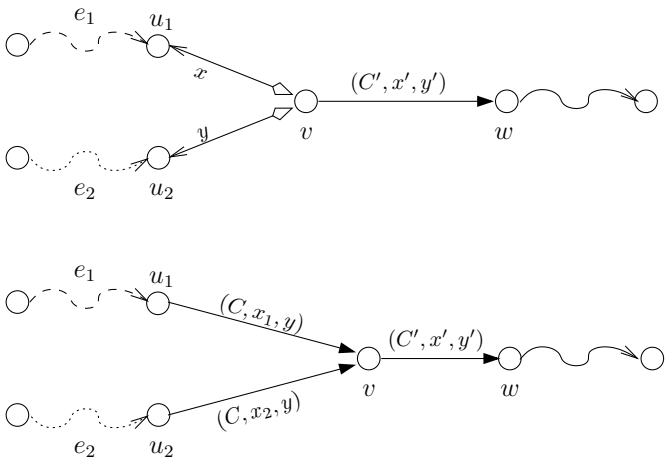



Figure 5.9: Possible ways two eligible path instances can converge in the consistent eligible path instance set $cs$.

- *they reach the node $v$ by a CD edge of type $\leftrightarrow\!\diamond$, or*
- *$e_1$ reaches $v$ by an edge $\xrightarrow{(C_1,x_1,y_1)}$ and $e_2$ reaches $v$ by an edge $\xrightarrow{(C_2,x_2,y_2)}$ where $C_1 = C_2$ and $y_1 = y_2$.*

**Proof.** Figure 5.9 shows these two cases. The important thing is that the cardinality of data type $v$ must be preserved when the path instances converge. In the top picture, for each instance of $v$, $x$ instances of $u_1$ and $y$ instances of $u_2$ are necessary. These $x$ and $y$ instances produce one instance of $v$, not two. In the bottom picture, the component $C$ converts $x_1$ instances of $u_1$ and $x_2$ instances of $u_2$ into $y$ instances of $v$. If the components on these two edges were different, i.e., $(C_1, x_1, y_1)$ and $(C_2, x_2, y_2)$, $y_1 + y_2$ instances of $v$ would have been created and the cardinality for the rest of the path would have changed inconsistently. Therefore, the same component name must appear on each edge and the number of created $v$ instances must also be the same. We could easily see that the cardinality would not be preserved with other types of graph edges. □

The following lemma is the direct result of Lemma 5.2.

**Lemma 5.3** *If two edges $u_1 \xrightarrow{(C_1,x_1,y_1)} v$ and $u_2 \xrightarrow{(C_2,x_2,y_2)} v$ ($C_1 \neq C_2$ or $y_1 \neq y_2$) belong to two eligible path instances converging at $v$ in an eligible path instance set, that set would not be consistent.* □

Consider a dependency $d \in \kappa_G^1$ with $\mathcal{P}(d) = \{p_1, p_2, \cdots, p_{|\mathcal{P}|}\}$, where each partial dependency $p_k$ $(1 \leqslant k \leqslant |\mathcal{P}|)$ has an eligible path instance set $\mathcal{E}_k = \mathcal{E}(p_k) =$

$\{e_1^k, e_2^k, \cdots, e_{|\mathcal{E}_k|}^k\}$. Based on the above discussion, from each $\mathcal{E}_k$ $(1 \leqslant k \leqslant |\mathcal{P}|)$ at least one $e_l^k$ $(1 \leqslant l \leqslant |\mathcal{E}_k|)$ must be in each consistent set $cs$ for $d$. The best way to find all possible consistent eligible path instance sets $\mathcal{CS}(d)$, according to Lemmas 5.2 and 5.3, is shown in Algorithm 5.2. Note that, to simplify the discussions, we use Observation 5.1 to unify CD and dependency edges.

To explain the algorithm, note that $\mathcal{CS}$ is supposed to hold every possible consistent eligible path instance set of the dependency $d \in \kappa_G^1$. The algorithm finds the result, through the main loop (lines 3-23), by considering every possible path instance set and then checking its consistency. At each execution of this loop, the algorithm first considers a new possible path instance set $cs$ (line 6). If this set is already part of a consistent set in $\mathcal{CS}$, the algorithm continues with the next possible path instance set (line 7). Otherwise, the consistency of every edge $u \xrightarrow{(C,x,y)} v$ from each path instance $e_i$ in $cs$ is checked (lines 9-15); i.e., the algorithm looks for all the related graph edges $w \xrightarrow{(C,z,y)} v$ and makes sure they all belong to some eligible path instance $e_j \in \mathcal{E}_j$ that converges with $e_i$ at $v$ (Lemma 5.2). It adds all such path instances to $cs$ (line 10). If there is one such edge which is not part of any eligible path instance in $cs$, the set is tagged as being inconsistent (lines 11-14) and the algorithm continues with the next eligible path instance set (through line 16). After all necessary eligible path instances are added to $cs$, the algorithm looks for any two eligible path instances $e_i$ and $e_j$ in $cs$ converging at node $v$ with edges $u_1 \xrightarrow{(C_1,x_1,y_1)} v$ and $u_2 \xrightarrow{(C_2,x_2,y_2)} v$ (line 19). As this indicates an inconsistency in the selected eligible path instances (Lemma 5.3), the algorithm marks the set as being inconsistent (lines 20). If both these consistency checks turn out to be successful, the set $cs$ is added to $\mathcal{CS}$ as a consistent eligible path instance set (line 22). At the end, when all possible path instance sets are processed, the set $\mathcal{CS}$ is returned (line 24).

The reason behind the path instance set expansion in line 10 is that if the edge $u \xrightarrow{(C,x,y)} v$ is in $cs$ and there is an edge $w \xrightarrow{(C,z,y)} v$ not in $cs$, the component $C$ cannot be used, because it would need to trigger all such edges at the same time. If one of these edges is missing in $cs$, none of them would be triggered by $C$. This expansion does not violate the eligibility of any of the path instances involved since the eligibility is determined for each path independently. However, we need to make sure that these path instances as a collection do not violate the required consistency in terms of the cardinality of data type instances involved.

For any two eligible path instances $e_1$ and $e_2$, the inconsistency of cardinalities could happen only at their common nodes. Here we study all possible cases:

**Algorithm 5.2**: *Find*$\mathcal{CS}(d)$: Returns all the consistent eligible path instance sets for $d \in \kappa_G^1$.

**input** : dependency graph $DG$, dependency $d \in \kappa_G^1$ with
$\mathcal{P} = \mathcal{P}(d) = \{p_1, p_2, \cdots, p_{|\mathcal{P}|}\}$, where for each $p_k$,
$\mathcal{E}_k = \mathcal{E}(p_k) = \{e_1^k, e_2^k, \cdots, e_{|\mathcal{E}_k|}^k\}$ $(1 \leqslant k \leqslant |\mathcal{P}|)$

**output** : all the possible assignments for $\mathcal{CS}(d)$ in $\mathcal{CS}$

**begin**
  $\mathcal{CS} = \emptyset$ // $\mathcal{CS}$ is the set of all consistent eligible path instance sets for $d$
  **foreach** $(e_1, e_2, \cdots, e_{|\mathcal{P}|}) \in (\mathcal{E}_1 \times \mathcal{E}_2 \times \cdots \times \mathcal{E}_{|\mathcal{P}|})$ **do**
    $cs = \emptyset$ // $cs$ is the current path instance set being investigated
    $bConsistent = true$
    **foreach** *eligible path instance* $e_i$ *in* $(e_1, e_2, \cdots, e_{|\mathcal{P}|})$ **do** $cs = cs \cup \{e_i\}$
    **if** *there is a consistent set* $cs' \in \mathcal{CS}$ *where* $cs \subseteq cs'$ **then continue**
    **foreach** *eligible path instance* $e_i \in cs$ **do**
      **foreach** *edge* $u \xrightarrow{(C,x,y)} v$ *in* $e_i$ **do**
        **foreach** *edge* $w \xrightarrow{(C,z,y)} v$ *in DG belonging to the eligible path instance* $e_j \in \mathcal{E}_j$ *where* $e_j \notin cs$ *and* $v = Converge(e_i, e_j)$ **do**
          $cs = cs \cup \{e_j\}$
        **if** *there is an edge* $w \xrightarrow{(C,z,y)} v$ *in DG not belonging to any path instance* $e_j \in cs$ *where* $v = Converge(e_i, e_j)$ **then**
          $bConsistent = false$
          **break**
        **end**
      **end**
      **if** $bConsistent = false$ **then break**
    **end**
    **if** $bConsistent = false$ **then continue**
    **if** *there there are two edges* $u_1 \xrightarrow{(C_1,x_1,y_1)} v$ *and* $u_2 \xrightarrow{(C_2,x_2,y_2)} v$ *(*$C_1 \neq C_2$ *or* $y_1 \neq y_2$*) in two eligible path instances* $e_i$ *and* $e_j$ *in cs where* $v = Converge(e_i, e_j)$ **then**
      $bConsistent = false$
    **end**
    **if** $bConsistent$ **then** $\mathcal{CS} = \mathcal{CS} \cup \{cs\}$
  **end**
  **return** $\mathcal{CS}$
**end**

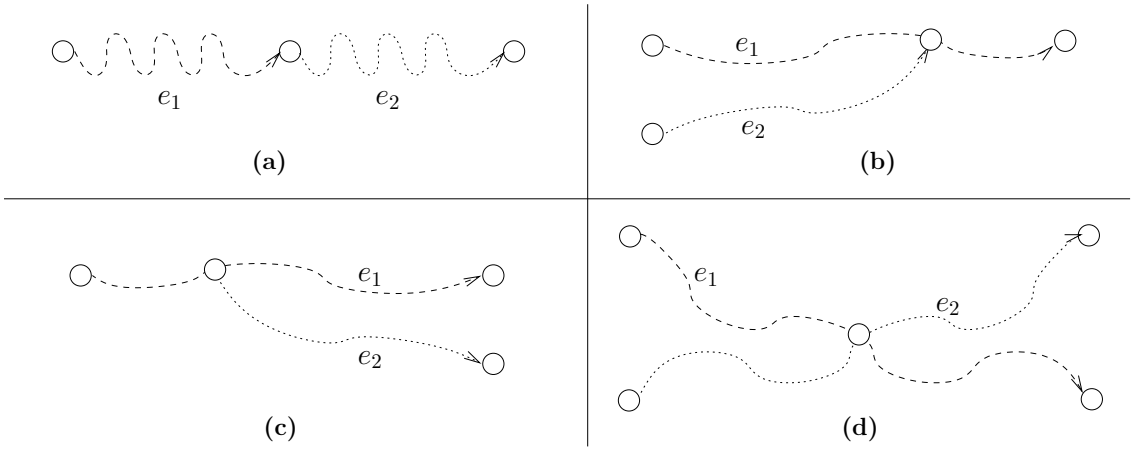



Figure 5.10: Possible shared nodes between two eligible path instances where the shared node is neither the start node in both nor the end node in both.

- $e_1$ and $e_2$ start at the same node $u$, i.e., $u \xrightarrow{(C_1,x_1,y_1)} v_1$ in $e_1$ and $u \xrightarrow{(C_2,x_2,y_2)} v_2$ in $e_2$: Since $e_1$ and $e_2$ are eligible path instances, $x_1 = x_2 = x$ instances of $u$ are in $I_G$. Because inputs can be shared between two components (a single data type instances can be fed to two components), $C_1$ and $C_2$ share $x$ instances of $u$ and no inconsistency happens.

- $e_1$ and $e_2$ converge at the same node, i.e., $u \xrightarrow{(C_1,x_1,y_1)} w$ in $e_1$ and $v \xrightarrow{(C_2,x_2,y_2)} w$ in $e_2$: Similar to the case above, $y_1 = y_2 = y$ instances of $w$ is in $O_G$. However, if $C_1 \neq C_2$, each component would create $y$ instances of $w$, and at the end, $2y$ instances of $w$ would be created, which is a violation of the cardinality constraint. However, if $C_1 = C_2$, this inconsistency would not happen as only $y$ instances of $w$ would be created.

- $e_1$ and $e_2$ share a node, but none of the above cases applies, i.e., one of the cases shown in Figure 5.10 applies: In all these cases, the shared node does not violate any cardinality constraint. In Figure 5.10-(a), where it is an end node (output) in one path instance and a start node (input) in the other, the cardinality of inputs and outputs do not affect each other. In Figure 5.10-(b)-(d), it is an intermediate node in at least one of the path instances, in which case its cardinality does not affect the cardinality of inputs and outputs.

Therefore, only when both path instances converge at a shared node, an inconsistency would happen if the labels of their last edges do not indicate the same

component. This part of the consistency checking is performed in lines 19-21 of Algorithm 5.2.

**Lemma 5.4** *Algorithm 5.2 works correctly. In other words, if there is a consistent eligible path instance set for $d \in \kappa_G^1$, it would belong to the set $\mathcal{CS}$ after the execution of the algorithm. Also, all the sets returned by the algorithm in $\mathcal{CS}$ are consistent.*

**Proof.** Let us consider that the dependency $d \in \kappa_G^1$ is given, against the dependency graph $DG$, with $\mathcal{P} = \mathcal{P}(d) = \{p_1, p_2, \cdots, p_{|\mathcal{P}|}\}$, where for each partial dependency $p_k$, $\mathcal{E}_k = \mathcal{E}(p_k) = \{e_1^k, e_2^k, \cdots, e_{|\mathcal{E}_k|}^k\}$ $(1 \leqslant k \leqslant |\mathcal{P}|)$.

We first prove that if there is a consistent eligible path instance set $cs$, then $cs \in \mathcal{CS}$. Based on the definition, $cs$ must have at least one eligible path instance from each $\mathcal{E}_k$, i.e., $\forall k : 1 \leqslant k \leqslant |\mathcal{P}| \centerdot cs \cap \mathcal{E}_k \neq \emptyset$. Therefore, the size of $cs$ is at least $|\mathcal{P}|$. Two cases are possible:

1. $|cs| = |\mathcal{P}|$: In this case $cs = \{e_1, e_2, \cdots, e_{|\mathcal{P}|}\}$, where $e_k \in \mathcal{E}_k$ $(1 \leqslant k \leqslant |\mathcal{P}|)$. Based on the consistency requirements, for each edge $u \xrightarrow{(C,x,y)} v$ in some $e_k$, there is no dependency graph edge $w \xrightarrow{(C,z,y)} v$ which does not belong to some path instance in $cs$. Also, there are no two edges $u_1 \xrightarrow{(C_1,x_1,y_1)} v$ and $u_2 \xrightarrow{(C_2,x_2,y_2)} v$ in any pair of path instances in $cs$ where $C_1 \neq C_2$ or $y_1 \neq y_2$. We can see that, by running Algorithm 5.2, at some point the above path instance set $cs$ would be considered, and since both consistency requirements hold for this set, no other path instances would be added to $cs$. Hence, $cs$ would be marked as a consistent eligible path instance set.

2. $|cs| > |\mathcal{P}|$: We pick one eligible path instance $e_k$ belonging to both $cs$ and $\mathcal{E}_k$ (for each $1 \leqslant k \leqslant |\mathcal{P}|$) and put them in the set $es$. The algorithm must consider this path instance set at some point (since it has only one path instance from each $\mathcal{E}_k$). Since $|cs| > |\mathcal{P}|$, there is a set $\mathcal{E}_j$ $(1 \leqslant j \leqslant |\mathcal{P}|)$ from which more than one path instance, say two, are in $cs$. Since these two path instances correspond to the same partial dependency, they must start and end at the same nodes. Therefore, they converge at some node along their path (the last node on their path at the latest). Hence, according to Lemma 5.2, the edges right before the converge node must indicate the same component. This is what the algorithm does in line 10.

Therefore, each consistent eligible path instance set would be captured by the algorithm.

Now, let us assume the set $\mathcal{CS}$ returned by the algorithm. In order to show that each set in $\mathcal{CS}$ is consistent, assume that there is a set $s \in \mathcal{CS}$ which is inconsistent. Two cases are possible:

1. There is a path instance in $s$ which is not eligible: This is a contradiction since all the path instances added to $s$ by the algorithm are from eligible path instance sets.

2. Some cardinality constraint is not met: The only reason for this inconsistency would be $s$ including two edges $u_1 \xrightarrow{(C_1,x_1,y_1)} v$ and $u_2 \xrightarrow{(C_2,x_2,y_2)} v$, where $C_1 \neq C_2$ or $y_1 \neq y_2$, in some path instances $e_i$ and $e_j$. Either $e_i$ and $e_j$ have been added to $s$ at line 6 of the algorithm, or one of them (say $e_j$) has been added at line 10 while the other ($e_i$) had already been in $s$. In the former case, $s$ would have been rejected at lines 19-21; and in the latter case, $e_j$ could not have been added at line 10 because it violated the condition. So the cardinality constraints cannot violated.

Therefore, each set in $\mathcal{CS}$ returned by the algorithm is consistent. □

For each consistent eligible path instance set $cs$ in $\mathcal{CS}(d)$, the set of potential components $\mathcal{C}(d)$ includes every component that appears in some edge label of some path instance in $cs$.

**Example 5.25** *Example 5.23 briefly captures the functionality of Algorithm 5.2. It returns* $\mathcal{CS}(d = \texttt{phoneNo\_CAN}^2 \rightarrow \texttt{distance\_Mile}) = \{cs_1, cs_2\}$*, where* $cs_1 = \{e_1, e_2\}$ *and* $cs_2 = \{e_3\}$*, where* $e_1$*,* $e_2$ *and* $e_3$ *were listed in Example 5.18, respectively. By considering these path instances, we conclude that*

- $\mathcal{C}_1(d) = \{$*PhoneNoLocation*, *CANDistance*, *Km2MileConvertor*$\}$
- $\mathcal{C}_2(d) = \{$*PhoneNoLocation*, *CANAddressDecomp*, *ZipCodeDistance*, *Km2MileConvertor*$\}$

*where* $\mathcal{C}_i(d)$ *is the set of potential components for dependency* $d$ *according to the consistent eligible path instance set* $cs_i$*.* □

In the next section we find out how many times we need to use each potential component and how we need to execute them such that the overall exposed behavior satisfies the given request.

### 5.2.4 Finding the Composition Plan

Considering the request $G$ given as

- $I_G = \{i_1^{k_1}, i_2^{k_2}, \cdots, i_m^{k_m}\}$
- $O_G = \{o_1^{l_1}, o_2^{l_2}, \cdots, o_n^{l_n}\}$
- $\kappa_G = \{\kappa_G^1\}$, where $\kappa_G^1 = \{d = I_G \rightarrow O_G\}$

the following are what we achieved in the previous section:

- $\mathcal{P}(d)$, the set of partial dependencies for the dependency $d \in \kappa_G^1$,
- $\mathcal{E}(p)$, the set of eligible path instances for every partial dependency $p \in \mathcal{P}(d)$,
- $\mathcal{CS}(d) = \{cs_1, cs_2, \cdots\}$, the set of all consistent eligible path instance sets for $d \in \kappa_G^1$,
- $\mathcal{C}(d) = \{\mathcal{C}_1(d), \mathcal{C}_2(d), \cdots\}$, the set of all potential component sets for dependency $d \in \kappa_G^1$ according to the consistent eligible path instance sets in $\mathcal{CS}(d)$.

The only question that remains is how the components we found in the sets $\mathcal{C}(d)$ should be executed so that the composition provides the desired behavior. The answer to this question would be a composition plan in terms of the components involved, their order of execution, and possible dataflow among them.

We saw the definition of convergence in the last section. Here, we define the concept of divergence in a similar way.

**Definition 5.9** *Two path instances $e_1$ and $e_2$* diverge *at node $v$, if and only if the node $v$ is the last node that both $e_1$ and $e_2$ visit, and until which both path instances are the same. This means that until reaching the node $v$, and not any succeeding node, the edge types and their labels are the same in both $e_1$ and $e_2$.* □

For example, paths 1 and 2 in Example 5.18 diverge at node ***address_CAN***, and so do pairs 1 and 3, and also 2 and 3. Similar to Algorithm 5.1, Algorithm 5.3 finds a node at which two path instances diverge.

Note that there is no similar lemma for divergence as Lemma 5.2 for convergence. Different components might appear on outgoing edges from the diverging node $v$, because it is assumed that each instance of a data type can be used by different

**Algorithm 5.3**: *Diverge*$(e_1, e_2)$: Returns the node at which path instances $e_1$ and $e_2$ diverge.

```
input  : Two path instances e1 and e2
output : A graph node at which e1 and e2 diverge; nil if they do not diverge.
1  begin
2      if e1 =nil or e2 =nil then return nil
3      if the first node in e1 and e2 is the same (node x) then
4          y =nil
5          if labels of the first edge of e1 and e2 are the same then
6              y =Diverge(e1.RemoveFirst(), e2.RemoveFirst())
               //RemoveFirst() removes the first node and edge from a path instance
7          end
8          if y =nil then return x else return y
9      else
10         return nil
11     end
12 end
```

components as an input. So if there are two different components on the outgoing edges from $v$ in two eligible path instances where each component needs only one $v$ instance, producing only one instance would suffice and the two components can share that instance as their input.

In order to find the composition plan for a dependency $d \in \kappa_G^1$, we first consider one of its consistent eligible path instance sets $cs$. Then for each eligible path instance $e \in cs$ we find out how many times each component in this path instance should be used to realize the corresponding partial dependency. In fact, we have already found these numbers in the last section when we were checking the eligibility of path instances using the numeric equations of Definition 5.6. In that definition, numbers $n_1$, $n_2$, ..., $n_m$ respectively indicate how many times components $C_1$, $C_2$, ..., $C_m$ should be used.

Now consider a consistent eligible path instance set $cs$ including only one path instance $e$, which would look like $i \xrightarrow{(C_1,x_1,y_1)} u_1 \xrightarrow{(C_2,x_2,y_2)} u_2 \cdots u_{m-1} \xrightarrow{(C_m,x_m,y_m)} o$. Moreover, assume we have found that components $C_1$, $C_2$, ..., $C_m$ should be used $n_1$, $n_2$, ..., $n_m$ times, respectively. The composition plan, in composition algebra, that satisfies $cs$ would be

$$\overbrace{(C_1 \parallel \cdots \parallel C_1)}^{n_1 \text{ times}} \odot \overbrace{(C_2 \parallel \cdots \parallel C_2)}^{n_2 \text{ times}} \odot \cdots \odot \overbrace{(C_m \parallel \cdots \parallel C_m)}^{n_m \text{ times}}$$

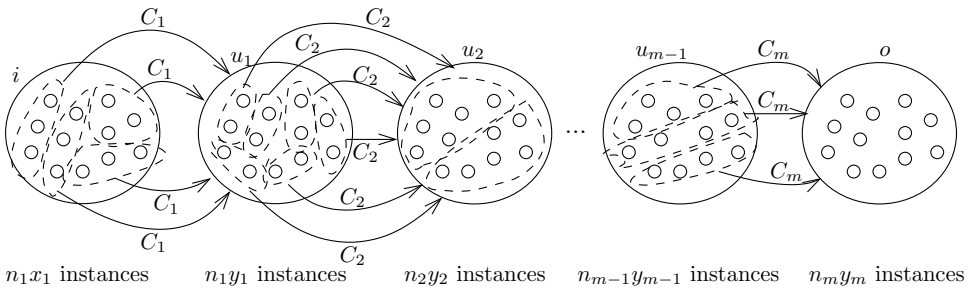



Figure 5.11: How data type instances are used or created by components in a consistent eligible path instance set of size 1.

Note that the path instance $e$ would correspond to the partial dependency $i^{n_1 \times x_1} \rightarrow o^{n_m \times y_m}$. Figure 5.11 shows how this composition plan works. At step 1, all the inputs ($n_1 \times x_1$ instances of $i$) are used by $n_1$ instances of the component $C_1$ producing $n_1 \times y_1$ instances of $u_1$. The partial composition for this step is shown as $cp_1 = \overbrace{C_1 \mid\mid \cdots \mid\mid C_1}^{n_1 \ times}$. Similarly, $n_1 \times y_1$ instances of $u_1$ would be converted to $n_2 \times y_2$ instances of $u_2$ using the parallel execution of $n_2$ instances of the component $C_2$, i.e., $cp_2 = \overbrace{C_2 \mid\mid \cdots \mid\mid C_2}^{n_2 \ times}$. Since the outputs of $cp_1$ is used by $cp_2$ as inputs, $cp_1$ and $cp_2$ are being synchronized based on those intermediate instances of $u_1$. Therefore, the partial composition plan until the creation of $u_2$ instances would be $cp_1 \odot cp_2$. Continuing this scenario we would see that the above composition plan is the satisfying plan for $cs$.

In case the consistent set $cs$ contains more than one path instances, and all those path instances neither converge nor diverge at any graph node, the above plan would work. It suffices to find the plan for each path instance as above, and then compose the results using the parallel composition. Since path instances are independent, no synchronization would be needed and their parallel execution would be sufficient.

The situations in which two eligible path instances converge or diverge at some graph node is shown in Figure 5.12. We assume that there is no other path instance in $cs$ that converges or diverges with any of these two path instances.

In part (a) two path instances $e_1$ and $e_2$ converge at node $w_1$. According to Lemma 5.2 the incoming edges of $w_1$ must have the same component in their labels (component $C_1$). Note that the number of instances of $w_1$, $w_2$, ..., $o$ in both $e_1$ and $e_2$ must be the same. This means that the number of times the components $C_1$,

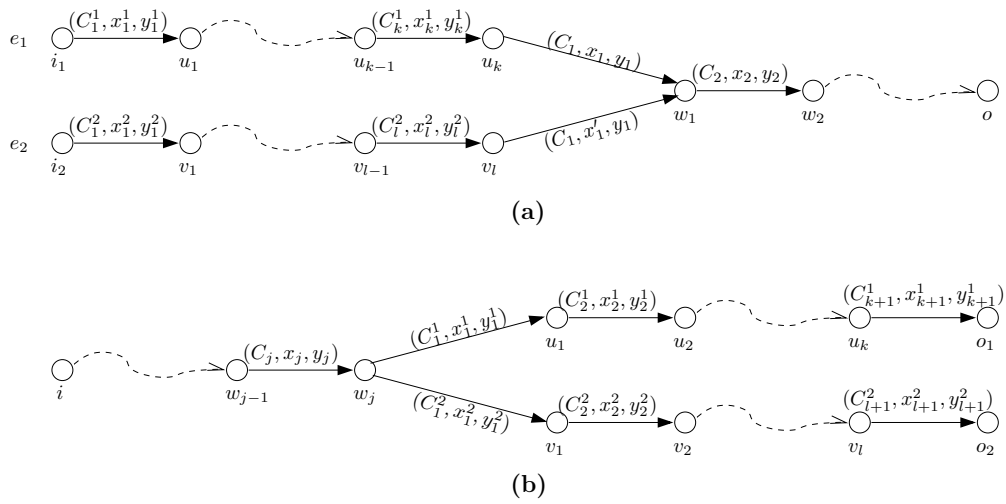



Figure 5.12: (a) Two eligible path instances converging at node $w_1$. (b) Two eligible path instances diverging at node $w_j$.

$C_2$, ... are needed would be the same according to both path instances. Assuming these numbers are $n_1$, $n_2$, ..., to satisfy both $e_1$ and $e_2$ together we also need the same number of these components, and there is no need to double $n_1$, $n_2$, ..., $n_m$ for that purpose.

The partial composition plans for the parts from $i_1$ to $u_k$, from $i_2$ to $v_l$, and from $w_1$ to $o$ can be obtained according to the above discussion. Let us assume these partial plans are $cp_1$, $cp_2$ and $cp_3$, respectively. It is obvious that $cp_1$ and $cp_2$ can run in parallel. However, both $cp_1$ and $cp_2$ must be executed before $cp_3$. According to $e_1$ and $e_2$, $cp_1$ produces $n_1 \times x_1$ instances of $u_k$, while $cp_2$ produces $n_1 \times x_1'$ instances of $v_l$. These $u_k$ and $v_l$ instances then are consumed by $n_1$ parallel execution of $C_1$ to produce $n_1 \times y_1$ instances of $w_1$, which, afterwards, are consumed by $cp_3$. This means that the composition plan for these two path instances would be $((cp_1 \parallel cp_2) \odot (\overbrace{C_1 \parallel \cdots \parallel C_1}^{n_1 \ times})) \odot cp_3$.

The composition plan for part (b) can be similarly found. The only difference is that since each input instance can be consumed by multiple components, outgoing edges of $w_j$ could have different components in their labels. If the partial composition plans for the parts from $i$ to $w_j$, from $u_1$ to $o_1$, and from $v_1$ to $o_2$ respectively are $cp_0$, $cp_1$ and $cp_2$, the composition plan for the two diverging path instances of Figure 5.12-(b) would be $(cp_0 \odot ((\overbrace{C_1^1 \parallel \cdots \parallel C_1^1}^{n_1^1 \ times}) \odot (\overbrace{C_1^2 \parallel \cdots \parallel C_1^2}^{n_1^2 \ times}))) \odot (cp_1 \parallel cp_2)$. In the special case where $C_1^1 = C_1^2$, we would have $x_1^1 = x_1^2$ and $n_1^1 = n_1^2$; and the

composition plan would be $(cp_0 \odot (\overbrace{C_1^1 \parallel \cdots \parallel C_1^1}^{n_1^1 \ times})) \odot (cp_1 \parallel cp_2)$.

There is an alternative way to find the composition plan for part (b) which is due to the fact that the restrictions are more relaxed on diverging path instances, and also the above mentioned difference (each input instance can be used by multiple components). In this alternative way, if the composition plan for two diverging path instances $e_1$ and $e_2$ (Figure 5.12-(b)) is $cp(e_1)$ and $cp(e_2)$, the composition plan for both of them would be $cp(e_1) \parallel cp(e_2)$. This way, the initial $n_1 \times x_1$ instances of $i$ are twice used by $n_1$ instances of $C_1$ for each path instance $e_1$ and $e_2$. This would continue until $2 \times n_j \times y_j$ instances of $w_j$ are created. Then half of these instances is used by $n_1^1$ instances of $C_1^1$ while the other half is used by $n_1^2$ instances of $C_1^2$. Therefore, the consistency would not be violated in this alternative solution. In the special case where $C_1^1 = C_1^2$, $n_1^1 \times y_1^1$ instances of $u_1$ and $v_1$ are created for the path instance $e_1$, and the same number of them are created for the path instance $e_2$. Since for path instance $e_1$, created $v_1$ instances are not useful, they would be ignored. Similarly, $n_1^1 \times y_1^1$ instances of $u_1$ are ignored for the path instance $e_2$.

Although the alternative solution is not optimal, as the numbers of required instances of the components on the shared part of the path instances are doubled, it is easier to achieve.

This way, the composition plan for every type of consistent eligible path instance set could be found. The set of all possible composition plans for a given dependency $d$ is shown as $\mathcal{CP}(d)$, in which there is one composition plan for each consistent eligible path instance set in $\mathcal{CS}(d)$.

**Example 5.26** *Let us continue Example 5.23 to find the corresponding composition plans for* $d =$ `phoneNo_CAN`$^2 \rightarrow$ `distance_Mile`*. In that example, we found two possible consistent eligible path instance sets, i.e.,* $cs_1 = \{e_1, e_2\}$ *and* $cs_2 = \{e_3\}$*, where* $e_1$*,* $e_2$ *and* $e_3$ *were given in Example 5.18.*

*The corresponding eligible path instances are shown in Figure 5.13. Note that in this figure CD edges have been simulated by dependency edges (Observation 5.1) and the pair of nodes at the two ends of each GenSpec edge have been combined.*

*After solving the numeric equations of Definition 5.6, the followings are the results in terms of the potential components and the number of times they should be used for each eligible path instance:*

- *Path instance* $e_1$*:* $n_1^1 = 2$ *instances of* `PhoneNoLocation`*,* $n_2^1 = 2$ *instances of*

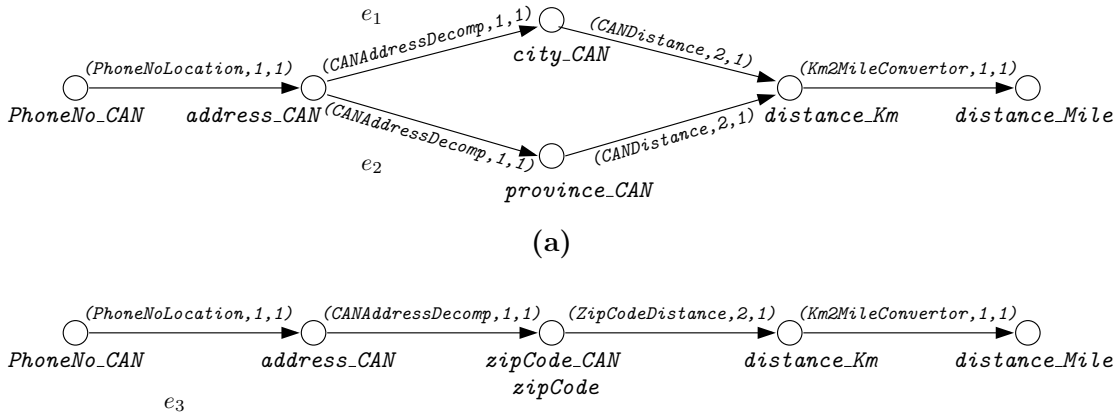



Figure 5.13: The corresponding path instances for the request of Example 5.26. (a) Path instances for $cs_1$. (b) The path instance for $cs_2$.

*CANAddressDecomp*, $n_3^1 = 1$ *instance of* *CANDistance* *and* $n_4^1 = 1$ *instance of* *Km2MileConvertor*.

- *Path instance* $e_2$*:* $n_1^2 = 2$ *instances of* *PhoneNoLocation*, $n_2^2 = 2$ *instances of* *CANAddressDecomp*, $n_3^2 = 1$ *instance of* *CANDistance* *and* $n_4^2 = 1$ *instance of* *Km2MileConvertor*.

- *Path instance* $e_3$*:* $n_1^3 = 2$ *instances of* *PhoneNoLocation*, $n_2^3 = 2$ *instances of* *CANAddressDecomp*, $n_3^3 = 1$ *instance of* *ZipCodeDistance* *and* $n_4^4 = 1$ *instance of* *Km2MileConvertor*.

*The case for* $cs_1$ *is a mixture of converging and diverging path instances. Following the above instructions, to satisfy both* $e_1$ *and* $e_2$*, the composition plan would be* (((*PhoneNoLocation* || *PhoneNoLocation*) $\odot$ (*CANAddressDecomp* || *CANAddressDecomp*)) $\odot$ *CANDistance*) $\odot$ *Km2MileConvertor*. *Composition algebraic axioms prove that this expression is equivalent to* (((*PhoneNoLocation* $\odot$ *CANAddressDecomp*) || (*PhoneNoLocation* $\odot$ *CANAddressDecomp*)) $\odot$ *CANDistance*) $\odot$ *Km2MileConvertor*. *The case for* $cs_2$ *is quite simple to resolve. According to the corresponding instructions for a consistent set of size 1, the composition plan would be* (((*PhoneNoLocation* || *PhoneNoLocation*) $\odot$ (*CANAddressDecomp* || *CANAddressDecomp*)) $\odot$ *ZipCodeDistance*) $\odot$ *Km2MileConvertor*. *So there are two possible composition plans for the given request. Of course, in both plans we would need to ignore unwanted outputs from the* *CANAdressDecomp* *component. We can use the hiding operator in the resulting algebraic expressions for this purpose.*

**Algorithm 5.4**: *CompPlan*($d$): Returns all the composition plans for dependency $d$.

**input** : dependency $d = \{i_1^{k_1}, i_2^{k_2}, \cdots, i_m^{k_m}\} \rightarrow \{o_1^{l_1}, o_2^{l_2}, \cdots, o_n^{l_n}\}$
**output** : $\mathcal{CP} = \mathcal{CP}(d)$, the set of all composition plans to satisfy $d$

```
1  begin
2      CS = FindCS(d)
3      foreach cs ∈ CS do
4          foreach output data type o_i in d do
5              s_i = the set of all path instances in cs that end at o_i
6          end
7          cp = (FindComp(o_1, s_1) || FindComp(o_2, s_2) || ··· || FindComp(o_n, s_n))
           // cp is the composition plan for the current consistent set
8          CP = CP ∪ {cp}
9      end
10     return CP
11 end
```

*To validate the compositions found for both $cs_1$ and $cs_2$ we use the composition algebraic axioms. For $cs_1$,* $\texttt{PhoneNoLocation} \odot \texttt{CANAddressDecomp} \equiv \texttt{phoneNo\_CAN} \cdot (\overline{\texttt{city\_CAN}} \,||\, \overline{\texttt{province\_CAN}} \,||\, \overline{\texttt{zipCode\_CAN}} \,||\, \overline{\texttt{streetAddr}})$. *Since only two of the outputs are required,* $R_1 \equiv (\texttt{PhoneNoLocation} \odot \texttt{CANAddressDecomp}) \backslash \{\overline{\texttt{zipCode\_CAN}}, \overline{\texttt{streetAddr}}\} \equiv \texttt{phoneNo\_CAN} \cdot (\overline{\texttt{city\_CAN}} \,||\, \overline{\texttt{province\_CAN}})$. *As a consequence,* $(R_1 \,||\, R_1) \odot \texttt{CANDistance} \equiv (\texttt{phoneNo\_CAN} \,||\, \texttt{phoneNo\_CAN}) \cdot \overline{\texttt{distance\_Km}}$. *This result, if synchronized with the component* `Km2MileConvertor`*, proves that the proposed solution performs as requested; i.e., it receives two instances of* `phoneNo_CAN` *and returns one instance of* `distance_Mile`*. Composition algebraic rules can be similarly used to prove that the plan found for $cs_2$ is also valid.* □

Algorithms 5.4 and 5.5 show, in a more formal way, how the composition plan of a given dependency can be determined. Algorithm 5.4 simply states that the composition plan for $d$ is the parallel execution of all the composition plans for each output in $d$. The reason behind this is that we can safely assume that each output in $d$ is produced independently. Therefore, the dependency $d$ can be decomposed into $n$ dependencies $d_1 = \{i_1^{k_1}, i_2^{k_2}, \cdots, i_m^{k_m}\} \rightarrow o_1^{l_1}$, $d_2 = \{i_1^{k_1}, i_2^{k_2}, \cdots, i_m^{k_m}\} \rightarrow o_2^{l_2}$, ..., $d_n = \{i_1^{k_1}, i_2^{k_2}, \cdots, i_m^{k_m}\} \rightarrow o_n^{l_n}$. In fact, two sets $\{d\}$ and $\{d_1, d_2, \cdots, d_n\}$ are considered behaviorally equivalent. This algorithm also finds parts of the consistent eligible path instance set which correspond to each of the decomposed dependencies. These parts are put into sets $s_1$, $s_2$, ..., $s_n$ accordingly.

**Algorithm 5.5**: *FindComp*$(o, s)$: Returns the composition plan for a single output type $o$ according to the set of path instances $s$.

**input** : an output type $o$ and a set of path instances $s$
**output** : the composition plan to produce $o$ according to $s$

1 **begin**
2 $C =$*LastComp(e)*, where $e \in s$ //*LastComp(e)* is the component on the last edge of $e$
3 $n =$*LastNo(e)*, where $e \in s$ //*LastNo(e)* is the number of times $C$ should be used
4 $PrevNodes = \{u |\ u \rightarrow o \text{ is in some } e_i \in s, where\ |e_i| > 1\}$
5 **if** $PrevNodes = \emptyset$ **then return** $(\overbrace{C \ ||\ \cdots\ ||\ C}^{n\ times})$
6 **foreach** $i$ *such that* $1 \leqslant i \leqslant |PrevNodes|$ **do**
7 $s_i = \{e_i.RemoveLast() |\ e_i \in s \text{ and } |e_i| > 1 \text{ and } u_i \rightarrow o \text{ in } e_i\}$
8 $p_i = FindComp(u_i, s_i)$
9 **end**
10 **if** $|PrevNodes| = 1$ **then return** $(p_1) \odot (\overbrace{C \ ||\ \cdots\ ||\ C}^{n\ times})$
11 **else return** $(p_1 \ ||\ p_2 \ ||\ \cdots\ ||\ p_{|PrevNodes|}) \odot (\overbrace{C \ ||\ \cdots\ ||\ C}^{n\ times})$
12 **end**

The main processing in finding the composition plan is performed by Algorithm 5.5, in which it is shown how the composition plan is found for a decomposed dependency (like $d_1$, $d_2$, ..., $d_n$ above) according to its related subset of the consistent eligible path instance set. It is assumed that for each related path instance (in $s$) the result of the equations in Definition 5.6 is known and accessible. The algorithm starts from the output node $o$ in this decomposed dependency and goes all the way back towards the involved inputs through the path instances in $s$. Note that every pair of path instances in $s$ must converge at one of their shared nodes. Since this node would be $o$ at the latest, we conclude that all the path instances in $s$ point to the same component $C$ in their last edge (line 2). The number of times ($n$) this component should be used according to each path instance in $s$, which was already determined, must be the same as well (line 3). Then, the nodes which appear right before $o$ in path instances of length more than one in $s$ are put in the set *PrevNodes* (line 4). This set being empty means that all path instances in $s$ are of length one and, therefore, $n$ times running $C$ in parallel would be the desired composition plan (line 5). If *PrevNodes*$\neq \emptyset$, for each node $u_i$ in the set *PrevNodes* the corresponding composition plan $p_i$ is found recursively (lines 6-9). Based

on the discussion we had before (about Figure 5.12) the appropriate composition plan would be the parallel execution of all the $p_i$'s synchronized with the parallel execution of $n$ instances of $C$ (lines 10-14).

### 5.2.5 Complexity

To find the overall complexity of the proposed solution for the generic composition planning problem, we study the complexity of Algorithms 5.2, 5.5 and 5.4 separately. Basically, we need to find the complexity of these algorithms in finding appropriate composition plans for a dependency of the form $d = \{i_1^{k_1}, i_2^{k_2}, \cdots, i_m^{k_m}\} \rightarrow o^n$ against the dependency graph $DG = (V, E)$. More complex dependencies can be decomposed into simpler dependencies like $d$ whose corresponding composition plans can be found in parallel.

#### Complexity of Algorithm 5.2

Before discussing the complexity of this algorithm we need to study

- the complexity of finding eligible path instances $\mathcal{E}(p)$ for a partial dependency $p \in \mathcal{P}(d)$, and
- the complexity of Algorithm 5.1. i.e., $Converge(e_1, e_2)$.

The partial dependency $p$ would look like $i_j^{k_j} \rightarrow o^n$ $(1 \leqslant j \leqslant m)$. To find its eligible path instances, the first thing to do is finding all the graph paths from the node $i_j$ to the node $o$. This part is proved to be an NP-Complete problem [36]. To find this, we can start by finding all paths of length 1, then all paths of length 2, ..., and finally, all paths of length $|V| - 1$. We exclude all possible graph loops in this search. Therefore, the worst-case complexity would be $1 + \sum_{i=2}^{|V|-1} \prod_{j=2}^{i} (|V| - j) = O((|V| - 2)!)$, as the maximum number of paths between two graph nodes.

For each path that is found from $i_j$ to $o$ all the possible path instances should be analyzed using the equations of Definition 5.6. The number of path instances of a given graph path is highly dependent on its length and the number of triplets in its edge labels. The number of triplets in the label of the edge $u \rightarrow v$ is equal to the number of repository components that have $u$ in their inputs, and $v$ in their outputs. If the repository is rich enough, this number can be ignored against the

number of repository components ($N$). However, in the worst case this number would be $N$. Therefore, each graph path would have $O(N^{|V|-1})$ path instances in the worst case.

As mentioned above, in the worst case, the number of graph paths for each partial dependency $p$ is $O((|V|-2)!)$. Therefore, the number of path instances for $p$ would be $O((|V|-2)! \times N^{|V|-1})$.[7] According to Definition 5.6, it takes $O(|V|-1)$ to find if a path instance is eligible. Therefore, the worst-case complexity of finding eligible path instances for $p$, $\mathcal{E}(p)$, would be $O((|V|-1)! \times N^{|V|-1})$. Also, $|\mathcal{E}(p)|$ could be as big as $O((|V|-2)! \times N^{|V|-1})$, meaning that all possible path instances found for a partial dependency could be eligible.

We can see that, in order to find the eligible path instances for all partial dependencies, the running time complexity would be $O(|V|! \times N^{|V|-1})$.[8] However, if we consider the disk space needed for finding eligible path instances, we would have a linear complexity, i.e., $O(N + |V| + |E|)$, because they are examined one by one. In none of the intermediate steps we would need a space more than this. For example, for finding graph paths between two nodes, only $O(|V|-1)$ space would suffice, because we find paths one after the other. Also, for considering each path instance, again $O(|V|-1)$ space would be enough. To store the eligible path instances, in the worst case we would have to store $O((|V|-2)! \times N^{|V|-1})$ of them, requiring $O((|V|-1)! \times N^{|V|-1})$ space.

As a result, finding eligible path instances for a partial dependency takes exponential time in the worst case, due to existing an exponential number of graph paths between two nodes.

The complexity of finding out if two path instances $e_1$ and $e_2$ converge at some graph node, $\mathit{Converge}(e_1, e_2)$, is $O(min(|e_1|, |e_2|))$, in which $|e_1|$ and $|e_2|$ are the lengths of $e_1$ and $e_2$, respectively. Therefore, we can say that its complexity is $O(|E|)$.

Now, let us consider Algorithm 5.2. According to the above discussion, there are $O((|V|-2)! \times N^{|V|-1})$ eligible path instances for each partial dependency. Since $d$ has $m = O(|V|)$ partial dependencies, the main loop (lines 3-23) would be executed

[7]Note that the dependency graph might contain loops and there might be an infinite number of paths between a pair of nodes. This would be considered a trade-off between the performance and the correctness, as improving one might complicate the other. We believe that some heuristics could be used to find out if the loops are going to be helpful in finding eligible path instances. The discussion on these heuristics is beyond the scope of this thesis.

[8]Note that $m = O(|V|)$

$O((|V|-2)!^{|V|} \times N^{|V|^2})$ times. The worst-case complexity of different parts of the algorithm in each execution of the main loop is given below:

- lines 4,5: $O(1)$
- line 6: $O(|V|)$
- line 7: $O(|V| \times (|V|-2)!^{|V|} \times N^{|V|^2})$, since each execution of the loop could lead to a new consistent set which means $|\mathcal{CS}| = O((|V|-2)!^{|V|} \times N^{|V|^2})$.
- Loop at lines 8-17: $O(|V|)$
  - Loop at lines 9-15: $O(|V|-1)$
    - line 10: $O(N^2 \times |V||E|)$, where $O(N|V|)$ is for considering each possible edge instance ending at $v$, $O(N)$ is for checking if the current edge instance belongs to any eligible path instance (assuming that the eligible path instances are stored efficiently for this search), and $O(|E|)$ is for checking the convergence.
    - lines 11-14: $O(N^2 \times |V||E|)$, similar to the previous one.
  - line 16: $O(1)$
- line 18: $O(1)$
- lines 19-21: $O(|V|^3|E|)$, where $O(|V|)$ is for checking each node, $O(|V|^2)$ is for checking each pair of involved path instances, and $O(E)$ is for checking the convergence.
- line 22: $O(1)$

Then, the overall running time complexity of the algorithm in the worst cast, assuming $|E| = O(|V|^2)$, would be $O(N^{|V|}(N(|V|-2)!)^{2|V|})$. Note that, regarding the space complexity, during the execution of the algorithm, only $O(|V|^2)$ space is required for storing the current set of eligible path instances.

### Complexity of Algorithm 5.5

Note that in this algorithm $|s| = O(V)$. Assuming that $m$ is the size of longest path instance in $s$, and the worst-cast complexity of the algorithm is $f(m)$, its different parts would have the following worst-case complexities:

- lines 2-3: $O(1)$
- line 4: $O(|V||E|)$, where $O(|V|)$ is for each different eligible path instance in $s$, and $O(|E|)$ is for checking all its edges.
- line 5: $O(1)$
- Loop at lines 6-9: for each node is *PrevNodes*, it finds the corresponding path instances according to $s$ to call the algorithm recursively on each of them. Clearly, $\sum_{j=1}^{|PrevNodes|} |s_j| = O(|V|)$. In the worst case, the loop is executed $O(|V|)$ times, with
  - line 7: $O(1)$
  - line 8: $f(m-1)$
- lines 10-11: $O(1)$

Therefore, assuming that $|E| = O(|V|^2)$, the worst-cast running time complexity would be $f(|V|) = |V|^3 + |V|f(|V|-1)$, which results in $O(|V|^{|V|})$ complexity (note that $m = O(|V|)$. Regarding the space complexity, we can easily see that by efficiently storing the path instances, the space required can be calculated by the recursive equation $f(|V|) = |V| + f(|V|-1)$, showing that an $O(|V|^2)$ space would be enough. In the recursive formula, $f(|V|)$ represents the execution time when the maximum path instance size in $s$ is $|V|$.

### Complexity of Algorithm 5.4

This algorithm runs Algorithm 5.2 and then calls Algorithm 5.5 once for each different output type in $d$ and each consistent eligible path instance set. The running time of the first part is $O(N^{|V|}|V|(N(|V|-2)!)^{2|V|})$,[9] and that of the second part is $O(|\mathcal{CS}| \times |V||V|^{|V|}) = O((|V|-2)!^{|V|} \times N^{|V|^2}|V|^{|V|+1})$ in the worst case. Then, the overall running time complexity of the graph-based approach, in the worst case, is estimated to be $O(N^{|V|}|V|(N(|V|-2)!)^{2|V|})$. This worst-case complexity corresponds to finding all possible consistent sets, and therefore, all possible composition plans. It is quite easy to see that the worst-case complexity for finding the first

[9]Note that in finding the complexity of Algorithm 5.2 we assumed that there is only one output in the request. Therefore, a coefficient $|V|$ has to be considered for the worst-case running time complexity when dependencies in general are being studied.

possible composition plan would be the same. The reason is that in the worst case all possible eligible path instance sets have to be examined.

This algorithm would need $O(|V|^2)$ in terms of the space required, where $O(|V|^2)$ is for both storing a consistent set and the space needed for the calls to the previous algorithm. Here, we ignore the space required for storing the resulting composition plans. In general, because storing all the possible eligible path instances for each partial dependency needs $O(|V|! \times N^{|V|-1})$ space in the worst case, the worst-case space complexity of the graph-based approach would also be exponential.

## 5.3 Discussion

The dependency graph represented in this chapter has already been used by other researchers for web services composition. Specifically, Shin and Lee extend the dependency graph of this chapter to capture functional semantics of the repository components [95]. They make a little change in the structure of the dependency graph by creating graph nodes for repository components as well and claim that their model performs better for solving the composition plan. However, what they claim seems to be unrealistic, because even in their model the cardinality constraints must be met and consistent eligible path instance sets have to be found (similar to what we discussed in Section 5.2.3), and they do not emphasize these two aspects in their composition algorithm, which is too abstract. Therefore, the complexity of the problem does not change by introducing graph nodes for repository components. But this example shows the potential of graph models such as the dependency graph in finding appropriate approaches for component or web service composition.

# Chapter 6

# A Reasoning-Based Approach to Component Composition

In Chapter 5 we proposed a solution to the generic composition planning problem that was based on a dependency graph. We discussed why the proposed approach was exponential, in the worst case, in the size of this graph. This approach was presented using algorithms to find appropriate components for solving the problem, as well as finding valid composition plans. In this chapter, which is based on a second publication [50], we present another approach to solve the composition planning problem, based on a reasoning technique in first order logic called the forward chaining approach. We start the chapter by reviewing the necessary background, and then we present the solution.

## 6.1 Background on Logical Reasoning

In simple terms, logical reasoning is defined as the formal manipulation of symbols representing a collection of believed propositions to produce representations of new ones. These symbols are used to represent knowledge and also to infer it through some known rules. The collection of believed propositions is called the *knowledge base*. The type of logical reasoning we apply in composition planning is *logical inference*, in which the final result is considered to be a conclusion of the initial propositions. For example, if the knowledge base contains two propositions "patient $x$ is allergic to medication $m$" and "anyone allergic to medication $m$ is also allergic to medication $m'$", using logical inference we can conclude that "patient $x$ is allergic to medication $m'$".

In the reasoning problem we study in this chapter, there is a knowledge base $S$ containing the known propositions and a goal proposition $G$. What we expect as a result is whether $G$ can be inferred from the propositions in $S$, written as $S \models G$. It is trivial that to prove $S \models G$ is equivalent to prove that $S \cup \{\neg G\}$ is unsatisfiable. In other words, if it turns out that $S \cup \{\neg G\}$ is satisfiable, we conclude that $S \not\models G$; and if $S \cup \{\neg G\}$ is unsatisfiable, then $S \models G$. It is assumed that there are no contradicting propositions in $S$, which means $S \not\models \mathit{FALSE}$. If $S \cup \{\neg G\}$ is unsatisfiable, or $S$ *entails* $G$ ($S \models G$), sometimes we might also wish to know how $G$ is derived from $S$, i.e., which propositions from $S$ and in what order are applied to result in $G$.

The reasoning algorithm and its complexity depend on the expressivity of the underlying logic. Specifically, the way the knowledge is represented is a determining factor in how we reason about it. We would expect to reason simpler in propositional logic rather than in first-order predicate logic. Unfortunately, the logical reasoning even for propositional logic, as a non-parametric and simpler form of logic, is NP-Hard in the worst case. However, there is a less expressive form of logic, *Horn clauses*, that comes with less reasoning complexity [20].

A Horn clause is a Disjunctive Normal Form clause in first order logic with at most one positive literal. For example, the clause $\neg \mathit{child} \vee \neg \mathit{male} \vee \mathit{boy}$ is a Horn clause with two negative and one positive literals. Since Horn clauses have no more than one positive literal they can be converted into a unique implication clause involving only positive literals, as the above clause is equivalent to $\mathit{child} \wedge \mathit{male} \rightarrow \mathit{boy}$. Therefore, every implication in propositional logic clause with a conjunction of positive literals in its left-hand side and at most one positive literal in its right-hand side is a Horn clause [20].

In order to reason about Horn clauses we need to see how we can infer a new clause from two Horn clauses. As the main inference rule, $a_1 \wedge \cdots \wedge a_m \rightarrow b_1$ and $b_1 \wedge \cdots \wedge b_n \rightarrow c_1$ imply $a_1 \wedge \cdots \wedge a_m \wedge b_2 \wedge \cdots \wedge b_n \rightarrow c_1$. This inference rule is used to solve the reasoning problem for Horn clauses. To do so, we start with a knowledge base $S$ and, by resolving Horn clauses from $S$ and the intermediate clauses created through the reasoning process, we try to prove a goal clause $G$. If this is successful, we say that $G$ can be derived from $S$, and show it by $S \vdash G$.

A restricted but sufficient form of resolution is the SLD resolution. An SLD resolution starts with resolving two clauses from the knowledge base $S$ and in every intermediate step the result of the last step is resolved with another clause from $S$. Therefore, two intermediate results cannot be resolved in an SLD resolution. The

**Algorithm 6.1**: The SLD forward chaining procedure [20].

**input** : a finite list of literals $q_1, \cdots, q_n$

**output** : YES or NO according to whether a knowledge base $S$ entails all $q_i$'s

```
1 begin
2     if every goal q_i is marked as solved then return YES
3     check if there is a clause p_1 ∧ ··· ∧ p_m → p in S, such that all the literals
      p_1, ··· , p_m are marked as solved, while p is not marked as solved
4     if there is such a clause then
5         mark p as solved and go to line 2
6     else
7         return NO
8     end
9 end
```

derivation continues until $G$ is proved to be either true or false.

There are two main SLD techniques for reasoning about propositional Horn clauses: *backward chaining* and *forward chaining*. The backward chaining procedure, which starts from the goal and goes all the way back to reach the clauses from the knowledge base, has two drawbacks; it might go into an infinite loop, or it might take exponential time to terminate. The forward chaining approach, on the other hand, is much more reliable and efficient as it always terminates and also performs the reasoning in linear time in the number of clauses. Algorithm 6.1 represents the forward chaining procedure. We use this procedure in the next section to provide a solution for the composition planning problem [20].

## 6.2 Composition Planning: Initial Proposition

We described in Chapter 4 how actions in composition algebra are modeled by non-parametric names. Since we focus on stateless components in this thesis, we can assume that in the composition algebra the underlying process of all these components is of the general form $P \equiv (i_1 \mid\mid \cdots \mid\mid i_m) \cdot (\overline{o_1} \mid\mid \cdots \mid\mid \overline{o_n})$, which correctly captures the intended behavior, i.e., receiving some inputs without any specific order, and then returning some outputs in a similar way. To solve the composition planning problem using the reasoning techniques, in this section we discuss our initial solution for the composition using the forward chaining procedure.

The algebraic representation of the behavior of stateless components has an

**Algorithm 6.2**: The modified version of the forward chaining algorithm [49].

**input** : a repository $S$ and a goal $G : I_G \rightarrow O_G$ with $O_G = \{o_1^G, \cdots, o_{n_G}^G\}$
**output** : YES or NO according to whether $S \cup I_G$ entails all the literals in $O_G$

1 **begin**
2  mark all the literals in $I_G$ as true
3  **if** *every literal in* $O_G$ *is marked as true* **then** return YES
4  check if there is a clause in $S$ such that all of its left-hand-side literals are marked as true, and there is at least one literal in its right-hand-side which is not marked as true
5  **if** *there is such a clause* **then**
6  add it to the list of clauses used so far, and mark all the unmarked literals on its right-hand-side as true and go to line 3
7  **else**
8  return NO
9  **end**
10 **end**

alternative representation in propositional logic, as it can be modeled by the implication $P : i_1 \wedge \cdots \wedge i_m \rightarrow o_1 \wedge \cdots \wedge o_n$. For example, the web service $\mathtt{CityStateByZip}$ is specified in composition algebra by the algebraic expression $\mathtt{CityStateByZip} = \mathtt{zipCode} \cdot (\overline{\mathtt{city}} \,||\, \overline{\mathtt{state}})$. Alternatively, it can be described in propositional logic using the implication $\mathtt{CityStateByZip} : \mathtt{zipCode} \rightarrow \mathtt{city} \wedge \mathtt{state}$. As a result, the composition planning problem can be expressed as follows.

**Problem 6.1** *There is a repository $S$ which contains a set of available components of the form $C : I \rightarrow O$, where $I = i_1 \wedge \cdots \wedge i_m$ and $O = o_1 \wedge \cdots \wedge o_n$. There is also a target component $G : I_G \rightarrow O_G$, with $I_G = i_1^G \wedge \cdots \wedge i_{m_G}^G$ and $O_G = o_1^G \wedge \cdots \wedge o_{n_G}^G$. The question is whether $S \vDash G$. If so, the corresponding derivation (composition plan) is also required.* □

We can provided a procedure based on Algorithm 6.1 to solve the above problem [49]. To do so, we make a little change in the above problem so that we can apply the forward chaining algorithm. The idea is to add $i_1^G, \ldots, i_{m_G}^G$ as known facts to the repository $S$, and try to prove that $o_1^G, \ldots, o_{n_G}^G$ hold. Algorithm 6.2 represents the proposed procedure. This algorithm runs in linear time in the size of the repository as well, as it is quite similar to the forward chaining algorithm in terms of the steps taken. The composition plan can be obtained based on the order of using knowledge base clauses in the algorithm.

**Example 6.1** *Let us assume a repository of available components containing:*

- $C_1 = \mathtt{address} \cdot \overline{\mathtt{zipCode}}$
- $C_2 = (\mathtt{name} \,||\, \mathtt{birthDate}) \cdot \overline{\mathtt{localMap}}$
- $C_3 = (\mathtt{address} \,||\, \mathtt{zipCode}) \cdot \overline{\mathtt{localMap}}$
- $C_4 = (\mathtt{name} \,||\, \mathtt{birthDate}) \cdot (\overline{\mathtt{zipCode}} \,||\, \overline{\mathtt{birthPlace}})$
- $C_5 = (\mathtt{sIN} \,||\, \mathtt{name}) \cdot \overline{\mathtt{address}}$
- $C_6 = (\mathtt{sIN} \,||\, \mathtt{birthDate} \,||\, \mathtt{birthPlace}) \cdot \overline{\mathtt{phoneNo}}$
- $C_7 = (\mathtt{sIN} \,||\, \mathtt{birthDate} \,||\, \mathtt{zipCode}) \cdot \overline{\mathtt{phoneNo}}$

*Given the target component* $G = (\mathtt{sIN} \,||\, \mathtt{name} \,||\, \mathtt{birthDate}) \cdot (\overline{\mathtt{localMap}} \,||\, \overline{\mathtt{phoneNo}})$*, we are interested to know whether* $G$ *can be built by composing some of these components.*

*We convert the component specifications into the propositional logic format and apply Algorithm 6.2 in order to find a solution. Therefore, the knowledge base* $S$ *would contain clauses* $C_1, \ldots,$ *and* $C_7$*, where*

- $C_1 : \mathtt{address} \rightarrow \mathtt{zipCode}$
- $C_2 : \mathtt{name} \wedge \mathtt{birthDate} \rightarrow \mathtt{localMap}$
- $C_3 : \mathtt{address} \wedge \mathtt{zipCode} \rightarrow \mathtt{localMap}$
- $C_4 : \mathtt{name} \wedge \mathtt{birthDate} \rightarrow \mathtt{zipCode} \wedge \mathtt{birthPlace}$
- $C_5 : \mathtt{sIN} \wedge \mathtt{name} \rightarrow \mathtt{address}$
- $C_6 : \mathtt{sIN} \wedge \mathtt{birthDate} \wedge \mathtt{birthPlace} \rightarrow \mathtt{phoneNo}$
- $C_7 : \mathtt{sIN} \wedge \mathtt{birthDate} \wedge \mathtt{zipCode} \rightarrow \mathtt{phoneNo}$

*The request would be reformatted to* $G : \mathtt{sIN} \wedge \mathtt{name} \wedge \mathtt{birthDate} \rightarrow \mathtt{localMap} \wedge \mathtt{phoneNo}$*. To find a solution, we start by marking* $\mathtt{sIN}$*,* $\mathtt{name}$ *and* $\mathtt{birthDate}$ *as true. If there are different choices from* $S$ *to use in line 4 of the algorithm, we can randomly pick one. Following the algorithm, we see that picking* $C_2$*,* $C_4$ *and* $C_6$ *in the same order is one solution*[1]*. To validate, we find the result of* $(C_2 \odot C_4) \odot C_6$ *in composition algebra:*

$$
\begin{aligned}
C_2 \odot C_4 &\equiv ((\mathit{name} \parallel \mathit{birthDate}) \cdot \overline{\mathit{localMap}}) \\
&\quad \odot ((\mathit{name} \parallel \mathit{birthDate}) \cdot (\overline{\mathit{zipCode}} \parallel \overline{\mathit{birthPlace}})) \\
&\equiv (\mathit{name} \parallel \mathit{birthDate}) \cdot (\overline{\mathit{localMap}} \parallel \overline{\mathit{zipCode}} \parallel \overline{\mathit{birthPlace}}) \\
(C_2 \odot C_4) \odot C_6 &\equiv (\mathit{name} \parallel \mathit{birthDate}) \cdot (\overline{\mathit{localMap}} \parallel \overline{\mathit{zipCode}} \parallel \overline{\mathit{birthPlace}}) \\
&\quad \odot (\mathit{sIN} \parallel \mathit{birthDate} \parallel \mathit{birthPlace}) \cdot \overline{\mathit{phoneNo}} \\
&\equiv (\mathit{sIN} \parallel \mathit{name} \parallel \mathit{birthDate}) \cdot (\overline{\mathit{localMap}} \parallel \overline{\mathit{zipCode}}) \cdot \overline{\mathit{phoneNo}} \\
((C_2 \odot C_4) \odot C_6) \backslash \{\overline{\mathit{zipCode}}\} &\equiv {}^{1}(\mathit{sIN} \parallel \mathit{name} \parallel \mathit{birthDate}) \cdot \overline{\mathit{localMap}} \cdot \overline{\mathit{phoneNo}}
\end{aligned}
$$

*We realize that the result is slightly different from the specification of $G$; because the outputs are not generated in parallel in the proposed composition. We explain this small difference as follows.*

- *Normally, the parallel operator in such requests means that the relative order of inputs and outputs is not important; and as long as inputs are taken and then outputs are produced based on them, the result is acceptable. This way, we can accept the composition $(C_2 \odot C_4) \odot C_6$ as an approximate solution[2] to the request $(\mathit{sIN} \parallel \mathit{name} \parallel \mathit{birthDate}) \cdot (\overline{\mathit{localMap}} \parallel \overline{\mathit{phoneNo}})$.*
- *If the above result is not acceptable, and we need to produce the exact parallel expression, we may assume that the published composite component would take care of this ordering. In other words, it works as a wrapper around all*

[1] We assume that unwanted generated outputs can be ignored.

[2] Formally speaking, two algebraic expressions $e_1 \equiv I_1 \cdot O_1$ and $e_2 \equiv I_2 \cdot O_2$, in which

- $I_1$ and $I_2$ contain only input actions, and moreover, contain the same number of input actions of each type,
- $O_1$ and $O_2$ contain only output actions, and moreover, contain the same number of output actions of each type,

are *approximately equivalent*, shown as $e_1 \cong e_2$, if one or both of the followings hold:

- For some expression $I_0$, either $I_1 \equiv I_2 \oplus I_0$ $(I_0 \not\equiv I_2)$ or $I_2 \equiv I_1 \oplus I_0$ $(I_0 \not\equiv I_1)$.
- For some expression $O_0$, either $O_1 \equiv O_2 \oplus O_0$ $(O_0 \not\equiv O_2)$ or $O_2 \equiv O_1 \oplus O_0$ $(O_0 \not\equiv O_1)$.

For example, $e_1 = a \cdot (\overline{b} \parallel \overline{c})$ is approximately equivalent to $e_2 = a \cdot \overline{b} \cdot \overline{c}$, because

- $e_1 \equiv I_1 \cdot O_1$, where $I_1 = a$ and $O_1 = \overline{b} \parallel \overline{c}$,
- $e_2 \equiv I_2 \cdot O_2$, where $I_2 = a$ and $O_2 = \overline{b} \cdot \overline{c}$,
- $I_1$ and $I_2$ contain the same input actions, and $O_1$ and $O_2$ contain the same output actions,
- $I_1 \equiv I_2$ and $O_1 \equiv O_2 \oplus \overline{c} \cdot \overline{b}$.

*the constituent components and waits to receive both* `localMap` *and* `phoneNo` *before returning them to the user.*

*In case we needed to achieve a composition that is exactly equivalent to the request $G$ and we found a solution leading to a non-equivalent expression, like the above example, we would have had to backtrack to the last choice we made and continue the algorithm with another alternative. Although this apparently would add to the complexity of the procedure, we do not go into its details in this thesis assuming that the approximate results are also acceptable.* □

Although Algorithm 6.2 takes linear time in the number of knowledge base clauses to find a solution, it does not work properly in all stateless cases. In particular, it fails to correctly capture the cardinality of involved data type instances. In other words, since there is no cardinality involved in propositional logic expressions, e.g., $a \wedge a \equiv a$, this algorithm would not be able to deal appropriately with multiple instances of the same literal.

**Example 6.2** *Let us add to the repository of the previous example a new component $C_8 = (\texttt{zipCode} \parallel \texttt{zipCode}) \cdot \overline{\texttt{distance}}$, which has an input with cardinality 2. This expression cannot be appropriately converted into a propositional logic formula, because following the above mapping we obtain $C_8 : \texttt{zipCode} \wedge \texttt{zipCode} \rightarrow \texttt{distance} \equiv \texttt{zipCode} \rightarrow \texttt{distance}$, which specifies a different behavior, i.e., $\texttt{zipCode} \cdot \overline{\texttt{distance}}$.* □

Therefore, this logical representation for the behavior of components and the corresponding algorithm do not answer the composition planning problem in general. We show in the next section how they can be improved to capture the cardinality of literals as well.

## 6.3 Composition Planning: Generic Approach

The initial proposed composition procedure is promising, and therefore, we try to apply the same ideas to capture cardinalities. In the initial approach every component behavior could be seen as $C : I \rightarrow O$, in which $I$ and $O$ are sets containing the corresponding inputs and outputs identified by their data type names; i.e., $I = \{i_1, \cdots, i_m\}$ and $O = \{o_1, \cdots, o_n\}$. Since sets are unable to represent duplicate members and cardinalities, in the new procedure we assume that inputs and

outputs are multisets of type names. To distinguish duplicate type names that might appear in these multisets, we use unique identifiers and we call them *data instances* or *instances*, where *type*($m$) is the data type of the instance identified by $m$.

In order to solve the generic version of the stateless composition planning problem there are some extra constraints that must be taken into account.

1. Each component might be used more than once in a composition. In Algorithms 6.1 and 6.2 each clause is used at most once, because when its right hand side is marked as true, there is no need to use that clause again. Since cardinality of a data type can be more than one, a component might be needed more than once. Therefore, in the new procedure, when a component is selected to participate in a composition it should not be removed from the list of available components.

2. When some inputs are used by a component, they (exact same instances) cannot be used by the same component again. This is because after those inputs are used by the component the expected result is generated, and there is no point in running the component on them again, as it will produce the same result. Algorithms 6.1 and 6.2 automatically comply with this rule as they do not use the same clause more than once. To apply this constraint, we attach to each instance $m$, that is being processed, a set *usedBy* of identifiers of the form $C^i$ indicating that the component $C$ has been applied on $m$ in step $i$ of the reasoning algorithm.

3. For every single piece of functionality, all the given input instances must be used to produce each given output instance, unless otherwise is specified by the user. For example, for the request $(\texttt{sIN} \,||\, \texttt{name} \,||\, \texttt{birthDate}) \cdot (\overline{\texttt{localMap}} \,||\, \overline{\texttt{phoneNo}})$, in producing the outputs $\texttt{localMap}$ and $\texttt{phoneNo}$ all the three inputs must be involved. To comply with this constraint, which is not considered in Algorithm 6.2, in generating outputs through the composition algorithm we need to determine whether all the inputs have been used. Therefore, we attach to each instance $m$ a set *uses* containing instances from $I_G$ that have been used so far to generate $m$. If $uses(m) = I_G$ for some instance $m$, we conclude that all the inputs in $I_G$ have participated in producing $m$.

4. In order to find the composition plan in case the algorithm returns a 'YES', we keep a set *createdBy* for every instance $m$, which indicates the component

that has produced $m$. This way we can find the appropriate components from the repository that are used in each step.

Algorithm 6.3 represents the improved reasoning-based procedure for composition planning. It returns a 'YES/NO' response based on whether the given request can be built by the repository components or not.

In this algorithm, the set $M$ is the pool of instances that are already produced as the algorithm execution progresses (line 2). To every instance $m$ that is added to $M$ we assign

- *type*($m$) as its data type,
- *usedBy*($m$) as a set containing components that have used this instance so far,
- *uses*($m$) as a set of instances from $I_G$ that have been used, directly or indirectly, in producing $m$,
- *from*($m$) as a set containing instances from $M$ used directly to produce $m$,
- *createdBy*($m$) as the repository component that produced $m$,
- *createdAtStep*($m$) as a number which shows at which step $m$ is created, and
- *level*($m$) as a number which will be explained later.

In the beginning an instance is added to $M$ for every data type in $I_G$ (lines 3-12). We also use a step counter to record the components used in each step (line 13). In line 14 a test is performed to see if we have achieved a valid composition for the request $G$. A valid composition would produce an instance of the same type for every data type in $O_G$ (to satisfy the cardinality constraint), where in producing each of them the whole $I_G$ is used. If the test fails, we need to continue with a new component to produce more instances (line 15). We do so by finding some instances in $M$ that can be used by a component $C$, and have not been used by that component before. If there is such a component (line 16), $C^n$ (component $C$ at step $n$) is added to the set *usedBy* of each of those instances (line 17), and for each output $o$ of $C$ a new instance $m'$ is added to $M$ with the appropriate *type*, *usedBy*, *uses*, *from*, *createdBy*, *createdAtStep* and *level* values (lines 18-28). In particular,

- *type*($m'$) would be $o$;

**Algorithm 6.3**: The improved procedure for composition planning feasibility.

**input** : a repository $S$ of components of the form $C : I_C \rightarrow O_C$ where $I_C$ and $O_C$ are multisets of data types; and similarly, a request $G : I_G \rightarrow O_G$.

**output** : YES/NO according to whether there is a valid composition from $S$ for $G$.

1 **begin**
2  define $M$ as the set of instances, and set $M = \emptyset$
3  **foreach** *data type* $i$ *in* $I_G$ **do**
4  create a new instance $m$ in $M$
5  set $type(m) = i$
6  set $usedBy(m) = \emptyset$
7  set $uses(m) = \emptyset$
8  set $from(m) = \emptyset$
9  set $createdBy(m) = \emptyset$
10  set $createdAtStep(m) = 0$
11  set $level(m) = 0$
12  **end**
13  use a step counter $n$, and set $n = 1$
14  **if** *there is a set* $K \subseteq M$*, such that* $TYPE(K) = O_G$ ***and*** *for every instance* $m \in K$*,* uses$(m) = I_G$ **then** return YES $//\, TYPE(K) = \{type(m) | m \in K\}$
15  check if there is a component $C : I_C \rightarrow O_C$ in $S$ and a set $L \subseteq M$, such that $TYPE(L) = I_C$ **and** $C^k \notin \bigcap_{m \in L} usedBy(m)$ for all step $k$
16  **if** *there is such a component* **then**
17  **foreach** $m \in L$ **do** $usedBy(m) = usedBy(m) \cup \{C^n\}$
18  **foreach** $o$ *in* $O_C$ **do**
19  create a new instance $m'$ in $M$
20  set $type(m') = o$
21  set $usedBy(m') = \emptyset$
22  set $uses(m') = (I_C \cap I_G) \cup (\bigcup_{m \in L} uses(m))$
23  set $from(m') = L$
24  set $createdBy(m') = C$
25  set $createdAtStep(m') = n$
26  set $level(m') = 1 + Max_{m \in L}(level(m))$
27  **if** level$(m') > |S|$ **then** return NO
28  **end**
29  $n = n + 1$
30  go to line 14
31  **else**
32  return NO
33  **end**
34 **end**

- $usedBy(m')$ would be empty;

- *uses*($m'$) would contain all the instances from $I_G$ that are in $I_C$ too, plus all the instances from $I_G$ that have participated in producing instances in $L$;
- *from*($m'$) would obviously contain the instances in $L$;
- *createdBy*($m'$) would be the component $C$;
- *createdAtStep*($m'$) would be $n$.

Moreover, the step counter increases (line 29) and the algorithm continues to check if the goal is already satisfied (line 30). If there is no new component from the repository to use instances in $M$, the algorithm terminates with a negative result (lines 31-32).

We can consider the execution of the algorithm as building a multi-level graph, in which nodes at each level represent instances from $M$ and edges represent components from the repository that created those instances. In this structure, there is an edge with label $C^n$ from an instance $i$ at level $l_i$ to an instance $j$ at level $l_j$ ($l_j > l_i$), if and only if there is a component $C : I_C \rightarrow O_C$ used in step $n$ of the algorithm, such that *type*($i$) $\in I_C$ and *type*($j$) $\in O_C$ and for each other data type $t$ in $I_C$ there is an instance $k$ at some level $l$ ($l < l_j$) with *type*($k$) $= t$. Moreover, at least one instance of one of the data types in $I_C$ must have appeared at level $l_j - 1$. In other words, if the maximum level of instances in *from*($m$) is $l$, $m$ would sit at level $l + 1$. Let $min(n_l)$ and $max(n_l)$ denote the minimum and maximum of the set $\{n|$ there is an edge $i \xrightarrow{C^n} j$ such that $l_j = l\}$. In order to create the levels of this graph in a breadth-first order, we assume that $l_i < l_j \Leftrightarrow min(n_{l_j}) > max(n_{l_i})$. This assumption guarantees that all the possible instances at each level are created before creating instances at the next level. To start creating instances, we put instances corresponding to the initial inputs in $I_G$ at level 0. Then intermediate instances generated by the algorithm sit at the next levels. The *level* attribute attached to instances in Algorithm 6.3 is used for this purpose:

- at line 11, *level*($m$) is set to 0 for each instance corresponding to an input in $I_G$,
- at line 26, *level*($m'$) is set to one plus the maximum level of all instances in $L$, according to the discussion above.
- at line 27, the current graph level is checked to terminate the algorithm if necessary. Lemma 6.1 explains this.

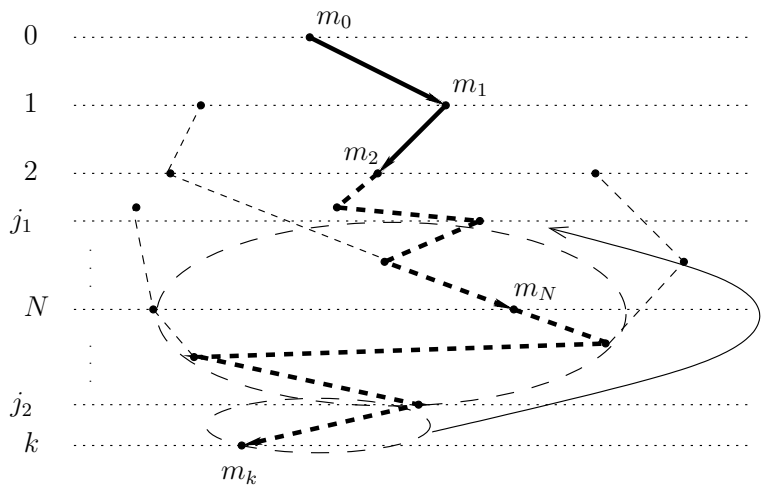



Figure 6.1: The case where the solution is found in a level after level $N$.

**Lemma 6.1** *If there is a solution for a given composition planning problem, Algorithm 6.3 is able to find it; and if there is none, it terminates.*

**Proof.** According to the above breadth-first method in applying the components from the repository $S$ and creating new instances, it is guaranteed that the algorithm will find a solution, if there is any, because all the instances leading to a valid composition would be at some level of this graph and finally would be reached by the algorithm.

To show that the algorithm terminates if there is no solution, we prove that if there are solutions for the given request, the first one must be found by level $N$ of the multi-level graph, where $N$ is the number of components. Suppose that the first solution for a given request is found at level $k$ ($k > N$), as $m_k$ in Figure 6.1. Since, for each intermediate instance $m$ in level $j$, at least one member of $from(m)$ resides in level $j-1$, we can conclude that there is at least one path of length $k$ from level 0 leading to $m_k$ (the bold path in Figure 6.1). Therefore, at least one repository component $C$ has been involved (at least) twice in creating $m_k$ resulting in instances at levels $j_1$ and $j_2$. This means that the part of this path between levels $j_1$ and $j_2$ can be collapsed, because the part of the path from level $j_2$ to level $k$ could have started at level $j_1$ (note that data types created by $C$ have been the same in both times) cutting the path length to $k-(j_2-j_1)$. This would continue until there is no repository component involved more than once in the path, in which case the path size would be less than or equal to $N$. □

As mentioned before, Algorithm 6.3 returns only a 'YES/NO' answer, and does

**Algorithm 6.4**: *FindComposition*($K$) returns the corresponding composition plan after Algorithm 6.3 returns a 'YES'.

**input** : a set $K$, returned by Algorithm 6.3, of instances that correspond to the outputs of the given request

**output** : the root of an expression tree structure which indicates the appropriate composition

```
1  begin
2      find the largest set of instances N ⊆ K created by different components or at different
       steps
3      if |N| = 1 then return FindComposition(m)  //N = {m}
4      pick the first instance m in N and set N = n − {m}
5      leftNode=FindComposition(m)
6      rightNode=FindComposition(N)
7      if leftNode=nil and rightNode=nil then return nil
8      else if leftNode=nil then return rightNode
9      else if rightNode=nil then return leftNode
10     else
11         rootNode.leftChild=leftNode
12         leftNode.parent=rootNode
13         rootNode.rightChild=rightNode
14         rightNode.parent=rootNode
15         rootNode.expression=(leftNode.expression) || (rightNode.expression)
16         return rootNode
17     end
18 end
```

not return the actual composition in case the answer is a 'YES'. In order to find the composition, we take advantage of the information stored along with instances in $M$ and also the set $K$ that was found right before the algorithm terminated (line 11). We can start by instances in $K$ and go back step by step to the inputs and components that created them until we reach the original inputs, i.e., the instances corresponding to $I_G$. The above multi-level graph could be used to find the actual composition through its paths from level 0 to the desired outputs. Algorithms 6.4 and 6.5 show how this can be done.

Algorithm 6.4 finds a valid composition plan for a set of instances $K$ corresponding to the outputs in $I_G$. The composition plan is given in terms of an expression tree which indicates the involved components to achieve the desired behavior.

The general procedure is to create output instances in parallel, due to the assumption we made earlier that outputs are considered independent of each other. Since some of the instances in $K$ might be created by the same component at the

**Algorithm 6.5**: *FindComposition*($m$) returns the corresponding composition for a single instance.

```
   input  : a single instance m which corresponds to one of the outputs of the given request
   output : the root of an expression tree indicating the appropriate composition for that
            instance
 1 begin
 2     if m.level=0 then return nil  //m ∈ I_G
 3     leftNode=FindComposition(from(m))
 4     if leftNode=nil then
 5         rootNode.leftChild=rootNode.rightChild=nil
 6         rootNode.expression=createdBy(m)
 7         return rootNode
 8     end
 9     rootNode.leftChild=leftNode
10     leftNode.parent=rootNode
11     rightNode.expression=createdBy(m)
12     rightNode.leftChild=rightNode.rightChild=nil
13     rootNode.rightChild=rightNode
14     rightNode.parent=rootNode
15     rootNode.expression=(leftNode.expression) ⊙ (rightNode.expression)
16     return rootNode
17 end
```

same step, the algorithm finds all such sets of instances at line 2. It chooses the largest possible set $N$ in such a way that not any two instances in $N$ are created by the same component at the same step. If $N$ contains only one instance $m$, then the composition that leads to $m$ is the result (line 3). Otherwise, one instance $m$ is removed from $N$ and the appropriate compositions for $m$ and also the reduced $N$ are found recursively in *leftNode* and *rightNode*, respectively (lines 4-6). Special cases where one or both of these results return null are handled in lines 7-9. In general, the results found for $m$ and the reduced $N$ need to be executed in parallel (line 15). So, a *rootNode* is created to show this parallel execution, where its left and right children are *leftNode* and *rightNode*, respectively (lines 11-14). Then, this *rootNode* is returned as the composition plan for $K$.

As opposed to Algorithm 6.4, which finds the appropriate composition plan for a set of instances, Algorithm 6.5 finds the composition plan for a single instance. In fact, Algorithm 6.4 is narrowed down to multiple executions of Algorithm 6.5.

Algorithm 6.5 receives an instance $m$ and finds the composition plan for $m$ by returning the root of the corresponding execution tree. It first checks the level of instance $m$, and returns null if it is at level 0 of the multi-level graph (line 2). If

an instance is at level 0, it is one of the input instances corresponding to $I_G$ and, therefore, it requires no composition. At the next step, we need to find a valid composition plan for the instances in *from*($m$) and put it in *leftNode*. If the result is null, it means that $m$ has been directly created from the input instances, and no intermediate instances are involved. In this case, a single node is created which indicates that *createdBy*($m$) is the composition plan for $m$ (lines 4-8). It is obvious that the instances in *from*($m$) are used by component *createdBy*($m$) to create $m$. So, if *leftNode* is not null, its corresponding composition plan has to be synchronized with *createdBy*($m$). For this purpose, a *rootNode* is created which has *leftNode* as its left child and *createdBy*($m$) as its right child, and represents the synchronization of the two (lines 9-15). This *rootNode* is then returned as the composition plan for $m$ (line 16). We go through a brief example to clarify this graph structure and the whole approach in more details.

**Example 6.3** *Consider the following components forming the repository:*

- $C_1$ : *name* $\rightarrow$ *email*
- $C_2$ : *name* $\rightarrow$ *phone*
- $C_3$ : *phone* $\rightarrow$ *zipCode*
- $C_4$ : *zipCode*$^2$ $\rightarrow$ *distance*
- $C_5$ : *phone* $\rightarrow$ *address*
- $C_6$ : *zipCode* $\rightarrow$ *city*
- $C_7$ : *address* $\rightarrow$ *zipCode*
- $C_8$ : *name* $\rightarrow$ *cell*

*The superscript* 2 *in* $C_4$ *indicates that this component takes two instances of* ***zipCode*** *as inputs. We explain the algorithm using the multi-level graph in Figure 6.2. In this graph, the set* usedBy *for each instance is the set of its outgoing edge labels. The set* uses *for each data instance is the multiset of the data types of all the instance at the top level of this graph which have a path to that specific instance. The set* from *for each instance is the set of its parents in the graph. Values for* createdBy *and* createdAtStep *are shown as the incoming edge label of the instance. And finally, the* level *of an instances would be the length of the longest path from some*

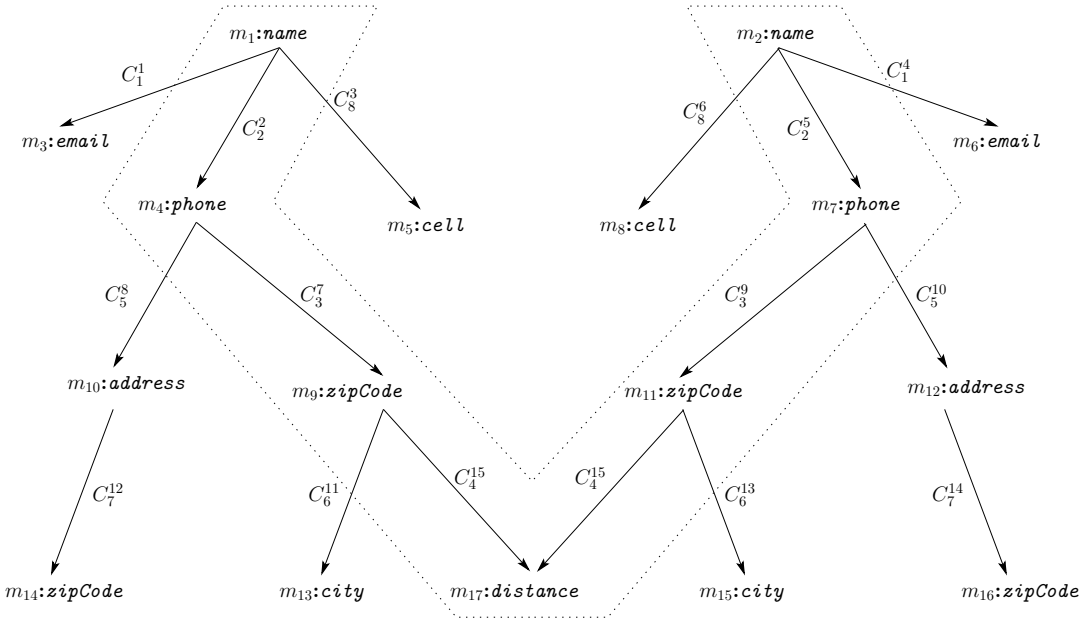



Figure 6.2: The result of Algorithm 6.3 for Example 6.3.

*node at level 0 to that instance. The section of the graph which is inside the dotted area represents the data instances leading to the 'YES' response in this example.*

*Given* $G$ : **name**$^2$ $\rightarrow$ **distance**, *Algorithm 6.3 starts by adding* $m_1$ *and* $m_2$ *of type* **name** *to* $M$. *Each of these instances produces one instance of* **email**, **phone** *and* **cell** *at level 1 in the next six steps (*$m_3$ *to* $m_8$ *by components* $C_1$, $C_2$ *and* $C_8$*). Since there is no solution yet, in the next four steps, each of the two* **phone** *instances that are currently in* $M$ *(*$m_4$ *and* $m_7$*) produces one instance of* **zipCode** *and* **address** *(*$m_9$ *to* $m_{12}$ *by components* $C_3$ *and* $C_5$*) at level 2. Again, there is no* **distance** *instance in* $M$ *up to this point. Continuing the algorithm, each* **zipCode** *instance produces one* **city** *instance (*$m_{13}$ *and* $m_{15}$ *by component* $C_6$*) at level 3, and each* **address** *instance creates one* **zipCode** *instance (*$m_{14}$ *and* $m_{16}$ *by component* $C_7$*), also at level 3. Then, two initial* **zipCode** *instances (*$m_9$ *and* $m_{11}$*) produce one* **distance** *instance (*$m_{17}$ *by component* $C_4$*) at level 3. Since in producing* $m_{17}$ *both* **name** *instances in* $I_G$ *have been used, the algorithm returns a 'YES'.*

*Now we follow Algorithms 6.4 and 6.5 to find the appropriate composition plan for* $G$. *Algorithm 6.3 returns 'YES' by finding the set* $K = \{m_{17}\}$. *This set is fed to the FindComposition method in Algorithm 6.4. Since it has only one instance in it line 3 would be executed, i.e., FindComposition*($m_{17}$). *For* $m_{17}$ *we have*

- *type*($m_{17}$) =**distance**

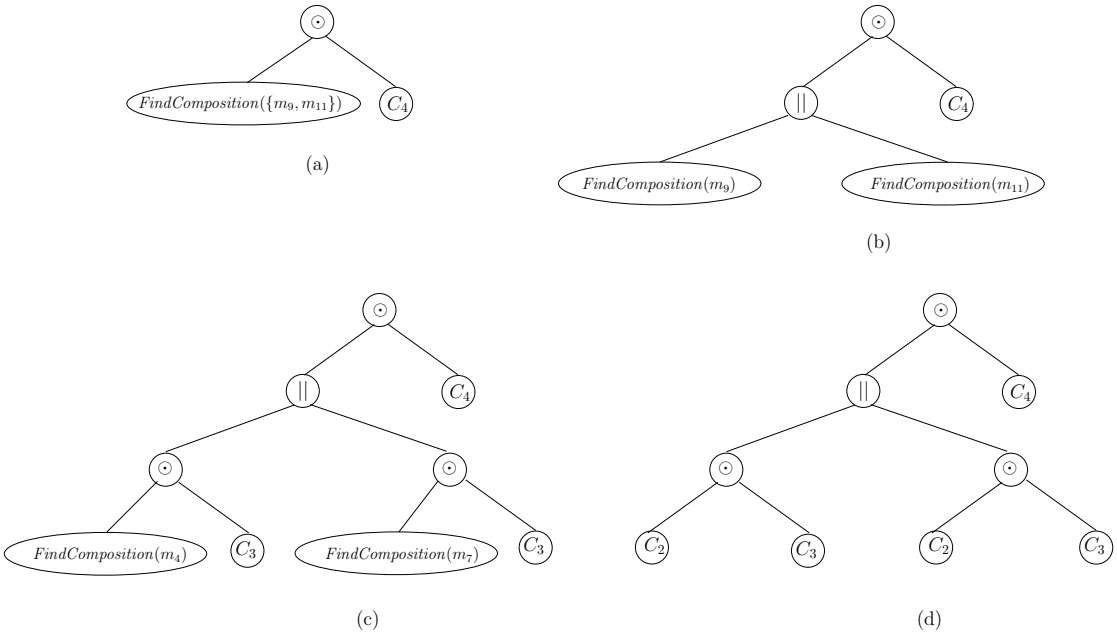



Figure 6.3: Stepwise generation of the composition plan for Example 6.3 using Algorithms 6.4 and 6.5.

- $usedBy(m_{17}) = \emptyset$
- $uses(m_{17}) = \{m_1, m_2\}$
- $from(m_{17}) = \{m_9, m_{11}\}$
- $createdBy(m_{17}) = C_4$
- $createdAtStep(m_{17}) = 15$
- $level(m_{17}) = 3$

*Algorithm 6.5 then makes a recursive call FindComposition*($\{m_9, m_{11}\}$) *to find the appropriate composition for from*($m_{17}$). *The result of this recursive call then would be synchronized with* $C_4$ *to form the desired composition plan (Figure 6.3-(a)). In the recursive call, Algorithm 6.4 calculates* $N$ *to be* $\{m_9, m_{11}\}$. *Then it picks* $m_9$ *from* $N$, *reduces* $N$ *to* $\{m_{11}\}$, *and makes the recursive calls FindComposition*($m_9$) *and FindComposition*($\{m_{11}\}$), *the latter of which would lead to the recursive call FindComposition*($m_{11}$). *The results of FindComposition for* $m_9$ *and* $m_{11}$ *would be composed in parallel to form the solution for FindCompositon*($\{m_9, m_{11}\}$) *(Figure 6.3-(b)). Considering the multi-level graph of Figure 6.2, the composition plan for* $m_9$ *and* $m_{11}$ *would be the same. So we discuss the procedure for* $m_9$

*and do the same for $m_{11}$ as we continue. Algorithm 6.5 suggests that the plan for creating $m_9$ would be synchronizing FindComposition($\{m_4\}$), which is equal to FindComposition($m_4$), and $C_3$ (Figure 6.3-(c)). To find the composition plan for $m_4$, Algorithm 6.5 first finds the plan for from($m_4$) = $\{m_1\}$, which turns out to be null because $m_1$ sits at level 0. Therefore, the result for FindComposition($m_4$) would be a single node with label $C_2$ (Figure 6.3-(d)). As a result, the appropriate composition for the request G would be $((C_2 \odot C_3) \,||\, (C_2 \odot C_3)) \odot C_4$. This composition plan can be validated using the composition algebraic rules.*

$$
\begin{aligned}
C_2 \odot C_3 &\equiv (\mathit{name} \cdot \overline{\mathit{phone}}) \odot (\mathit{phone} \cdot \overline{\mathit{zipCode}}) \\
&\equiv \mathit{name} \cdot \overline{\mathit{zipCode}} \\
(C_2 \odot C_3) \,||\, (C_2 \odot C_3) &\equiv (\mathit{name} \cdot \overline{\mathit{zipCode}}) \,||\, (\mathit{name} \cdot \overline{\mathit{zipCode}}) \\
&\equiv (\mathit{name} \cdot \mathit{name} \cdot \overline{\mathit{zipCode}} \cdot \overline{\mathit{zipCode}}) \\
&\oplus (\mathit{name} \cdot \overline{\mathit{zipCode}} \cdot \mathit{name} \cdot \overline{\mathit{zipCode}}) \\
((C_2 \odot C_3) \,||\, (C_2 \odot C_3)) \odot C_4 &\equiv ((\mathit{name} \cdot \mathit{name} \cdot \overline{\mathit{zipCode}} \cdot \overline{\mathit{zipCode}}) \\
&\oplus (\mathit{name} \cdot \overline{\mathit{zipCode}} \cdot \mathit{name} \cdot \overline{\mathit{zipCode}})) \\
&\odot ((\overline{\mathit{zipCode}} \,||\, \overline{\mathit{zipCode}}) \cdot \overline{\mathit{distance}}) \\
&\equiv (\mathit{name} \,||\, \mathit{name}) \cdot \overline{\mathit{distance}}
\end{aligned}
$$

*The above calculations confirm that the result found by Algorithms 6.3, 6.4 and 6.5 is valid and the returned composition plan provides the requested behavior.* □

### 6.3.1 Complexity

In this section, we study the worst-case complexity of finding a single valid composition plan for a request $G : I_G \rightarrow O_G$ against a repository $S$ of components $C : I_C \rightarrow O_C$. We start this discussion with Algorithm 6.3 to find the complexity of finding an appropriate set $K$ of instances corresponding to $O_G$ (line 14 of the algorithm).

We assume that there are $N$ components in the repository $S$, and repository components and the goal have $|V|$ input/output data types in the worst case, where $|V|$ is the number of all available data types. The following is a discussion concerning the worst-case running time of different parts of the algorithm.

- line 2: $O(1)$
- lines 3-12: $O(|V|)$

- line 13: $O(1)$
- line 14: $O(1)$, the outputs to be generated can be kept in a set, where each time an instance corresponding to $I_G$ is created, it is removed from that set.
- line 15: $O(N|V|)$, the algorithm can keep track of the created instances at each step, then check repository components one by one to see which one can be used (i.e., for which one all the inputs instances are already created).
- line 16: $O(1)$
- line 17: $O(|V|)$
- lines 18-28: $O(|V|)$
- lines 29-33: $O(1)$

Because of the "go to" statement at line 30, the algorithm runs lines 14-29 in a loop which continues until a valid set $K$ is found (line 14) or no more component from the repository can be used (line 15). Considering the multi-level graph that is the result of the algorithm, at each execution of the loop, some nodes and edges are added to the graph. As mentioned before, the graph would find a solution up to level $N$ if there is any. So, in the worst case, the graph has to be expanded to level $N$. This could be the case, when there is no solution for the given request.

In an unsuccessful attempt where the algorithm returns a 'NO', each component can be used at any step while not creating the required output instances. Therefore, if we start by $|V|$ instances at level 0, in the worst case, we would apply $N$ repository components

- for the total of $N|V|$ times and create $N|V|^2$ instances at level 1,
- for the total of $N^2|V|^2$ times and create $N^2|V|^3$ instances at level 2,
- $\ldots$,
- for the total of $N^N|V|^N$ times and create $N^N|V|^{N+1}$ instances at level $N$.

Consequently, the loop would be executed $\sum_{k=1}^{N}(N|V|)^k = O((N|V|)^N)$ times in the worst case.

We see that in the worst case we would have an exponential time complexity. However, for two reasons we believe that the complexity of the proposed algorithm would be much less in practice.

- The worst case happens only if at each level all instances are used by every possible repository component and those components create the most possible output instances ($O(|V|)$) as a result. This is, to our belief, far from practice. In a rich enough repository, we expect only few components use inputs like *weatherInfo*, *personName*, *phoneNo*, .... Also, most components in such a repository would return only a few instances as their output.

- We could use some heuristics in order to prune the multi-level graph when we believe that adding some new nodes and edges would not likely lead to a solution. For example, assume that there are two components $C_k : a \rightarrow b^3$ and $C_l : b \rightarrow a^2$ in the repository, and there is one instance of $a$ in $I_G$. If we further assume that there is no other repository component that returns outputs of type $a$ or $b$, by applying Algorithm 6.3, aside from instances of other data types, 9331 instances of type $a$ and 4665 instances of type $b$ would be created up to level 10 of the multi-level graph. This would happen if there is no solution for the given request, or the solution is expected to be found somewhere after level 10 of the graph. As heuristics that can be embedded into Algorithm 6.3, some suggestions would be

  - When there are a large enough number of instances of a specific type and none of them has been used by any repository component, we skip using components that produce instances of only this specific type. This could prune a large portion of the multi-level graph and improve the performance of the algorithm.

  - Now consider the following definition:

    **Definition 6.1** *A* path *of length one exists from a data type $t_1$ to a data type $t_2$ if and only if there is a repository component $C$ where $t_1 \in I_C$ and $t_2 \in O_C$. In general, a* path *exists from a data type $t_1$ to a data type $t_2$ if either a path of length one exists from $t_1$ to $t_2$, or there is a sequence of data types $u_1$, $u_2$, ..., $u_{k-1}$ where there is a path of length one from $t_1$ to $u_1$, from $u_1$ to $u_2$, ..., from $u_{k-1}$ to $t_2$.* □

    It is trivial that the existence of a path from each input date type to each output data type is a necessary, but not sufficient, condition for the existence of a composition plan. So, we can look for such paths at the beginning of the algorithm, and if we noticed that there is no path from one of the inputs to one of the outputs we would return a 'NO' right away without proceeding to the rest of the algorithm. This would be

quite helpful when there is no solution for the given request, as it saves a large amount of computation.

In terms of the space required for the execution of this algorithm, based on the above discussion, in the worst case, $\sum_{k=0}^{N} |V|(N|V|)^k = O(|V|(N|V|)^N)$ space is required. This is dominant compared to the space required for finding the composition plan and, therefore, could be considered as the overall worst-case space complexity for the reasoning-based approach.

Now, let us move on to Algorithms 6.4 and 6.5. These two algorithms make recursive calls to each other. The worst case happens when the set $K$ is found at level $N$, and for each involved instance $m$, $|\mathit{from}(m)| = O(|V|)$ all the way up to level 0. This way, if the complexity of Algorithm 6.4 for a set $K$ found in level $N$ is $f(N)$, and the complexity of Algorithm 6.5 for each instance at level $N$ is $g(N)$, we would have $f(N) = O(|V|)g(N)$ and $g(N) = f(N-1)$. Considering that $g(0) = O(1)$, the result would be $f(N) = O(|V|^{N+1})$, which means their running time is exponential in the worst case. Again, this worst case is quite unlikely to happen given the constraints above, and a faster running time is expected in practice.

Therefore, the overall worst-case running time complexity of the reasoning-based approach would be $O((N|V|)^N + |V|^{N+1})$, which is equal to $O(|V|^N(N^N + |V|))$.

In the next chapter, we evaluate these algorithms by running them against some realistic test data, and study their running time with respect to the involved parameters.

### 6.3.2 Optimizing the Composition Plan

In Algorithm 6.4 we assumed that outputs in $O_G$ could be produced independently of each other. This could lead to a composition plan which is not optimized.

**Example 6.4** *Consider the component repository $S$ containing the following components*

- $\mathtt{Books} : \mathtt{iSBN} \rightarrow \mathtt{author} \wedge \mathtt{publisher}$
- $\mathtt{Authors} : \mathtt{author} \rightarrow \mathtt{nationality}$
- $\mathtt{Publishers} : \mathtt{publisher} \rightarrow \mathtt{websiteURL}$

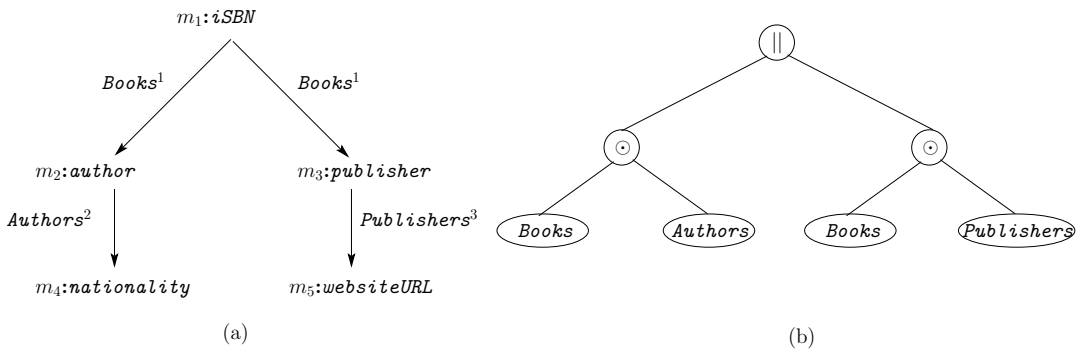



Figure 6.4: How a non-optimized composition plan might be returned by Algorithms 6.4 and 6.5. (a) The resulting multi-level graph for Example 6.4. (b) The resulting composition plan.

*and the request* $G$ : $\mathit{iSBN} \rightarrow \mathit{nationality} \wedge \mathit{websiteURL}$. *Using Algorithm 6.3, the multi-level graph of Figure 6.4-(a) would be created. The algorithm returns a 'YES' with* $K = \{m_4, m_5\}$. *We then use Algorithms 6.4 and 6.5 to find the corresponding composition plan. Following these algorithms, the composition plan shown in Figure 6.4-(b) is returned. This plan in composition algebra would be* $(\mathit{Books} \odot \mathit{Authors}) \,||\, (\mathit{Books} \odot \mathit{Publishers})$. *In this composition plan, since the output* $\mathit{publisher}$ *of* $\mathit{Books}$ *in the left subtree, and the output* $\mathit{author}$ *of* $\mathit{Books}$ *in the right subtree are ignored, the more detailed representation would be* $((\mathit{Books} \backslash \{\overline{\mathit{publisher}}\}) \odot \mathit{Authors}) \,||\, ((\mathit{Books} \backslash \{\overline{\mathit{author}}\}) \odot \mathit{Publishers})$. *Based on this plan, we need to use the component* $\mathit{Books}$ *twice and each of the components* $\mathit{Authors}$ *and* $\mathit{Publishers}$ *once in order to provide the requested behavior. But clearly we can solve the problem by using each of the components* $\mathit{Books}$, $\mathit{Authors}$ *and* $\mathit{Publishers}$ *only once; i.e.,* $\mathit{Books} \odot (\mathit{Authors} \,||\, \mathit{Publishers})$. *We see that in this alternative solution, we do not need to ignore any output from the* $\mathit{Books}$ *component. The difference between the two solution is, in fact, the difference between two equivalent composition algebraic expressions* $(\mathit{Books} \odot \mathit{Authors}) \,||\, (\mathit{Books} \odot \mathit{Publishers})$ *and* $\mathit{Books} \odot (\mathit{Authors} \,||\, \mathit{Publishers})$. □

Since Algorithms 6.4 and 6.5 return the extended non-optimized version of the composition plan, we can process the returned composition plan in another round to optimize it. The optimization is in fact converting the extended composition algebraic expression into its compact version.

At the moment, we consider only the optimization concerning the distributivity of the synchronization operator over the parallel operator. Other types of optimiza-

**Algorithm 6.6**: *Optimize*(*rootNode*) optimizes an expression tree, if possible.

```
   input  : the root of the expression tree, rootNode, to be optimized
   output : true, if the tree rooted at rootNode is optimized; no, otherwise
 1 begin
 2   if rootNode is a leaf then return false
 3   result=false
 4   if Optimize(rootNode.leftChild) or Optimize(rootNode.rightChild) then
 5     if rootNode used to represent a parallel expression then
 6       rootNode.expression = leftChild.expression || rightChild.expression
 7     if rootNode used to represent a synchronization then
 8       rootNode.expression = leftChild.expression ⊙ rightChild.expression
 9     result=true
10   end
11   if rootNode represents a parallel expression and
12     both rootNode.leftChild and rootNode.rightChild represent a synchronization and
13     rootNode has two grandchildren LGChild1 and RGChild1 of its left and right child
     where
14       LGChild1.expression=RGChild1.expression then
       //these conditions might lead to the specific optimization
15     LGChild2=Sibling(LGChild1)
16     RGChild2=Sibling(RGChild1)
17     remove rootNode.leftChild
18     rootNode.leftChild=LGChild1
19     LGChild1.parent=rootNode
20     rootNode.rightChild.leftChild=LGChild2
21     LGChild2.parent=rootNode.rightChild
22     rootNode.rightChild.rightChild=RGChild2
23     RGChild2.parent=rootNode.rightChild
24     rootNode.rightChild.expression = (LGChild2.expression) || (RGChild2.expression)
25     delete subtree RGChild1
26     rootNode.expression =
       (rootNode.leftChild.expression) ⊙ (rootNode.rightChild.expression)
27     result=true
28   end
29   return result
30 end
```

tions might be performed in a similar manner. Algorithm 6.6 represents how this optimization is actually done. This algorithm converts an expression tree like the one in Figure 6.6-(a) to an optimized tree like the one in Figure 6.6-(b).

The way the algorithm works is quite simple. It basically makes recursive calls on the children of *rootNode* to optimize them as well, and then changes the positions of four subtrees (grandchildren of *rootNode*) in case an optimization at the current

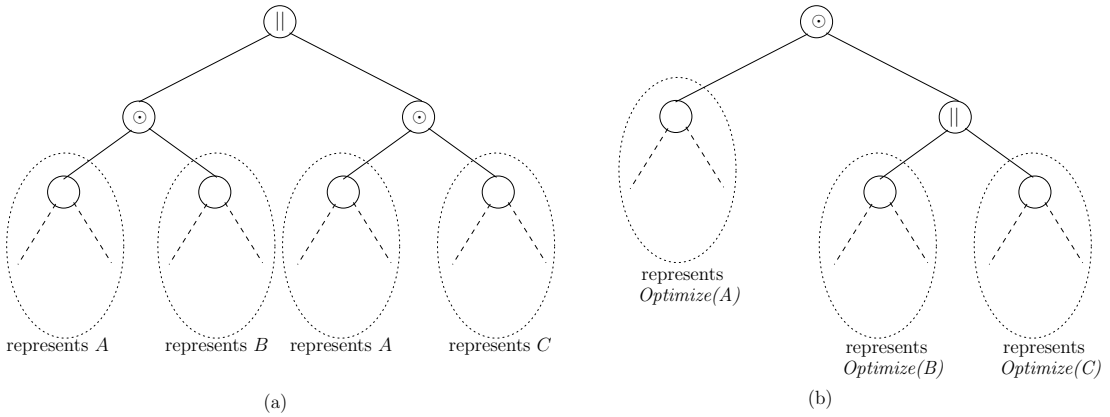



Figure 6.5: The *Optimize* algorithm converts the tree in part (a) to the tree in part (b).

level is possible. It starts by checking if *rootNode* is a leaf node. If so it returns false meaning that no optimization is possible (line 2). It then uses a temporary variable *result* which indicates if any optimization has been performed so that the algorithm returns an appropriate output (line 3). In lines 4-10 it recursively optimizes the left and right subtrees of *rootNode*, updates its expression accordingly, and sets *result* to true. In lines 11-28 it examines *rootNode* and its children and grandchildren to see if any optimization is possible at this level. For this optimization to happen,

- *rootNode* must represent a parallel execution of its left and right subtrees,
- both left and right children of *rootNode* must represent a synchronization of their left and right children (they must not be leaf nodes), and
- two grandchildren of *rootNode*, e.g., *LGChild1* and *RGChild1*, that are not siblings must represent the same expression.

These conditions, if held, represent an expression in which the synchronization operator has been distributed over the parallel operator. This expression is then converted to the compact version. The algorithm first finds its two other grandchildren *LGChild2* and *RGChild2* (lines 15-16). It then substitutes its left child with one of the common grandchildren (lines 17-19). Then, it rebuilds its right subtree as the parallel execution of its other two grandchildren (lines 20-24). It finally removes one of the common grand children (line 25), updates the expression represented by *rootNode* (line 26), and sets *result* to true indicating that the tree has been optimized (line 27). Figure 6.5 illustrates this conversion.

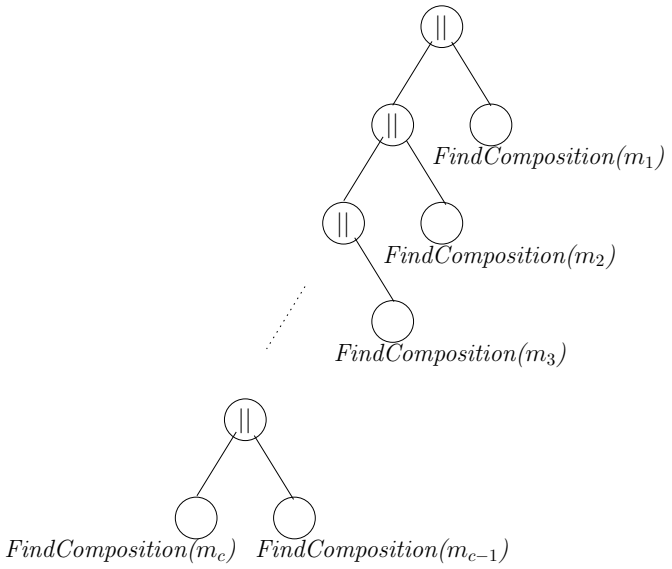



Figure 6.6: How the expression tree would look for set $K = \{m_1, m_2, \cdots, m_c\}$, where $c = O(|V|)$.

We use the following lemma to explain the complexity of the optimization process represented in Algorithm 6.6.

**Lemma 6.2** *The height of the expression tree created by Algorithms 6.4 and 6.5 is* $O(N|V|)$ *in the worst case.*

**Proof.** The worst case happens when the set $K$ is found at level $N$ of the multi-level graph, where each instance has a *from* set of size $c = O(|V|)$ all the way up to level 0. By running Algorithms 6.4 and 6.5, the structure of the tree at the beginning would look like the one in Figure 6.6. At this point the height of the tree would be $c-1$, and since there are $N-1$ more graph levels with the similar tree structure, the height of the tree in the worst case would be $(c-1)+(N-1)(c-1) = N(c-1) = O(N|V|)$. □

It is quite trivial that with a little change in Algorithm 6.4, the height $O(N \log |V|)$ could be achieved.

If we consider $f(h)$ as the order of complexity of Algorithm 6.6 for an expression tree of height $h$, we can easily see that $f(h) = 2f(h-1)$. Since $f(1) = O(1)$, we conclude that $f(h) = O(2^h)$. In the worst case, we would have $O(2^{N|V|})$ time complexity, which again indicates an exponential running time.

## 6.4 Discussion

In this chapter we proposed a forward chaining procedure for finding compositions from a component repository that satisfy a given request. We mentioned how this procedure analyzes temporary instances created by repository components according to a multi-level graph until it reaches a point at which all the necessary instances are created (successful search), or it realizes that a solution cannot ever be found (unsuccessful search). Since this search is performed in a breadth-first manner, the first solution is guaranteed to be the shortest-path solution, meaning that fewest number of components would be required to build that solution.

In Examples 6.1 and 6.3 we explained how the solution returned by the corresponding algorithms can be validated using the composition algebraic rules by checking if they provide the exact same behavior as the one given in the request. We showed in Example 6.1 that it is possible that these algorithms find solutions with a behavior that is close to, but not exactly the same as, the request. Although in most cases this closeness might be sufficient for the user, in case a 100% match is sought the solutions should be verified before being reported. This would suggest the presence of an automatic behavior verifier according to the rules of composition algebra as explained in Section 4.2. This behavior verifier would receive the behaviors of repository components and the specification of a composite component in terms of its constituent components and calculates the external behavior of the composite component in terms of its inputs, outputs, and their temporal order. Development of such an automatic behavior verifier will be left as a future work.

# Chapter 7

# Evaluation

In this chapter we provide a simple comparison between the composition approaches of Chapters 5 and 6. We also explain how we can achieve a better performance by taking advantage of positive features of both of them. Then we provide some experimental results we obtained by implementing the reasoning-based approach and running it on some sample repositories. In these experiments, we try to understand the run time performance of the composition approach against repositories with different numbers of components and data types. Then we discuss the applicability of two available reasoning tools in solving the composition problem. Finally, we explain how to adapt the proposed approaches to web services composition.

## 7.1 Graph-Based vs. Reasoning-Based Approach

Although the two approaches presented in Chapters 5 and 6 are inherently different, we can compare them by considering their corresponding worst-case running time and space complexities. Table 7.1 represents this information.

| | Running Time Complexity | Space Complexity |
|---|---|---|
| Graph-Based | $O(N^{\|V\|}\|V\|(N(\|V\|-2)!)^{2\|V\|})$ | $O((\|V\|-1)! \times N^{\|V\|-1})$ |
| Reasoning-Based | $O(\|V\|^N(N^N+\|V\|))$ | $O(\|V\|(N\|V\|)^N)$ |

Table 7.1: Worst-case running time and space complexities of the graph-based and reasoning-based approaches, where $N$ and $|V|$ are the number of repository components and data types, respectively.

We can see that both running time and space complexities in the graph-based approach are exponential in terms of the number of involved data types, while in the reasoning-based approach they are exponential in terms of the number of repository components. Since these two factors in the reasoning-based and graph-based approaches are highly dependent on the height of the multi-level graph and the size of the dependency graph, respectively, we would expect that their average complexity measures have the same exponents as well. Therefore, a reasonable suggestion would be to use the graph-based approach when the component composition is being performed on a repository of components which are of similar semantic domains. In this case, the number of data types would normally be small compared to the number of repository components. On the other hand, in the general case, it would be beneficial to use the reasoning-based approach for component composition, because in that case we would expect the number of involved data types to be larger compared to the number of repository components.

Moreover, in Section 6.3.1 we proposed some heuristics for improving the performance of the reasoning-based composition approach. One of these heuristics is based on the notion of reachability of data types (Definition 6.1). It states that for a given request, all the output data types must be reachable from the input data types through some repository components. We can easily see traces of the graph-based approach in this heuristic, as the graph-based solution is based on the concept of reachability in the dependency graph. Because of this fact, we can take advantage of other techniques used in the graph-based approach to improve the performance of the reasoning-based approach. For example, other than checking the reachability of the outputs in the given request from its inputs, we can use Algorithm 5.1 to make sure that the paths found would be valid candidates before starting the reasoning-based approach and getting involved with the complexity of creating data type instances. These two algorithms are capable of rejecting the given query much sooner compared to the reasoning-based procedure. Note that their complexity would be $O(|V|^2)$ and negligible compared to the average complexity of the reasoning-based approach. Therefore, a better solution would be taking advantage of the dependency graph properties in the reasoning-based approach.

## 7.2 Implementation

In Chapters 5 and 6 we studied the worst-case running time complexity of the graph-based and the reasoning-based composition approaches. We explained in

those chapters why those calculated worst-case complexities are believed to be quite far from the real world. In order to study the performance of the composition approaches in more realistic situations, we implemented the reasoning-based approach which, according to the results of the previous section, is more general compared to the graph-based approach.

The basic reasoning-based approach was implemented for these experiments and only one heuristic was considered to improve the run time performance. Specifically, a large enough limit was set for the total number of instances created from each data type to avoid the situations such as exponential increase in the number of instances, like the example given in Section 6.3.1.

The implementation was performed on Microsoft® Windows XP Professional Edition platform (Service Pack 2) using Microsoft® Visual C++, which is part of the Microsoft® Visual Studio .NET 2003 package. MySQL Server 4.1, by MySQL® AB, was used as the database engine to play the role of the component repository. The code written for performing the evaluations is around 15000 lines. A desktop computer with an Intel® Pentium 4 (3200 MHz) CPU and 1 gigabytes of internal memory was used for running the experiments.

In this section, we present some of the experiments we ran along with their run time results. In each case we provide a justification as well. In these experiments we try to figure out the complexity of the implemented approach with respect to each of the parameters involved. We provide experimental results for the simple case where each repository component has a single input and a single output, and the generic case.

By taking a random sample of the existing web services from the web site *http://www.webservicelist.com/* we tried to come up with the distribution of the four involved parameters relative to the component repository. These four parameters are the number of input data types and the number of output data types in each component, and the cardinality of each data type when it appears as an input, and when it appears as an output. The result for the total of 104 web service operations, according to the Chi-Square Goodness of Fit test[1], is the following:

[1]In the goodness of fit test, the null hypothesis $H_0$ is tested, and if the test turns out to be successful, it would mean that the null hypothesis cannot be rejected on the given data. In the Chi-Square version of this test, there are two parameters involved: one is the degrees of freedom ($df$) which is equal to the number of possible and independent outcomes, and the other is the probability of rejecting the null hypothesis when the null hypothesis is true ($\alpha$). For example, $\chi^2_{5,0.05}$ indicates a Chi-Square test with 5 degrees of freedom and $\alpha = 0.05$ [11].

- Number of input data types per operation ($N_I$): Geometric distribution, mean = 3.04 ($\chi^2_{4,0.05}$).
- Number of output data types per operation ($N_O$): Geometric distribution, mean = 1.34 ($\chi^2_{2,0.05}$).
- Cardinality of a data type as an input ($C_I$): Geometric distribution, mean = 1.06 ($\chi^2_{4,0.05}$).
- Cardinality of a data type as an output ($C_O$): Geometric distribution, mean = 1.04 ($\chi^2_{2,0.05}$).

The related charts are shown in Figures 7.1. Using this information we created several component repositories and ran large numbers of tests against them in order to obtain good average run times. For each repository we built we knew how many components and data types would be involved. For each component in such a repository in the generic tests (Experiments 4-6), we used the above random distributions to, first, figure out how many inputs and outputs it would have, and, second, what the cardinality of each of those inputs and outputs would be.

### 7.2.1 Experiments

In the rest of this section we present the results of various experiments with the reasoning-based approach. In all these experiments the goal has been finding the first composition that satisfies the request. Other than the above four parameters, the number of components in the repository ($N_C$) and the number of distinct data types in the repository ($N_T$) are also inputs. In all these experiments we assumed that all data types have equal chances of appearing in inputs/outputs of repository components. The requests generated in all the experiments have the same properties, e.g., the distribution of number of input/output data types, as the corresponding repository.

#### Experiment 1: Simple Repository (I)

The first experiment was run on a set of simple repositories containing components with a single input and a single output. The number of components in the repositories was fixed at 1000, while the number of data types changed from 1000 to 3990. Figure 7.2 contains the results of this experiment as a line chart. In this chart, the light line represents the average run time when the composition planner

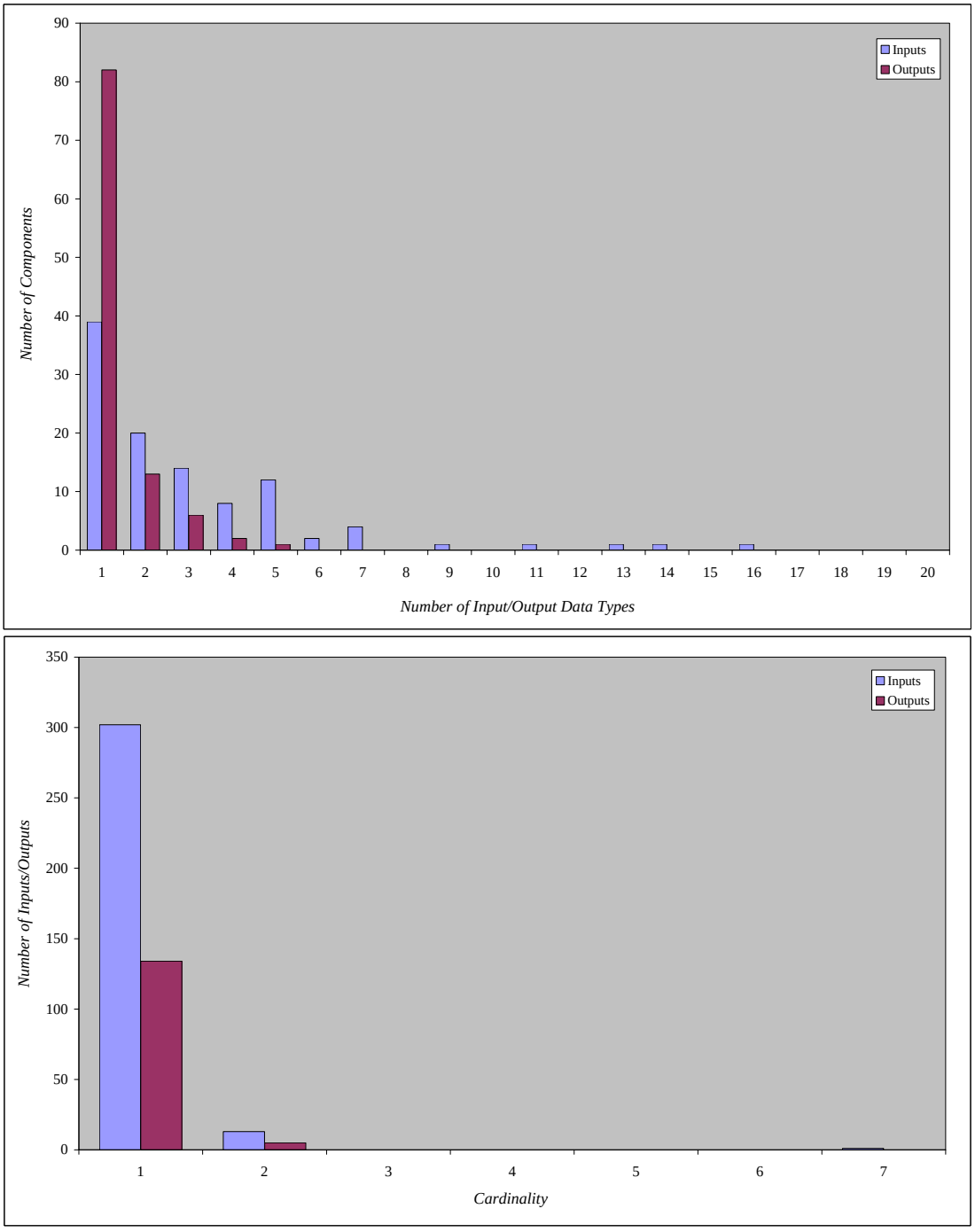



Figure 7.1: Number of web service operations with respect to the number of input/output data types (top) and number of input/output data types against their cardinality (bottom) in the sample of 104 operations.

did not return successfully, i.e., when a composition plan could not be found. The dark line, on the other hand, represents the average time for successful searches,

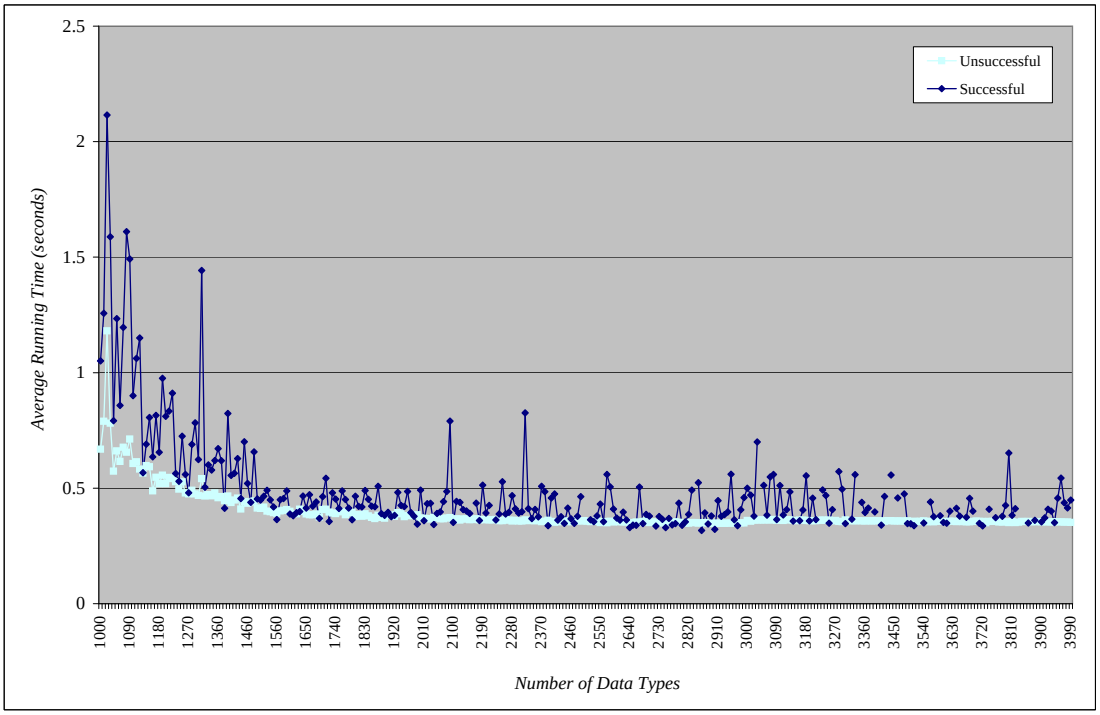



Figure 7.2: Average run times for Experiment 1: $N_C = 1000$, $N_T = 1000 \cdot \cdot 3990$ (steps of 10), $N_I = 1$, $C_I = 1$, $N_O = 1$, $C_O = 1$. For each specific number of data types, as many as 10000 random requests were generated and solved using the composition approach.

i.e., when a composition plan was returned.

Regarding the unsuccessful tries, we notice that the run time is slightly decreasing as the number of data types increases. To justify this, we realize that data types play the role of connectors among different repository components. When a data type $t$ is an output in one component and an input in another, we say that $t$ *connects* these two components. If the connectivity among repository components is low there would be fewer instances in each level of the multi-level graph of Section 6.3. When we increase the number of data types this connectivity decreases and, therefore, if a request is destined to be unsuccessful, fewer instances would be created in the multi-level graph until reaching a dead-end. A *dead-end* is a point at which no more repository component can be applied. All this means less run time.

We see more fluctuations in the average run times for successful tries. The reason is simply that the tests were performed for small number of successful results (at most 20) because the chance of finding a solution diminishes as the number of data types grows. In fact, this is why the diagram is not continuous, as we removed the

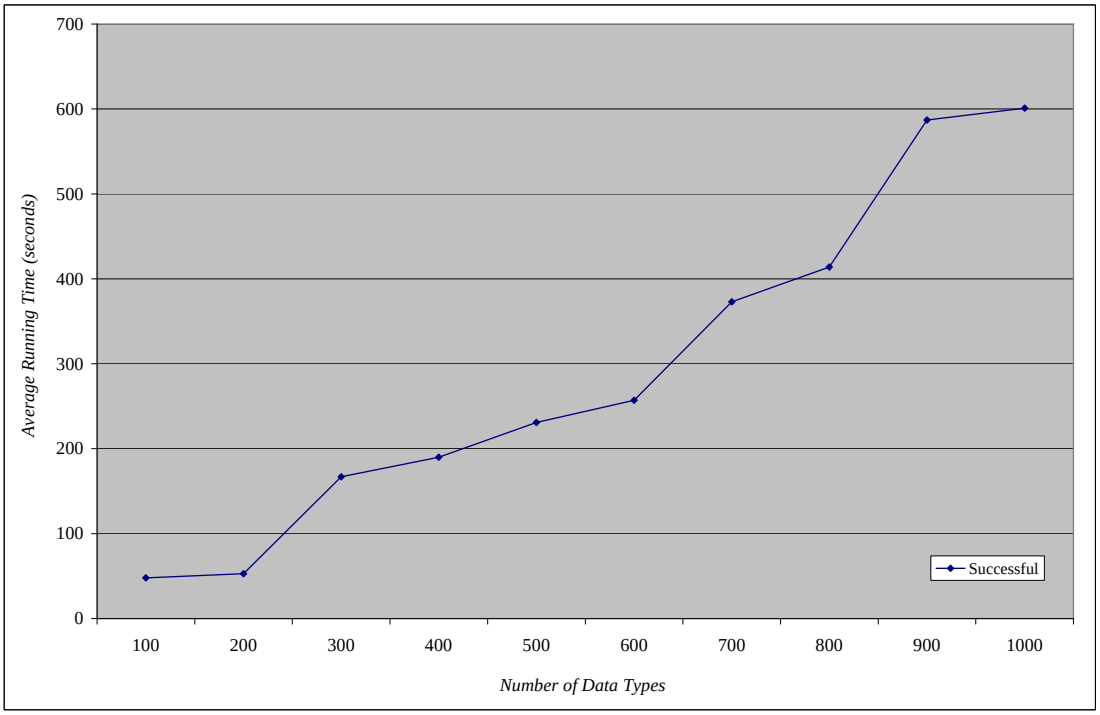



Figure 7.3: Average run times for Experiment 2: $N_C = 10000$, $N_T = 100 \cdot \cdot 1000$ (steps of 100), $N_I = 1$, $C_I = 1$, $N_O = 1$, $C_O = 1$. For each specific number of data types, 100 random successful requests were considered.

points where no result could be found (even after trying 10000 random queries). For the same reason, the run times seem to be decreasing as we go towards more data types.

We conclude that as the number of data types with respect to the number of components grows, the chance of finding a satisfying composition and the run time of the reasoning-based composition approach both decrease.

### Experiment 2: Simple Repository (II)

In this example, we again consider the repository of components with a single input and a single output. This time the number of repository components was 10000, while the number of data types changed from 100 to 1000. Figure 7.3 shows the results of this experiment. Since the number of data types with respect to the number of components is small, the chance of finding a composition for a given request is much higher compared to the previous experiment. That is why in this experiment we study only the run time for successful searches. In fact, none

of the 100 random tests for each number of data types returned unsuccessfully, which means (at least) in the case of the simple repository of single input/output components, if the number of components are much higher than the number of data types, normally we would be able to find a composition for the given request. The chart shows that the average run time can be estimated to be linearly dependent on the number of repository data types.

**Experiment 3: Simple Repository (III)**

Again, we considered the simple repository as the previous two experiments. However, this time we tried to keep the number of data types fixed at 100 and change the number of repository components from 100 to 2000. Similar to the previous experiment, we noticed that the average run time is smoothly increasing as we increase the number of components. We believe that the fact that this increase in the average run time does not occur at a higher rate is significantly related to the notion of *average graph expansion*.

**Definition 7.1** *Algorithm 6.3 processes a given request by creating a multi-level graph. It keeps expanding this graph until either it finds a solution or it reaches a point that from that point forward no solution can ever be found. In each case the last level number of the graph is called the* graph expansion *for the given request against the given repository. For example, the corresponding graph expansion in Figure 6.2 would be 3. By submitting multiple requests against the same repository, we can obtain an average value for this graph expansion, which is called the* average graph expansion *for that repository.* Average successful graph expansion *and* average unsuccessful graph expansion *are the similar terms used only for successful and unsuccessful searches, respectively.* □

In this experiment, we notice that the average successful graph expansion decreases as we increase the number of components. This means that every time we increase the number of components, although more instances would be created at each graph level, the graph is expanded less before finding a solution. In other words, decreasing graph expansion would be the reason we do not see a higher slope in Figure 7.4. Figure 7.5 shows the maximum and average successful graph expansions for the repository of this experiment. The maximum successful graph expansion is the maximum of all graph expansions for successful requests. We see in this diagram that, for example, when the number of components passes ten times the

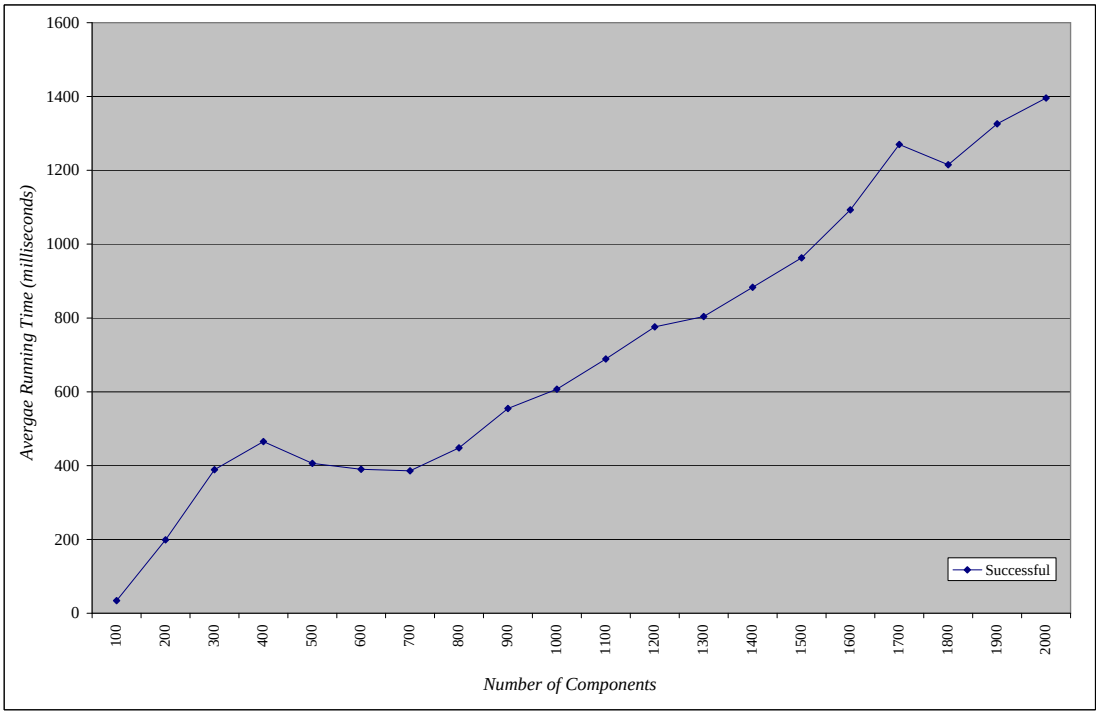



Figure 7.4: Average run times for Experiment 3: $N_C = 100 \cdot \cdot 2000$ (steps of 100), $N_T = 100$, $N_I = 1$, $C_I = 1$, $N_O = 1$, $C_O = 1$. For each specific number of data types, 100 random requests leading to a composition were considered.

number of data types, the average successful graph expansion is around 2 and the successful graph expansion would never go beyond 3.

The *success rates*, i.e., the chance of finding a composition for a given request, for this experiment are shown in Table 7.2. The result indicates that in a simple

| # Components | 100 | 200 | 300 | 400 | 500 | 600 - 2000 |
|---|---|---|---|---|---|---|
| Success Rate | 0.057 | 0.676 | 0.939 | 0.965 | 0.979 | 1.000 |

Table 7.2: Success rates for Experiment 3.

repository of single input/output components, if the number of components is three times or more than the number of data types there is a high chance of finding a solution for all given requests. Also, if the number of components is six times or more than the number of data types, we should expect to find a solution for all requests.

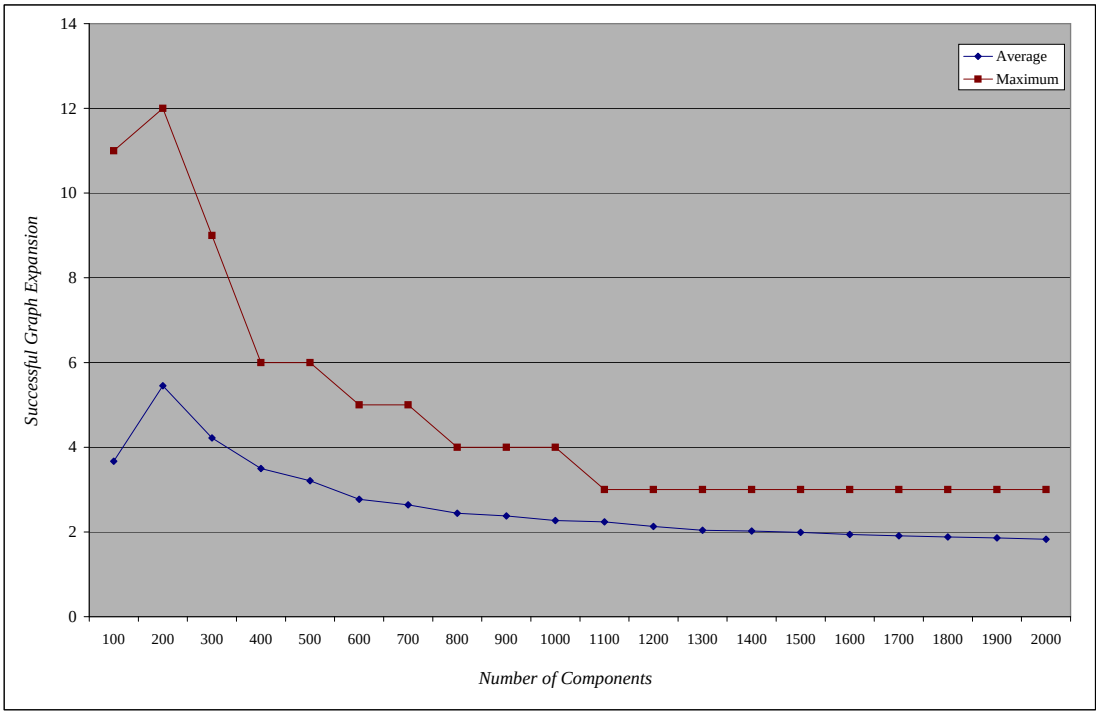



Figure 7.5: Maximum and average successful graph expansions for Experiment 3.

### Experiment 4: Generic Repository (I)

After studying the simple case repository we now move to experiments involving a more general repository. We use the repository parameters based on the results of the random sampling discussed earlier in this chapter. In this experiment, we picked parameter values, i.e., number and cardinality of inputs and outputs in each component, the same as those results. The geometric distributions with means 3.04, 1.06, 1.34 and 1.04 were picked for the number of input data types, cardinality of each input data type, number of output data types, and cardinality of each output data type, respectively. We fixed the number of data types at 1000 and varied the number of repository components from 100 to 1000. Figure 7.6 illustrates the average run time for unsuccessful searches. In this experiment the number of components compared to the number of data types is small and, therefore, the success rates have been very low. As a result, there were few successful searches, and we did not include their average run time in this chart. We can easily see that the average run time increases somewhat linearly when we increase the number of components.

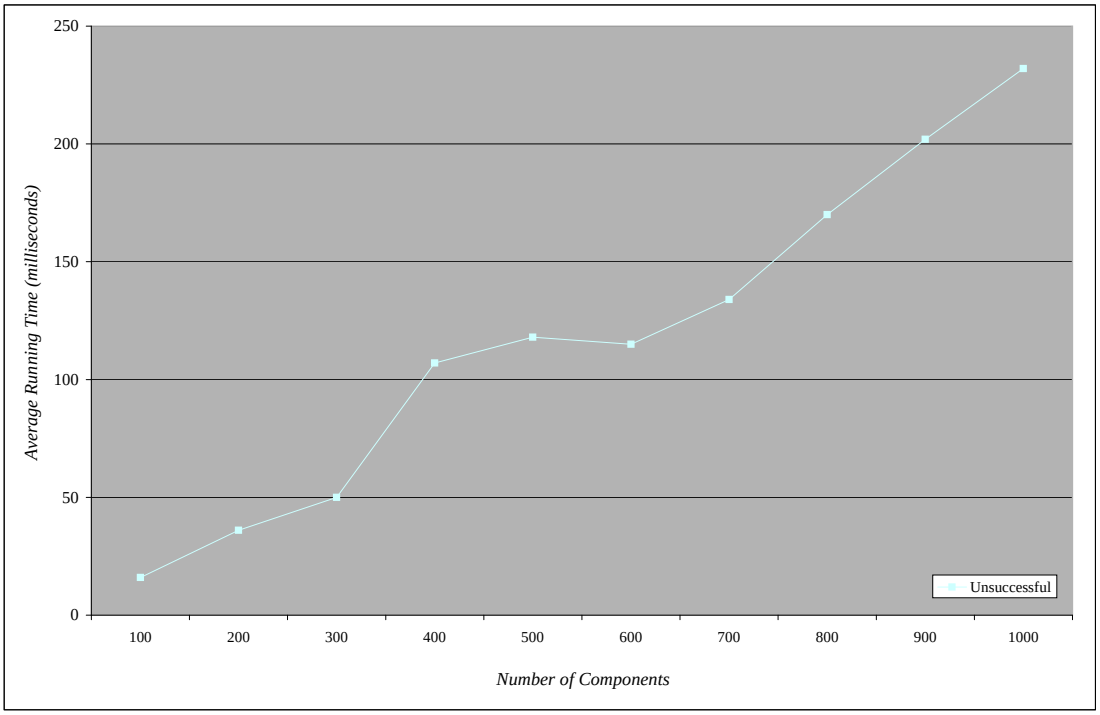



Figure 7.6: Average run times for Experiment 4: $N_C = 100 \cdot \cdot 1000$ (steps of 100), $N_T = 1000$, $N_I = Geometric(3.04)$, $C_I = Geometric(1.06)$, $N_O = Geometric(1.34)$, $C_O = Geometric(1.04)$. For each specific number of data types, 100 random requests leading to a composition were considered.

### Experiment 5: Generic Repository (II)

In this experiment we used the same distribution parameters as in Experiment 4, except that this time we fixed the number of components at 1000 and changed the number of data types from 100 to 1000. The average run times for successful and unsuccessful searches are shown in Figure 7.7. Note that the chart in this figure is logarithmic on the values of the average run time. Although the diagrams for successful and unsuccessful searches in this chart are rather unusual, they can be simply explained. The justification, which is somewhat similar to the one presented in Experiment 1, is based on Table 7.3, which includes the success rates for different numbers of data types, and also Figure 7.8. Table 7.3 shows that as we increase the number of data types, the chance of finding a composition significantly decreases. Also, by increasing this number the probability that two repository components are connected goes lower as well, which means repository components become less connected. Now, when we start with 1000 components and 100 data types, since the number of components is much bigger than the number of data types, we expect

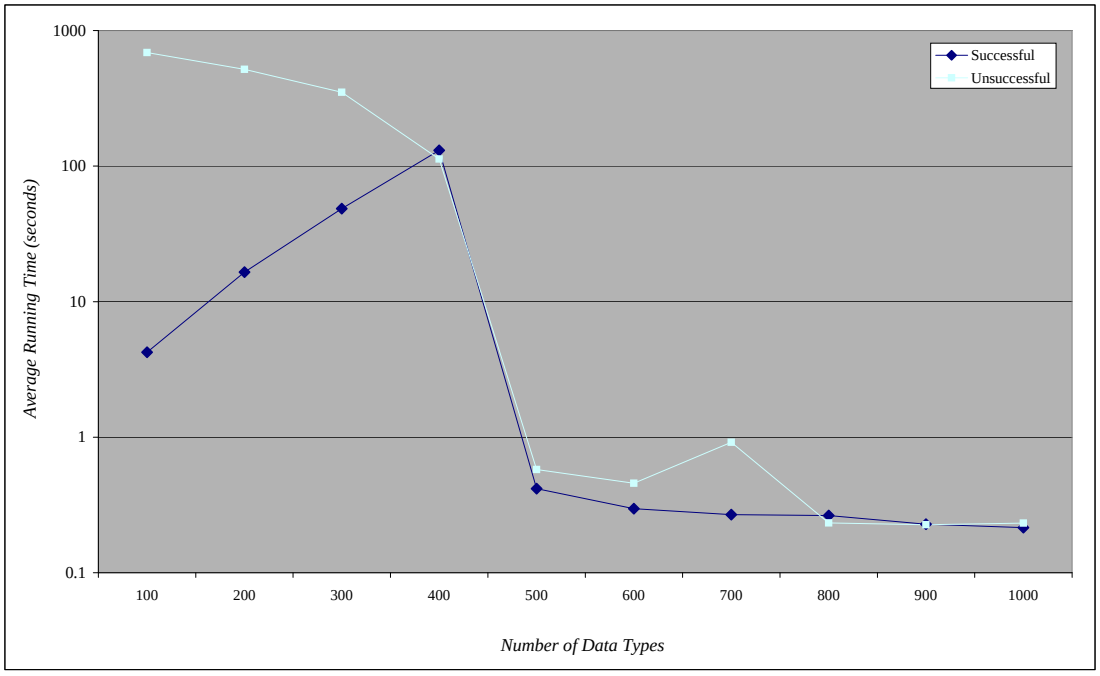



Figure 7.7: Average run times for Experiment 5: $N_C = 1000$, $N_T = 100 \cdot\cdot 1000$ (steps of 100), $N_I = Geometric(3.04)$, $C_I = Geometric(1.06)$, $N_O = Geometric(1.34)$, $C_O = Geometric(1.04)$. For each specific number of data types, 100 random requests leading to a composition were considered.

| # Data Types | 100 | 200 | 300 | 400 | 500 | 600 - 700 | 800 - 1000 |
|---|---|---|---|---|---|---|---|
| Success Rate | 0.709 | 0.429 | 0.262 | 0.047 | 0.002 | 0.001 | 0.000 |

Table 7.3: Success rates for Experiment 5.

the repository to be well connected. Note that, we assume repository components form the nodes and shared data types between components create the edges and connections between components. As we start increasing the number of data types, we break the connections between some components, but up to some point, we expect the whole repository to be still one connected set. This point, according to Figure 7.7, is when there are 400 data types in the repository. That is why we see a big difference in average run times between the experiments for 400 and 500 data types. The average graph expansions in Figure 7.8 confirm this theory by showing that the average graph expansion for both successful and unsuccessful searches reaches to its maximum when there are approximately 400 data types in the repository.

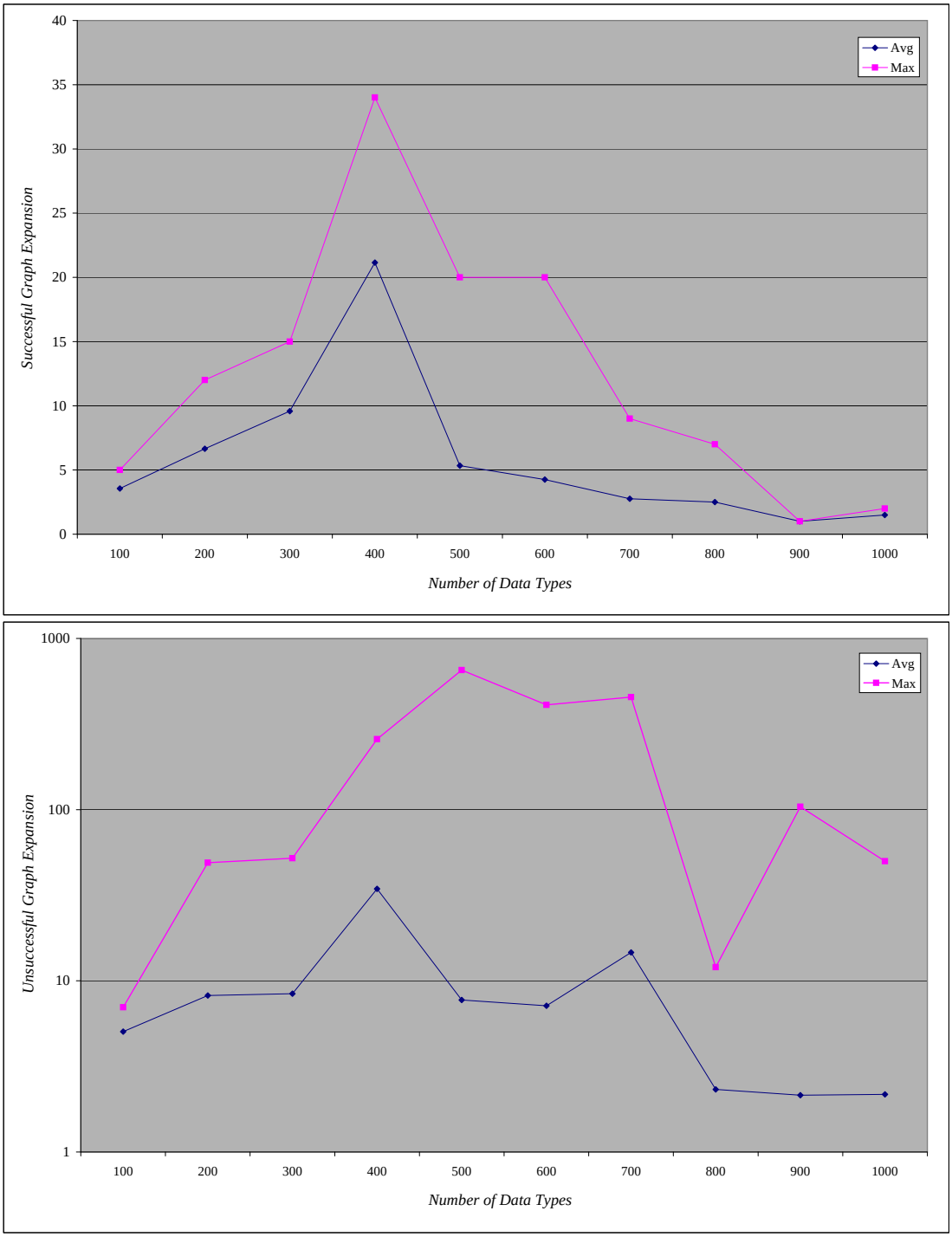



Figure 7.8: Maximum and average graph expansions for Experiment 5.

By comparing the charts in Figures 7.7 and 7.8 we notice some facts that seem to be unusual. We try to explain and analyze a few of them in this part.

- Although the average unsuccessful graph expansion for 100 data types is less than this average for 500 data types, the average run time for unsuccessful searches is significantly bigger for 100 data types. This is because when the repository is well connected, dead-ends rarely occur during the graph expansion. Therefore, we expect the performance improvement heuristics to help in these situations by, for example, limiting the number of instances created from each type. This takes much longer compared to the case in which graph expansion comes to a dead-end due to low connectivity of repository components. This exactly is the case for the two experiments for 100 and 500 data types. For 100 data types, almost all unsuccessful searches were terminated by those heuristics. However, for 500 data types, this happened for only a few of them. Note that for 100 data types, the graph levels are quite more crowded than those for 500 data types. It is obvious that the heuristics that are used in the approach directly affect the average run time for unsuccessful searches.

- The average successful graph expansion for 100 data types is less than this average for 500 data types, and yet the average run time for successful searches is considerably larger for 100 data types. The reason behind this is only the connectivity of repository components for the two cases. For 100 data types, the connectivity is stronger and therefore, many more instances are created in the first levels. However, for 500 data types, due to lower connectivity, we expect to see fewer instances in those first levels. Since fewer created instances means shorter run time, the experiment with 500 data types finds compositions at a faster rate.

Since the multi-level graph is an example of random graphs studied by Erdös and Rényi [40], their properties would also confirm the changes we see in this experiment.

**Experiment 6: Generic Repository (III)**

This experiment is similar to Experiment 4, except that the number of data types is fixed at 100 and the number of components changes from 100 to 2000. Because we expect higher success rates, we can study the performance for successful searches as well. Figure 7.9 contains the average run time for successful and unsuccessful searches. Each case was run until 25000 random requests or 100 successful searches were submitted. Note that the chart in this figure is also logarithmic on the values

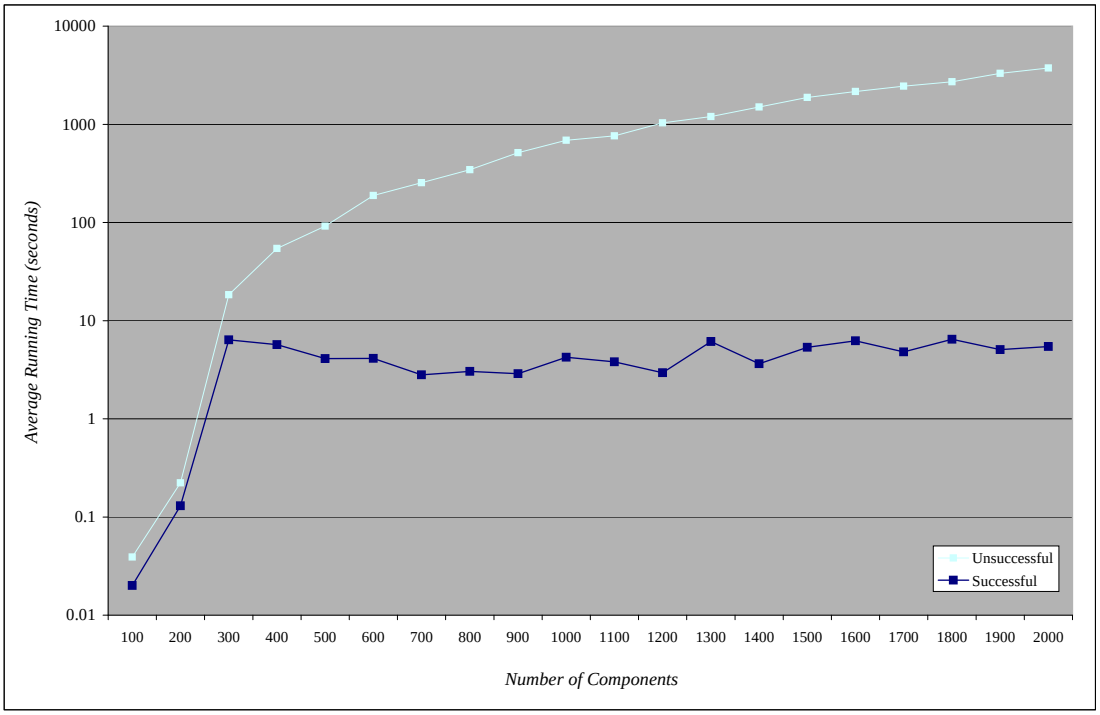



Figure 7.9: Average run times for Experiment 6: $N_C = 100 \cdot \cdot 2000$ (steps of 100), $N_T = 100$, $N_I = Geometric(3.04)$, $C_I = Geometric(1.06)$, $N_O = Geometric(1.34)$, $C_O = Geometric(1.04)$. For each specific number of data types, 100 random requests leading to a composition were considered.

of the average run time. The average and maximum graph expansions for successful and unsuccessful searches are shown in Figure 7.10. To study the result of this experiment in more detail the success rates and the average number of created instances in successful and unsuccessful searches is shown in Figure 7.11. By comparing the charts related to this experiment we observe that since the number of instances created for the cases with 100 and 200 components is considerably small, the average run times are quite low compared to the other cases. For these two cases the success rate is quite low as well. Although the number of components is not less than the number of data types in these two cases, it is not large enough to cause most repository components to become involved. If we consider the number of instances created in each level of the multi-level graph, we see that the number of instances at each level hardly passes 3 and 10 for 100 and 200 components, respectively. When there are few instances in a level not many repository components can be applied on those instances and, therefore, the chance of finding a composition would be low as there is a high possibility of reaching a dead-end in such a

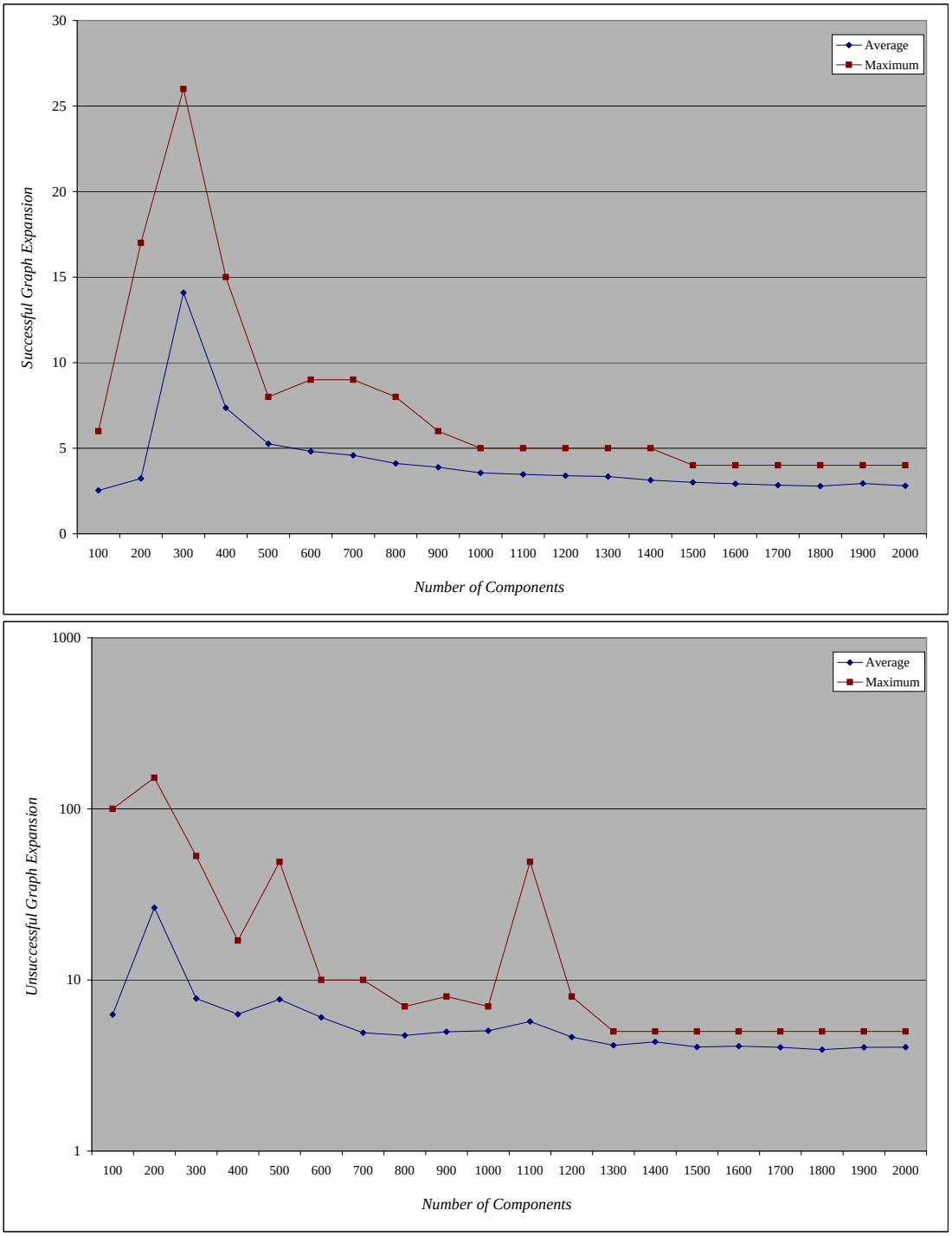



Figure 7.10: Maximum and average graph expansions for Experiment 6.

situation.

When the number of components increases, the difference between the average run time for successful and unsuccessful searches highly increases as well. This can

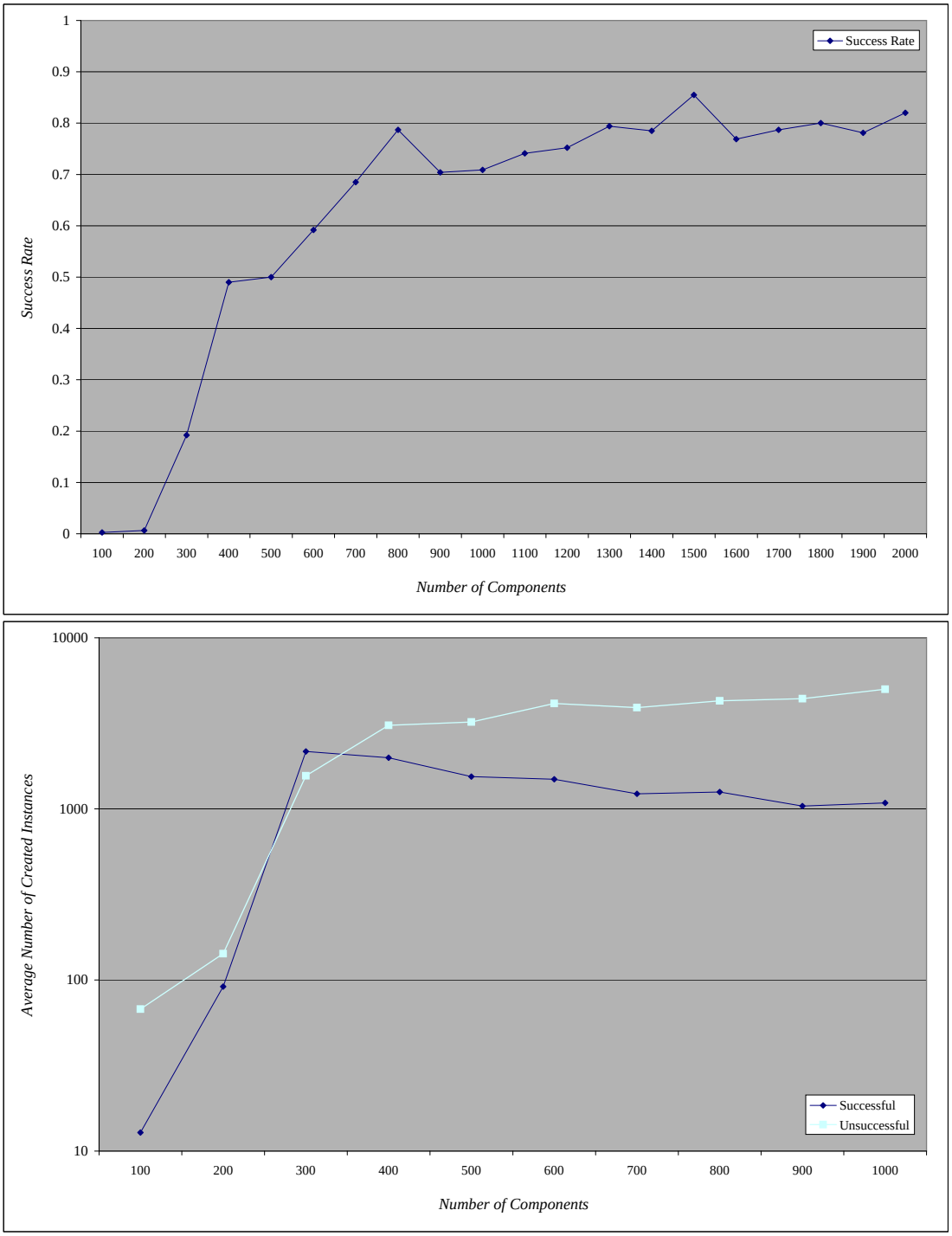



Figure 7.11: Success rates and average number of instances in Experiment 6.

be justified by the fact that increasing the number of components lowers the chance of reaching a dead-end and ending with an unsuccessful search, as more instances are created in each level of the multi-level graph. Therefore, the basic way to determine

that a search is unsuccessful would be the performance heuristics mentioned before. At the moment, according to the implemented performance heuristic, this means more created instances and, consequently, more processing time. The chart in Figure 7.9 highlights the need for even more performance heuristics to improve the run time performance for unsuccessful searches.

Regarding the successful searches, we observe that the run time is smoothly increasing when the number of components goes up. However, this increase is much less compared to the case for unsuccessful searches. This smooth increase can be justified using the corresponding average successful graph expansion in Figure 7.10. As mentioned before, when we increase the number of components, more instances would be created in each level which adds to the run time. On the other hand, since the average successful graph expansion goes lower, we would expect less run time. These two factors partly neutralize each other and cause the changes at a lower rate.

Comparing the charts for the average run time and the average number of created instances, we notice that they are quite similar. This simply means that, as we expect, there is a direct relation between the number of instances created and the run time of the approach.

### Summary of Results

According to the experiments we reported in this section, we observe that the case for the simple repository of Experiments 1-3 is rather different from the generic case. In the experiments involving the simple repository we obtained diagrams with much more linearity involved for the average run times with respect to the number of components or data types. However, in Experiments 4-6 we had to use logarithmic scales in some cases to represent the results.

We saw in this section how the run time performance of our reasoning-based approach can be justified by studying the underlying multi-level graph, the success rates, the average graph expansions, and the number of instances created during the search. Using these justification techniques we can also predict the behavior of the approach for repositories with different attributes.

According to the reported experiments we realize that although our reasoning-based approach performs well in most circumstances, there is still room for improvements. For instance, in Experiment 6 we noticed that in unsuccessful searches we face huge increases in the run time when we increase the number of components.

This suggests that we need to improve and apply even more performance heuristics so that an unsuccessful search can be guessed well in advance. This task is left as a future work in performance improvement of the approach. As mentioned in Section 7.1 applying some of the techniques from the graph-based approach is expected to substantially improve the composition approach from this point of view.

## 7.3 Discussion of Other Possible Approaches

In Chapter 6 we proposed a reasoning-based approach in automatic composition planning. Although this approach is motivated by a reasoning algorithm for Horn clauses, it does not explicitly take advantage of existing logical reasoning programming languages and tools. Prolog [29] is one such programming language that comes with different implementations. Prolog is, in fact, based on the principles of the Horn clause logic. Although its main application is in the Artificial Intelligence area, it is being used in other areas as well, such as compiler construction, computer algebra and database systems. For a complete review of the Prolog programming constructs the reader is referred to [60].

To answer queries, Prolog looks into its current knowledge base to find the solution. Its knowledge base contains a set of facts and rules processed in a sequential order.

**Example 7.1** *Consider the following set of facts as the knowledge base of a Prolog engine:*

```
animal(tiger).
animal(elephant).
animal(dove).
```

*If the query* `?- animal(X).` *is submitted to this engine, the followings would be the result:*

```
X = tiger ;
X = elephant ;
X = dove
```

*Since Prolog processes the above knowledge base from the top to the bottom, it is not possible to submit the above query and get the results in a different order.* □

Another property of Prolog is that it uses the backtracking strategy in searching the knowledge base to solve a given query. Backtracking is especially used when Prolog tries to use knowledge base rules to answer given queries.

**Example 7.2** *Consider that the following knowledge base, which is the extended version of the one in the previous example, is given to a Prolog engine:*

```
animal(tiger).
animal(elephant).
animal(dove).
flies(balloon).
flies(dove).
bird(X) :- animal(X), flies(X).
```

*Upon receiving the query* `?- bird(X).`*, Prolog starts to find all the objects that it can prove to be birds. It first tries to satisfy the first condition, i.e.,* `animal(X)`*. By considering the knowledge base from the top to the bottom, it first finds that* `X = tiger` *satisfies* `animal(X)`*. It keeps a pointer to this knowledge base location and tries to satisfy the next condition, i.e.,* `flies(X)` *with* `X = tiger`*, which fails. Now, it goes back to that pointer (which was stored for the first condition), and continues from that point (this part is called backtracking). The next guess would be* `X = elephant`*, which similarly fails. After another backtracking,* `X = dove` *would be guessed, which, this time, due to the fact* `flies(dove)` *is proved to be correct. Therefore,* `X = dove` *is returned. If more solutions are requested, another backtracking takes place, but since there is no more object that satisfies* `animal(X)`*, the search fails.* □

The backtracking strategy is similar to a search in the depth-first order. For a rule of the form `r :- r1, r2, ..., rn`, Prolog first tries to satisfies `r1` and if it does, it keeps a pointer at the corresponding location in its knowledge base corresponding to `r1`. Then it does the same for `r2` using the possible variable assignments it finds by satisfying `r1`. This process would continue until either it could satisfy `rn` as well (successful return), or for some `i` (`i`$\leqslant$`n`), it cannot satisfy `ri` (unsuccessful return). Upon an unsuccessful search Prolog goes back to the last pointer it has kept in the knowledge base and tries to satisfy the corresponding clause from that point forward. This would also happen if the search has been successful and more solutions are required.

Based on how Prolog searches its knowledge base to find solutions for a given query, we make a simple comparison between the composition approaches of Chapters 5 and 6 and a composition approach that would use Prolog.

**Example 7.3** *Consider the following simplistic knowledge base for a composition planning engine based on Prolog. Note that the rules in this knowledge base are*

*not complete for solving the composition planning as proposed by composition algorithms of Chapters 5 and 6. However, since the composition planning using Prolog must take advantage of recursion, one such recursive rule is given in this sample knowledge base.*

```
  1: component(c1, [(a1,1)], [(a2,1)]).
  2: component(c2, [(a2,1)], [(a3,1)]).
  3: component(c3, [(a3,1)], [(a4,1)]).
     ...
  n: component(cn, [(an,1)], [(b,1)]).
n+1: component(d, [(a2,1)], [(b,1)]).
n+2: composition(IL, OL, C) :- component(C, IL, OL).
n+3: composition(IL, OL, C) :- component(C1, IL, TL),
                               composition(C2, TL, OL),
                               C=(C1*C2).
```

*The numbers appearing before the knowledge base clauses are there to simplify addressing them in this example. Also, the* `*` *symbol in the last rule refers to synchronization and is a substitute for the* $\odot$ *symbol. Clause* `1`*, for instance, indicates that component* `c1` *receives one instance of data type* `a1` *as its input, and returns one instance of data type* `a2` *as its output. Clauses* `n+2` *and* `n+3` *define two possible ways of finding a solution for the problem. In clause* `n+2`*, component* `C` *is defined to be the solution for a request with inputs* `IL` *and outputs* `OL`*, if* `C` *receives and returns the exact same parameters. However, in clause* `n+3`*, a synchronization is returned, in which component* `C1` *receives* `IL` *and returns* `TL`*, while there is a composition* `C2` *that receives* `TL` *and returns* `OL` *(the recursive part).*

*Now, suppose that the query* `?- composition([(a1,1)], [(b,1)], C)` *is submitted against this knowledge base and only the first solution found by Prolog would suffice. Upon receiving this query, Prolog tries to apply the clause* `n+2`*, which would be unsuccessful. Then it uses the clause* `n+3`*, which breaks down the request to* `component(c1, [(a1,1)], [(a2,1)])`*,* `composition(C2, [(a2,1)], [(b,1)])`*, and* `C=(c1*C2)`*. It keeps a pointer to clause* `1` *at this step, and continues to satisfy the new query* `composition(C2, [(a2,1)], [(b,1)])`*. This process goes on in a similar way until the solution* `C=(c1*(c2*(...*(cn-1*cn))))` *is returned as the overall result. However, we can easily see that* `C=(c1*d)` *is also a solution, which is much simpler.* □

In the above example, Prolog fails to find the shortest path solution, because of the two properties mentioned earlier; i.e., it processes the knowledge base sequentially,

and it uses backtracking. Similarly, in case more than one solution is required, Prolog cannot guarantee to return the shortest path ones. This is a huge disadvantage comparing to our composition approaches that guarantee to return shortest path solutions[2]. The graph-based approach of Chapter 5 does so by using the BFS search in finding appropriate graph paths. Also, the reasoning-based approach of Chapter 6 creates instances according to a multi-level graph in a BFS manner, which guarantees to find shortest path solutions.

Aside from the reasoning-based languages and tools, specification languages might also be useful in solving the composition planning problem. One of these languages is Alloy, which is a lightweight language for modeling software systems. It draws many of its ideas from Z [1]: in particular, representing all data structures with sets and relations, and representing behavior and properties with simple formulas. Although Alloy was designed to be flexible and expressive, unlike Z, it is amenable to fully automatic simulation and checking. A simple first order logic constraint solver based on reduction to SAT can check properties of Alloy models. Alloy has been applied to problems from very different domains, from checking the conventions of Microsoft COM to debugging the design of a name server [56, 57]. Detailed information about Alloy can be found in [55].

To study if Alloy is a good candidate in modeling the composition planning problem, we tried to perform an initial evaluation by implementing the forward chaining algorithm for Horn clauses. The corresponding specification is as follows:

```
open util/ordering[State] as steps

sig Literal {}

sig Clause {
  left: set Literal,
  right: Literal
}

sig State {
  true: set Literal,
  usable: set Clause,
```

[2]The only way we can make a Prolog engine process the knowledge base in a breadth-first order is to do some form of meta-programming in order to simulate the breadth-first processing. Because of the complexities involved in developing such a simulation program, we do not discuss it in this thesis.

```
  uses: lone Clause
}

fact initialState {
  let s0=steps/first {
    s0.true = req.left
    s0.usable = {c: Clause | #c.left >0 && #c.right>0} - req
    no s0.uses
  }
}

pred NoChange [s: State, s': State] {
  s'.true = s.true
  s'.usable = s.usable
  s'.uses = s.uses
}

pred Use [s: State, s': State] {
  some c: Clause | {
    c in s.usable
    c.left in s.true
    c.right not in s.true
    s'.true = s.true + c.right
    s'.usable = s.usable - c
    s'.uses = c
  }
}

fact stateTransition {
  all s: State-last |
    let s'=steps/next[s] |
      ( NoChange[s,s'] || Use[s,s'] )
}

// example: c1: a->b, c2: b->c, c3: c->d. Is req: a->d true?
one sig a, b, c, d extends Literal {}
one sig c1, c2, c3, req extends Clause {}
fact { left = c1->a + c2->b + c3->c + req->a }
fact { right = c1->b + c2->c + c3->d + req->d }
run { req.right in steps/last.true } for 1 but 4 State
```

This specification implements the forward chaining approach by introducing the concept of states. `Literal` and `Clause` represent literals and Horn clauses in the knowledge base, while `State` captures any step in Algorithm 6.1. The request `req` is also a Horn clause that the algorithm tries to prove/disprove. Each state contains a set of literals marked as true (`true`), a set of clauses that have not been used yet (`usable`), and a single clause `uses` that have been used before the current state is reached. In the initial state, all the literals in the left-hand side of `req` form the set `true`, all the clauses with some literal in their both left and right-hand sides form the set `usable`, and there would be no clause in `uses` as no clause from the knowledge base has been used yet.

In order to move from one state `s` to another state `s'`, there should be some knowledge base clause `c` in `s.usable` where the left-hand side literals of `c` are already marked as true (`c.left in s.true`), while its right-hand side literal is not (`c.right not in s.true`). As a result of this transition from `s` to `s'`, `c.right` would be added to the literals marked as true (`s'.true = s.true + c.right`), `c` would be removed from the clauses that can be used (`s'.usable = s.usable - c`), and `c` would be marked as the clause used in this transition (`s'.uses = c`). This transition is specified in the predicate `Use`.

There is another predicate `NoChange` in this specification indicating that there might be transitions in which no change is made in the attributes of the states. This `NoChange` predicate is necessary in case Alloy analyzer is considering more state instances than the actual required number. The fact `stateTransition` formulates a move from one state to its next according to this explanation.

To start the analysis we need to specify the knowledge base, i.e., literals and clauses, plus the requested Horn clause. This is done in the last section of the above Alloy specification through some `sig` and `fact` expressions. The last line, i.e., `run { req.right in steps/last.true } for 1 but 4 State`, asks Alloy to start analyzing the model. More precisely, it asks Alloy to do the analysis by creating one instance of the top level data types introduced by `sig`, except for `State` which would have four instances. If the analysis is successful and given facts and predicates are proved to be consistent, it means that the algorithm has a solution based on the given constraints.

In the above example clauses $\texttt{a} \rightarrow \texttt{b}$, $\texttt{b} \rightarrow \texttt{c}$ and $\texttt{c} \rightarrow \texttt{d}$ form the knowledge base, and the request is the clause $\texttt{a} \rightarrow \texttt{d}$. The result returned by Alloy Analyzer is shown in Figure 7.12. The attributes of the four states in this figure are shown in Table 7.4.

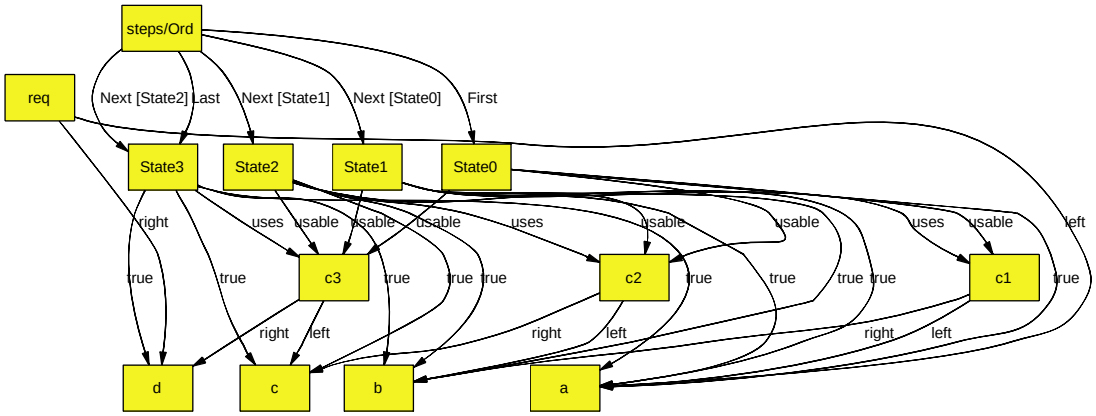



Figure 7.12: The solution returned by Alloy Analyzer for the given Alloy specification.

| | true | usable | uses |
|---|---|---|---|
| State0 | {a} | {c1,c2,c3} | |
| State1 | {a,b} | {c2,c3} | c1 |
| State2 | {a,b,c} | {c3} | c2 |
| State3 | {a,b,c,d} | {} | c3 |

Table 7.4: Different states found by Alloy Analyzer for the given Alloy specification.

Running this model with three states would not return a solution. This shows that at least four states are required to find a solution. Since the number of clauses used from the knowledge base is the number of states minus one, we conclude that using three knowledge base clauses we can satisfy the request.

We can make minor changes to the above Alloy code to make it a model of Algorithm 6.2, in which we ignored the cardinality of input/output data types. That way, by running the model on a specific repository, we can find a sequence of components that could build the requested one. However, this sequence is not always the best solution.

**Example 7.4** *Consider the repository of components* $C_1 : a \rightarrow b$, $C_2 : a \rightarrow c$, $C_3 : \{b, c\} \rightarrow d$ *and the request* $a \rightarrow d$. *Running the above Alloy code on this example would return the sequence* $C_1$, $C_2$, $C_3$ *as a possible solution. It does not return any solution implying the composition* $(C_1 \parallel C_2) \odot C_3$, *which is the best answer.* □

To achieve the best solution many improvements must be made in the above model, which obviously adds to its complexity. If we try to model the generic algorithm, Algorithm 6.3, by involving the cardinalities, this complexity is expected to increase substantially, especially if we consider Alloy's inventor's claim regarding the difficultly of modeling integer arithmetic using Alloy [26]. Moreover, since there is no built-in notion of states in Alloy and its specifications cannot be automatically checked against properties containing temporal operators [26], we would need to simulate these temporality the way we did in the above Alloy example. This could be considered as another drawback in implementing the generic composition algorithm using Alloy.

Moreover, in order to avoid false negatives, i.e., getting a negative result for a given request where a solution does exist, we need to have an estimate about the number of states required by Alloy in processing the model. For example, if we use the command `run { req.right in steps/last.true } for 1 but 4 State` in the above model, Alloy Analyzer would find only one solution with four states. Using `run { req.right in steps/last.true } for 1 but 3 State` would not lead to any solution though, giving the impression that the model is unsatisfiable. Using `run { req.right in steps/last.true } for 1 but 5 State`, on the other hand, would lead to four solutions each with five states. For this really small and easy example we know that the solution with four states is the best one. However, when the knowledge base is much larger, it becomes impossible to estimate the number of states needed by the analyzer in order to avoid false negatives and, also, non-optimal solutions, which use many more states than actually needed. If the estimate is too low, the analyzer fails to find a solution. On the other hand, if the estimate is too high, the analyzer finds a solution that is far from the best one. Moreover, if the scope and the number of instances in a model becomes large, Alloy Analyzer would become quite inefficient in processing the model.

Finally, when there are multiple solutions for a given model, the order of solutions returned by Alloy depends on the implementation of its underlying SAT solver, and is outside the control of Alloy. Therefore, there is no guarantee that the first solution returned is the best solution, or in terms of the composition planning problem, the one corresponding to the shortest path (which uses least number of components).

## 7.4 Towards A Practical Solution

In Chapters 1 and 2 we discussed web services and their composition as the potential target for the automatic component composition we study in this thesis. That is why in the previous section we picked a number of web services to create a realistic component repository, based on their statistical information, in studying the average-case performance of the proposed composition planning approach. The parameters we focused on were the number of inputs/outputs in each web service, and the cardinality of each data type appearing as an input/output.

In this section we study, in more details, how our composition planning approach can be practically used for web services. To do so, we analyze the structure of WSDL documents, as standard and approved specifications for the functionality of web services, and how to use them. Then, we explain how we can extract necessary information from these WSDL specifications in order to create the component repository and implement the planning approach. In this section, discussions and examples on WSDL are taken from [105].

### 7.4.1 WSDL

A WSDL 2.0 document contains the following generic structure:

```
<description>

  <documentation>
    <!-- additional documentation -->
  </documentation>

  <types>
    <!-- definition of types -->
  </types>

  <interface>
    <!-- definition of an interface -->
  <interface>

  <binding>
    <!-- definition of a binding -->
  </binding>

  <service>
```

```
    <!-- definition of the web service -->
  </service>

</description>
```

As we see in this structure, the specification of a web service is put inside a root `description` element. Different namespaces used throughout this specification are defined as the parameters of this element. The following is an example of the `description` element.

```
<description
  xmlns="http://www.w3.org/ns/wsdl"
  targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
  xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
  . . . >

  . . .
</description>
```

In simple terms, we define different vocabularies that are used inside the current WSDL document in this `description` element. In this specification, `xmlns` is the XML namespace for WSDL 2.0. Therefore, every other namespace is specified by a `xmlns:` prefix, which indicates that the given URI is a namespace. The `targetNamespace` attribute defines the default namespace for the current WSDL document. This means that all the newly introduced terms in the specification would fall in the target namespace. Note that it is not a namespace declaration, as there is no `xmlns:` prefix attached to it. The target namespace is the default namespace for the given WSDL specification. The next item specified by `xmlns:tns`, which is an actual namespace declaration also referring to the target namespace, is used in case we want to use the prefix `tns:` to emphasize that its following term is from the target namespace. We return to the notion of target namespaces with some more examples later in this section.

Since each web service would communicate with the outside world by sending and receiving messages, the type of these messages must be defined properly in the WSDL document. Their definitions would reside inside the `types` element, which is a child of the root `description` element. Here is an example of the `types` element, which corresponds to a web service for checking the availability of some hotel.

```
<description
  xmlns="http://www.w3.org/ns/wsdl"
```

```
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
    . . . >

    . . .

    <types>
      <xs:schema
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://greath.example.com/2004/schemas/resSvc"
        xmlns="http://greath.example.com/2004/schemas/resSvc">

        <xs:element name="checkAvailability" type="tCheckAvailability"/>
        <xs:complexType name="tCheckAvailability">
          <xs:sequence>
            <xs:element  name="checkInDate" type="xs:date"/>
            <xs:element  name="checkOutDate" type="xs:date"/>
            <xs:element  name="roomType" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>

        <xs:element name="checkAvailabilityResponse" type="xs:double"/>

        <xs:element name="invalidDataError" type="xs:string"/>

      </xs:schema>
    </types>

    . . .
  </description>
```

A new target namespace, `http://greath.example.com/2004/schemas/resSvc`, is being used in this message type definition. In this specification, message types `checkAvailability`, `checkAvailabilityResponse`, and `invalidDataError` are defined as XML elements (`xs:element`). The `checkAvailability` element is of type `tCheckAvailability`. Note that since there is no namespace attached to this type, it is considered to be from the above target namespace. Later, `tCheckAvailability` is defined as a complex type containing three simpler types in order (because of `xs:sequence` ordering info): a `checkInDate` of type `xs:date`, a `checkOutDate` of type `xs:date`, and a `roomType` of type `xs:string`. The `checkAvailabilityResponse` and `invalidDataError` elements, which are again from the target namespace, are of type XML Schema `double` and `string`, respectively.

Since XML Schema data types, such as `string`, `date` and `double`, are too general for the purpose of automated composition planning, we can attach the given element names to make them represent more specific concepts. Ignoring the element names in the above specification would result in `roomType` and `invalidDataError` being semantically similar terms, as they are both of type XML Schema `string`. However, since they are two different terms from the target namespace, they should not be considered semantically similar. As a result, in order to disallow false positives in the search for valid composition plans we would consider name-type pairs instead of only types.

The order of elements in a complex type could be of three kinds: `sequence`, `all`, or `choice`. The `sequence` indicator explained above is similar to the way programming languages define the signature of their functions. The `all` ordering indicates that the child elements may appear in the complex type in any order as long as they all appear in it. The `choice` indicator specifies that one or the other child can occur.

Similarly, attributes `minoccurs` or `maxoccurs` might be attached to elements of a complex data type to indicate the cardinality of the corresponding data type element in input or output messages. As an example, the element definition `<xs:element minoccurs="1" maxoccur="1" name="checkInDate" type="xs:date"/>` indicates that only one `checkInDate` element would appear in the complex type. The default value for both `minoccurs` and `maxoccurs` is 1.

The next part of a WSDL description defines interfaces as a set of operations each representing a simple interaction between the service and its client. Along with each operation the types of messages it can receive and return, and also the expected order of those messages, called the *message exchange pattern*, are defined. An example of this message exchange pattern is *in-out*, which indicates that if the client sends a message to the service, the service will respond with either the reply to the sent message or a fault message. Here is an example:

```
<description
  xmlns="http://www.w3.org/ns/wsdl"
  targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
  xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
  xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
  . . .
  xmlns:wsdlx="http://www.w3.org/ns/wsdl-extensions">

  . . .
```

```
  <types>
    . . .
  </types>

  <interface name = "reservationInterface" >

    <fault name = "invalidDataFault"
      element = "ghns:invalidDataError"/>

    <operation name="opCheckAvailability"
      pattern="http://www.w3.org/ns/wsdl/in-out"
      wsdlx:safe = "true">
        <input messageLabel="In"
          element="ghns:checkAvailability" />
        <output messageLabel="Out"
          element="ghns:checkAvailabilityResponse" />
        <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
    </operation>

  </interface>

  . . .
</description>
```

This specification defines a single interface `reservationInterface` for this web service. This interface has a fault message named `invalidDataFault` which is of type `invalidDataError` (namespace `http://greath.example.com/2004/schemas/resSvc`). It has a single operation `opCheckAvailability`, which will be referenced later in the specification, and uses the in-out message exchange pattern. The `safe` property of the operation shows that invoking this operation will not obligate the client in any way (such as making him/her to buy something). Then the input and output of the operation are defined. The `In` and `Out` labels somehow emphasize the corresponding message exchange pattern. Finally, the output fault of the operation is defined referring to a previously defined fault in the interface.

The `binding` part in the WSDL specification, which comes after the `interface` definition, defines the communication protocols used by the web service. So far, the WSDL document has defined *what* the operations supported by the web service are. In the `binding` section, it defines *how* those operations can be actually invoked. It basically specifies binding details for each operation and the fault defined earlier in the specification. The following is an example of such binding definitions:

```
<description
  xmlns="http://www.w3.org/ns/wsdl"
  targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
  xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
  xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
  xmlns:wsoap= "http://www.w3.org/ns/wsdl/soap"
  xmlns:soap= "http://www.w3.org/2003/05/soap-envelope">

  . . .

  <types>
    . . .
  </types>

  <interface  name = "reservationInterface" >
    . . .
  </interface>

  <binding name="reservationSOAPBinding"
    interface="tns:reservationInterface"
    type="http://www.w3.org/ns/wsdl/soap"
    wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">

    <operation ref="tns:opCheckAvailability"
      wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>

    <fault ref="tns:invalidDataFault"
      wsoap:code="soap:Sender"/>

  </binding>

  . . .
</description>
```

Two namespaces `xmlns:wsoap` and `xmlns:soap` are added to the description element of this specification. `xmlns:wsoap` is used for the SOAP binding extensions defined in WSDL 2.0; and `xmlns:soap` is used for the SOAP specification itself. A binding element with name `reservationSOAPBinding` has been defined in this specification to specify the binding information for `reservationInterface` defined above. The type of message format and the transmission protocol used for this binding is SOAP and HTTP, respectively. The next part references the `opCheckAvailability` operation defined above to specify its binding details. The `wsoap:mep` indicates that GET is used as the corresponding HTTP method. The last part provides the binding

information for the fault element defined earlier in the interface section by specifying the SOAP fault code that causes this fault message to be sent.

Now that the *what* and *how* questions regarding the functionality of the web service are answered, we need to answer the *where* question, i.e., where this service can be accessed. This is done using the `service` element. Each `service` element specifies one interface that the service supports, and a list of endpoint locations where the service can be accessed. Each endpoint references a previously defined binding to indicate the protocols and transmission formats used at that endpoint. Here is an example:

```
<description
  xmlns="http://www.w3.org/ns/wsdl"
  targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
  xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
  xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
  xmlns:wsoap= "http://www.w3.org/ns/wsdl/soap"
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  . . .

  <types>
    . . .
  </types>

  <interface  name = "reservationInterface" >
    . . .
  </interface>

  <binding name="reservationSOAPBinding"
  interface="tns:reservationInterface"
    . . . >
    . . .
  </binding>

  <service name="reservationService"
    interface="tns:reservationInterface">

    <endpoint name="reservationEndpoint"
      binding="tns:reservationSOAPBinding"
      address="http://greath.example.com/2004/reservation"/>

  </service>
</description>
```

The service `reservationService` is defined in this specification which supports `reservationInterface` defined in the above `interface` element. Then, an endpoint `reservationEndpoint` is defined for this service at which the previously defined binding `reservationSOAPBinding` is used. The address attribute of this endpoint defines the physical address at which the service can be accessed using the above binding.

The only part left is the documentation of a WSDL specification. Although the basic information on how to use the service is given by the WSDL structure explained so far, additional explanation might be needed, for example to provide the meaning of the messages, their constraints, .... The optional `documentation` element is used for this purpose and contains human-readable contents. This element can be used at different places in the WSDL specification, such as the beginning.

```
<description
  . . . >

  <documentation>
    This document describes the hotel reservation Web service.
    Additional requirements for use of this service -- beyond
    what WSDL 2.0 is able to describe -- are available at
    http://greath.example.com/2004/reservation-documentation.html
  </documentation>

  . . .
</description>
```

The complete WSDL specification of the service we used in the above examples is the following:

```
<description
  xmlns="http://www.w3.org/ns/wsdl"
  targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
  xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
  xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
  xmlns:wsoap= "http://www.w3.org/ns/wsdl/soap"
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsdlx= "http://www.w3.org/ns/wsdl-extensions">

  <documentation>
    This document describes the hotel reservation Web service.
    Additional requirements for use of this service -- beyond
    what WSDL 2.0 is able to describe -- are available at
```

```
  http://greath.example.com/2004/reservation-documentation.html
</documentation>

<types>
  <xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://greath.example.com/2004/schemas/resSvc"
    xmlns="http://greath.example.com/2004/schemas/resSvc">

    <xs:element name="checkAvailability" type="tCheckAvailability"/>
    <xs:complexType name="tCheckAvailability">
      <xs:sequence>
        <xs:element  name="checkInDate" type="xs:date"/>
        <xs:element  name="checkOutDate" type="xs:date"/>
        <xs:element  name="roomType" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>

    <xs:element name="checkAvailabilityResponse" type="xs:double"/>

    <xs:element name="invalidDataError" type="xs:string"/>

  </xs:schema>
</types>

<interface name = "reservationInterface" >
  <fault name = "invalidDataFault"
    element = "ghns:invalidDataError"/>

  <operation name="opCheckAvailability"
    pattern="http://www.w3.org/ns/wsdl/in-out"
    style="http://www.w3.org/ns/wsdl/style/iri"
    wsdlx:safe = "true">
      <input messageLabel="In"
        element="ghns:checkAvailability" />
      <output messageLabel="Out"
        element="ghns:checkAvailabilityResponse" />
      <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
  </operation>
</interface>

<binding name="reservationSOAPBinding"
  interface="tns:reservationInterface"
  type="http://www.w3.org/ns/wsdl/soap"
  wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
```

```
      <fault ref="tns:invalidDataFault"
        wsoap:code="soap:Sender"/>

      <operation ref="tns:opCheckAvailability"
        wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>
    </binding>

    <service name="reservationService"
      interface="tns:reservationInterface">

      <endpoint name="reservationEndpoint"
        binding="tns:reservationSOAPBinding"
        address ="http://greath.example.com/2004/reservation"/>
    </service>
  </description>
```

There have been some syntactical changes in the transition from WSDL 1.1 to WSDL 2.0, the latter of which is now a W3C recommendation. For example, some of the element names have changed, such as `definition` to `description`, `portType` ro `interface`, and `port` to `endpoint`. Some main elements have been removed, such as `message`, and some elements have been refined. In general, WSDL 2.0 documents are simpler to understand and more structured, compared to WSDL 1.1 documents. The World Wide Web Consortium (W3C) offers an online converter for transforming WSDL 1.1 specifications into their equivalents in WSDL 2.0.

### 7.4.2 Compatibility with the Proposed Approach

In the proposed composition planning approach the information we need for each offered functionality is its inputs, outputs, and their corresponding cardinality. Considering WSDL 2.0 specifications, each triplet (service, interface, operation) would indicate a unique functionality and can be modeled as a component in our composition planning repository. For example, in the above WSDL specification the triplet `(tns:reservationService, tns:reservationInterface, tns:opCheckAvailability)` would specify the provided functionality, where `tns` is the target namespace defined in the document.

The `element` parameter in the definition of an operation input/output would specify its corresponding data type. For example, in the above WSDL specification, `ghns:checkAvailability` is the name of the input element, which is of type

`tns:tCheckAvailability`. Similarly, `ghns:checkAvailabilityResponse` is the corresponding output element, which is of type `xs:double`. Although the data type `tns:tCheckAvailability` is specific enough, the data type `xs:double` is not. In other words, `tns:tCheckAvailability` provides enough information about what this component expects to receive as an input. On the other hand, this is not the case for `xs:double`, as we do not know if this double number stands for a cost, temperature, distance, or something else. However, if we attach the element name `ghns:checkAvailabilityResponse` to it we would have a better understanding about what this component returns. That is why we pick the pair (element name, element type) to indicate the data type of inputs/outputs for each component. In creating realistic composite repositories during the performance evaluation, we assumed that data types are uniformly distributed among the repository components. As shown in Figure 7.13 choosing the pair (element name, element type) is expected to conform much better to this assumption, rather than choosing only element types.

In Figure 7.13 and in the top chart we see that the frequency of element types hugely differs for the types at the beginning. Those are in fact corresponding to XML Schema data types, such as `string` and `double`, which appear frequently in WSDL specifications. The other parts of this chart with much lower frequencies correspond to data types that are defined in WSDL documents, such as `tsn:tCheckAvailability` in the previous WSDL example. Since these locally defined element types are normally used by only the operations defined in the same WSDL document we would expect them not to be used by other web services.

In the bottom chart in Figure 7.13 we see that by adding the element names to data types the type distribution becomes much closer to a uniform distribution. In fact, the frequencies are quite dependent on the number of operations each service provides. The reason is that, as mentioned above, element names are usually local to WSDL documents, and the more the number of operations in a WSDL document, the more each locally defined element name is likely to be used. In other words, if we had picked only one operation from each web service this bottom chart would have been much more similar to a uniform distribution. This chart shows that creating semantic links between different namespaces is quite necessary, as it will highly increase the chance of finding composition plans. This can be done by introducing dummy components to the repository, where, for example, each component represents a data type subsumption or equivalency between two terms in different namespaces, or it represents a composite type and its constituent component types. We did this in Section 5.2.2.

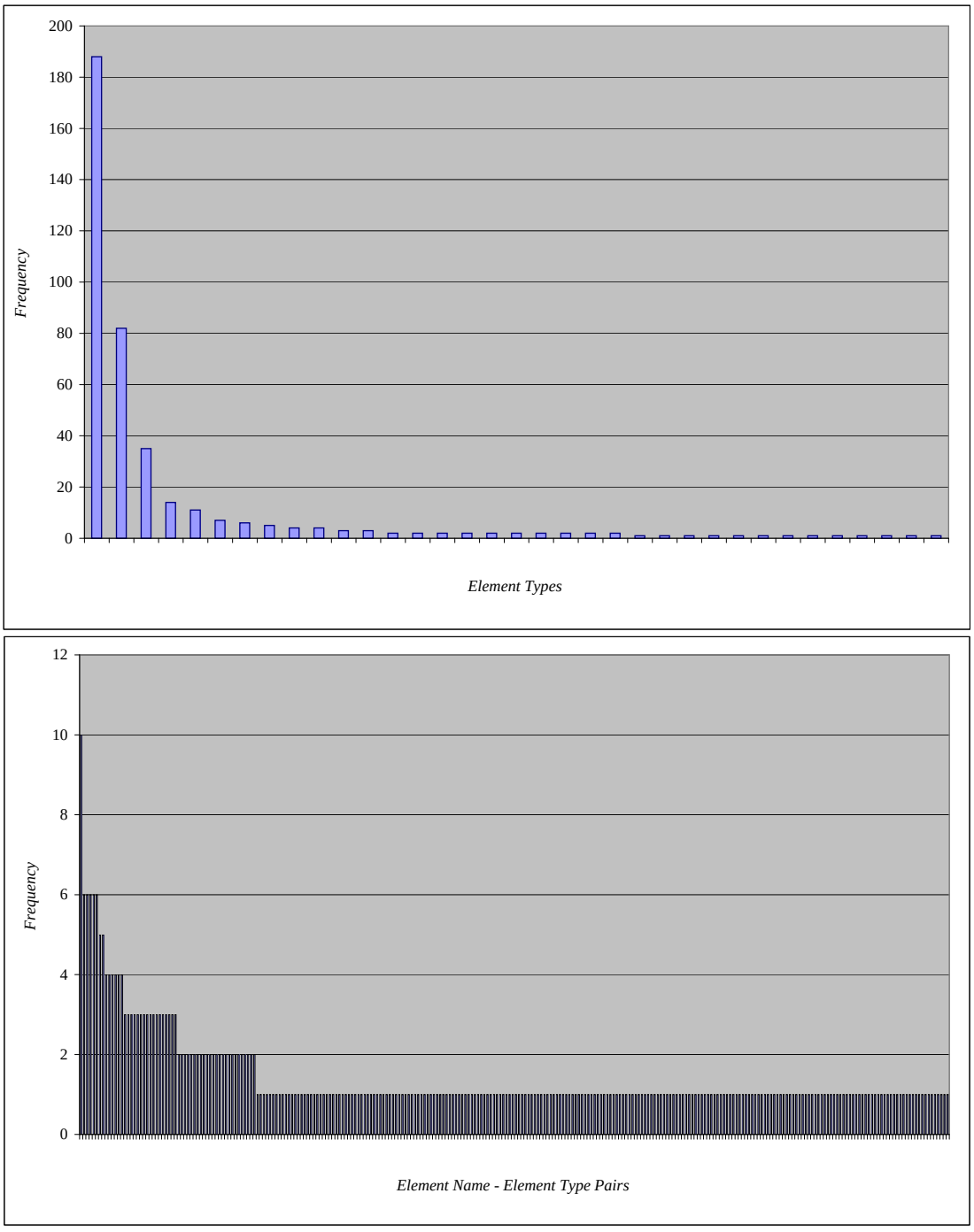



Figure 7.13: The distribution of WSDL element types in operation inputs/outpus of web services (top) and the distribution of WSDL element name and WSDL element types in operation inputs/outpus of web services (bottom).

As mentioned earlier in this section, the cardinality of input/output data types is specified using `minoccurs` and `maxoccurs` attributes representing a range of values. In the current version of our composition planning approach we consider a single constant number for the cardinality of data types. Also, we used web services which had a single cardinality value in their input/output data types (where `minoccur` and `maxoccur` specified the same number) for creating a realistic repository. Our approach is still applicable if only inputs are assigned a cardinality range. For example, if a component has a single input of type $t$ specified by `minoccurs="1"` and `maxoccurs="2"` attributes, we can use this component with either one instance of type $t$, or two instances of type $t$. However, if an output is specified with a cardinality range some nondeterminism appears, as we do not know how many instances would actually be created by that component. In case the precondition determines how a component behaves, involving preconditions and effects might still make the current approach work by avoiding this nondeterminism.

In the logic-based composition planning approach we did not capture composite types and their component types. As this would improve the composition plans returned by the approach, we can use the same technique we used in the dependency graph (Section 5.2.2).

In WSDL specifications, different order indicators might be used for components of a composite type, as in the above WSDL example `sequence` is used for the composite type `tCheckAvailability`. Other indicators are `choice` and `all`. Considering operation inputs, for a `sequence` indicator, we can assume that the elements are given to the component in the same order; while for a `choice` indicator, whenever one of the elements is ready, the component can be triggered. The only potential problem is when a `choice` indicator appears in an output, meaning that it is not known what data type is returned by the component, which creates another type of nondeterminism. In practice, this is not expected to happen, as the study of a large number of WSDL documents shows that the output composite types do not use the `choice` order indicator.

As the overall conclusion, with some minor modifications, we can use the composition planning approach discussed in this thesis for web services and create composite service plans for given requests.

## 7.5 Summary

In this chapter we evaluated the proposed composition planning approaches from different points of view. First, we compared their running time and space complexities and discussed how each of them would be a suitable option for specific domains. Second, we explained the experimental results of implementing the reasoning-based approach, and showed that its expected run time performance would be quite far from the calculated worst case. Third, we picked Prolog and Alloy as two reasoning tools for solving the composition planning problem, and discussed their disadvantages compared to our proposed solutions. Finally, we studied web services specification documents in details, and explained how our composition planning approaches could be applied to web services in practice. We also emphasized on the steps we still need to take to realize such an application.

# Chapter 8

# Conclusion

In the final chapter of this thesis, we provide a short summary of the materials covered in the previous chapters. Also, we suggest some of the directions that could be followed in continuing this work.

## 8.1 Summary

In this thesis we studied a challenging, yet interesting problem towards the automation of component-based software development. To achieve this goal, the first requirement is the availability of a rich enough repository of already developed software components that, if put together appropriately, could constitute new components with new functionalities. Then, given the specification of a new component to be built, the expectation would be finding a set of components from this repository that, by communicating with each other, provide the requested functionality. Therefore, the challenge would be how to find this set of repository components and necessary communications among them. We called this problem the composition problem.

We proposed a top level architecture of a component composition engine that would solve the overall problem. That architecture identified different subproblems of the composition problem, from extracting behavioral information from the available components to publishing a composition as a new component ready to be used by its users. From these subproblems, the most theoretically challenging one would be the composition planning or synthesis, which is the focus of this thesis. Composition planning refers to finding a plan involving the participating components, their temporal order of execution, and the communications among them that

provides the requested behavior for a software component.

In order to solve the composition planning problem we picked a specific subset, in which the repository components are all stateless, meaning that they receive some inputs and then return some outputs as a result. We studied different types of component composition, including the sequential composition, parallel composition, conditional composition, and synchronization. To formally represent how each of these compositions work, we took advantage of interface automata and existing process algebras and proposed a new process algebraic model called the composition algebra. Composition algebra is a minimal model that supports all these different types of composition by borrowing some properties of CSP, CCS, and interface automata.

In the first composition approach proposed in this thesis, the repository of available components was modeled by a graph structure, the dependency graph, in which nodes represent data types, and edges represent connections among these data types. These connections might be functional, which are imposed by repository components, or semantic. The search for a composite component to satisfy a request was narrowed down to the reachability of all output data types in the request from its input data types in the dependency graph. However, there were some constraints that made the problem more complex than a simple reachability search. One of these constraints is related to the notion of cardinality, and the fact that each component might receive/return multiple instances of the same data type. As another constraint, we assumed that the outputs of each component is dependent on all its inputs. These restrictions together converted a linear graph search to a search with exponential time worst-case complexity. We reasoned why the approach is expected to perform much better in realistic situations.

The second approach takes advantage of a reasoning algorithm for Horn clauses. It implements the reachability procedure discussed above in some other way which is easier to understand. Using this approach a multi-level graph is built, in each level of which there are instances of data types created by repository components. This graph is extended to a point at which either all the necessary instances are created, or no more required instance can ever be created. In the former case, a composition can be found, while in the latter, the search is terminated unsuccessfully. In a successful search, the links between instances created in the multi-level graph and the corresponding stored information lead us to the specification of the composite component in terms of the involved components and composition types. Similar to the first approach, the worst-case running time complexity of this one is also exponential. However, after implementing the approach and applying some

performance heuristics, we presented experimental results which indicate a much better performance complexity. The performance heuristics used in implementing the reasoning-based approach borrow some techniques from the graph-based approach. In general, the expectation is that merging the two approaches would provide better running time results.

We considered Prolog and Alloy as two available reasoning tools, and compared their performance against the performance of the proposed composition planning approaches. Specifically, we explained why Prolog and Alloy are not good choices when we are searching for shortest-path solutions. We also studied WSDL documents as the specification documents of web services and discussed the adaptability of the proposed approaches for web services composition.

## 8.2 Future Work

The work presented in this thesis can be extended from different points of view. We explain some of these directions in this section. We start with higher level suggestions regarding the composition problem, and then move to the specific plans for composition planning.

As shown in Figure 2.2, other than the composition planning, there are other subproblems in the composition problem. We addressed some of the works in progress in those subproblems in Chapter 3. Since the research in those areas is in its initial stages there is no known solution that is widely accepted by the community.

We studied a simple form of composition planning, in which we assumed all repository components and the request are stateless. Although, a large percentage of components are stateless, this assumption is restrictive. As an example, two components $C_1 = a \cdot \overline{b} \cdot \overline{c}$ and $C_2 = b \cdot c \cdot \overline{d}$ that are not stateless, if synchronized, result in a stateless component whose behavior is $a \cdot \overline{d}$. Therefore, when we put a limitation that repository components are stateless, we might lose or decrease the chance of finding a valid composition. Improvements in this regard would include allowing the repository to contain all types of components in search for a stateless component, or not restricting both the repository components and the request to be stateless. However, these changes are expected to add much to the complexity of the composition approach.

In Chapter 5 we explained how we can use auxiliary components in order to create semantic relations among the repository data types that are realized by

GenSpec and CD edges between graph nodes. Although we did not apply auxiliary components in the reasoning-based approach, we can simply improve the approach by using these components. Auxiliary components can be specified similar to other repository components, and therefore, can be simply added to the repository. The only thing that should be considered is that GenSpec components come with an external behavior, but they do not actually do anything. As a result, when a composition takes advantage of a GenSpec component some considerations have to be taken into account. For instance, if we know that type $c$ is a subtype of type $d$, we would add a component $B = c \cdot \overline{d}$ to the repository. If $A = a \cdot \overline{b}$ and $C = (b \mid\mid d) \cdot \overline{e}$ are also repository components and the request $(a \mid\mid c) \cdot \overline{e}$ is submitted against this repository, the composition $(A \mid\mid B) \odot C$ would be returned by the current reasoning-based approach. However, since $B$ is a dummy component, the correct solution would be $A \odot C$. In this regard, we leave the necessary improvements to the approach as a future work.

Since the goal of this thesis is automatic signature matching in component composition, we did not consider preconditions and effects of repository components along with their nonfunctional properties. Therefore, it is quite possible that the composition planning approaches of this thesis return a composition which has a valid signature, but is not valid in general since, for instance, the effect of one component in the composition is not compatible with the precondition of another. As a future work, preconditions, effects and nonfunctional properties of repository components have to be considered as well.

We mentioned in Chapters 1 and 2 that the main application of the composition problem is in the world of web services. Inputs and outputs of web services are specified using namespaces, and these namespaces usually represent different ontologies. Since a large number of ontologies exist based on which published web services are specified, it is vital to make semantic connections among these ontologies to increase the chance of finding valid compositions for service requests. This problem, that could be referred to as ontology matching, is another related problem that should be solved.

# References


[1] Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer. Specification Language. In *On the Construction of Programs*, pages 343–410. Cambridge University Press, 1980. 147

[2] Sudhir Agarwal and Anupriya Ankolekar. Automatic Matchmaking of Web Sservices. In *WWW '06: Proceedings of the 15th International Conference on World Wide Web*, pages 1057–1058. ACM Press, 2006. 5, 17

[3] Sudhir Agarwal and Rudi Studer. Automatic Matchmaking of Web Services. In *ICWS '06: Proceedings of the 2006 IEEE International Conference on Web Services*, pages 45–54, 2006. 17

[4] Rohit Aggarwal, Kunal Verma, John Miller, and William Milnor. Constraint Driven Web Service Composition in METEOR-S. In *SCC '04: Proceedings of the 2004 IEEE International Conference on Services Computing*, pages 23–30. IEEE Computer Society, 2004. 24

[5] Marco Aiello, Mike P. Papazoglou, Jian Yang, M. Carman, Marco Pistore, Luciano Serafini, and Paolo Traverso. A Request Language for Web-Services Based on Planning and Constraint Satisfaction. In *TES '02: Proceedings of the Third International Workshop on Technologies for E-Services*, pages 76–85. Springer-Verlag, 2002. 21

[6] Reyhan Aydoğan and Hande Zırtıloğlu. A Graph-Based Web Service Composition Technique Using Ontological Information. In *ICWS '07: Proceedings of the 2007 IEEE International Conference on Web Services*, pages 1154–1155. IEEE Computer Society, 2007. 19, 24

[7] Franz Baader, Diego Calvanese, Deborah L. Mcguinness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003. 17

[8] J.C.M. Baeten. A Brief History of Process Algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005. 5

[9] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, 1990. 41, 45

[10] Arindam Banerji, Claudio Bartolini, Dorothea Beringer, Venkatesh Chopella, Kannan Govindarajan, Alan Karp, Harumi Kuno, Mike Lemon, Gregory Pogossiants, Shamik Sharma, and Scott Williams. Web Services Conversation Language (WSCL). `http://www.w3.org/TR/wscl10/`, March 2002. 18

[11] Jerry Banks, John S. Carson, Barry L. Nelson, and David M. Nicol. *Discrete-Event System Simulation (3rd Edition)*. Prentice Hall, 2000. 128

[12] Boualem Benatallah, Marlon Dumas, Quan Z. Sheng, and Anne H. H. Ngu. Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, pages 297–308. IEEE Computer Society, 2002. 10, 22, 56

[13] Boualem Benatallah, Mohand-Said Hacid, Alain Leger, Christophe Rey, and Farouk Toumani. On Automating Web Services Discovery. *The VLDB Journal*, 14(1):84–96, 2005. 3

[14] Boualem Benatallah, Quan Z. Sheng, and Marlon Dumas. The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 7(1):40–48, 2003. 22

[15] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. Automatic Composition of E-services That Export Their Behavior. In *ICSOC '03: Proceedings of the First International Conference on Service-Oriented Computing*, pages 43–58. Springer Berlin / Heidelberg, 2003. 5, 22

[16] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, and Massimo Mecella. Reasoning about Actions for e-Service Composition. In *Proceedings of ICAPS'03 Workshop on Planning for Web Services*, 2003. 16

[17] Daniela Berardi, Giuseppe De Giacomo, Massimo Mecella, and Diego Calvanese. Composing Web Services with Nondeterministic Behavior. In *ICWS '06: Proceedings of the 2006 IEEE International Conference on Web Services*, pages 909–912. IEEE Computer Society, 2006. 16

[18] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. RFC 3986, Uniform Resource Identifier (URI): Generic Syntax. `http://tools.ietf.org/html/rfc3986`, 2005. 12, 13

[19] David Booth. From Web Services to the Semantic Web: Global Data Reuse (Presentation at University of Toronto and University of Waterloo). `http://www.w3.org/2005/Talks/0110-dbooth-semweb/`, April 2005. 11, 12

[20] Ronald Brachman and Hector Levesque. *Knowledge Representation and Reasoning (The Morgan Kaufmann Series in Artificial Intelligence)*. Morgan Kaufmann, 2004. 101, 102

[21] Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987. 1

[22] Tevfik Bultan, Xiang Fu, Richard Hull, and Jianwen Su. Conversation Specification: A New Approach to Design and Analysis of e-Service Composition. In *WWW '03: Proceedings of the 12th International Conference on World Wide Web*, pages 403–410. ACM Press, 2003. 16

[23] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An Approach for QoS-Aware Service Composition Based on Genetic Algorithms. In *GECCO '05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, pages 1069–1075. ACM Press, 2005. 24

[24] Jorge Cardoso, Amit P. Sheth, John A. Miller, Jonathan Arnold, and Krys Kochut. Quality of Service for Workflows and Web Service Processes. *Journal of Web Semantics*, 1(3):281–308, 2004. 24

[25] Fabio Casati and Ming-Chien Shan. Dynamic and Adaptive Composition of e-Services. *Information Systems*, 26:143–163, 2001. 23

[26] Felix Sheng-Ho Chang and Daniel Jackson. Symbolic Model Checking Of Declarative Relational Models. In *ICSE '06: Proceeding of the 28th International Conference on Software Engineering*, pages 312–320. ACM Press, 2006. 151

[27] Moonyoung Chung, Sooyoung Oh, Kyung-Il Kim, HyeonSung Cho, and Hyun-Kyu Cho. Visualizing and Authoring OWL in ezOWL. In *ICACT '05: Proceeding of the 7th International Conference on Advanced Communication Technology*, pages 528–531, 2005. 12

[28] Paul C. Clements. From Subroutines to Subsystems: Component-Based Software Development. In *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute*, pages 3–6. IEEE Computer Society Press, 1996. 1

[29] Alain Colmerauer and Philippe Roussel. The Birth of Prolog. In *HOPL-II: The Second ACM SIGPLAN Conference on History of Programming Languages*, pages 37–52. ACM Press, 1993. 144

[30] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001. 19, 57, 58, 61

[31] Ivica Crnkovic and Magnus Larsson. Component-Based Development - a New Approach in Software Development. Technical report, May 2001. 2

[32] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, Katia Sycara. OWL-S: Semantic Markup for Web Services. `http://www.w3.org/Submission/OWL-S/`, November 2004. 9, 16

[33] Luca de Alfaro, Leandro Dias da Silva, Marco Faella, Axel Legay, Pritam Roy, and Maria Sorea. Sociable Interfaces. In *FroCoS '05: Proceedings of the 5th International Workshop on Frontiers of Combining Systems*, pages 81–105, 2005. 36, 42

[34] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In *ESEC/FSE-9: Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001. 6, 25, 26, 36, 44

[35] Luca de Alfaro and Thomas A. Henzinger. Interface Theories for Component-Based Design. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 148–165. Springer-Verlag, 2001. 49

[36] Yihong Ding, Deryle Lonsdale, David W. Embley, Martin Hepp, and Li Xu. Generating Ontologies via Language Components and Ontology Reuse. In *NLDB '07: Proceedings of the 12th International Conference on Applications of Natural Language to Information Systems*. Springer LNCS, 2007. 95

[37] Xin Dong, Alon Y. Halevy, Jayant Madhavan, Ema Nemes, and Jun Zhang. Simlarity Search for Web Services. In *VLDB '04: Proceedings of the 30th International Conference on Very Large Data Bases*, pages 372–383. Morgan Kaufmann, 2004. 18

[38] Schahram Dustdar and Wolfgang Schreiner. A Survey on Web Services Composition. *International Journal of Web and Grid Services (IJWGS)*, 1(1):1–30, 2005. 18

[39] Islam Elgedawy, Zahir Tari, and Michael Winikoff. Exact Functional Context Matching for Web Services. In *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 143–152. ACM Press, 2004. 3

[40] P. Erdös and A. Rényi. On Random Graphs. I. *Publicationes Mathematicae*, 6:290–297, 1959. 139

[41] Keita Fujii and Tatsuya Suda. Dynamic Service Composition Using Semantic Information. In *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 39–48. ACM Press, 2004. 20

[42] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL—The Planning Domain Definition Language, 1998. 20

[43] Karl D. Gottschalk, Stephen Graham, Heather Kreger, and James Snell. Introduction to Web Services Architecture. *IBM Systems Journal*, 41(2):170–177, 2002. 10, 11

[44] Daniela Grigori, Juan Carlos Corrales, and Mokrane Bouzeghoub. Behavioral Matchmaking for Service Retrieval. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 145–152. IEEE Computer Society, 2006. 17

[45] Rachid Hamadi and Boualem Benatallah. A Petri Net-based Model for Web Service Composition. In *ADC '03: Proceedings of the 14th Australasian Database Conference*, pages 191–200. Australian Computer Society, Inc., 2003. 16

[46] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987. 5

[47] Seyyed Vahid Hashemian and Farhad Mavaddat. A Graph-Based Approach to Web Services Composition. In *SAINT '05: Proceedings of the The 2005 Symposium on Applications and the Internet*, pages 183–189. IEEE Computer Society, 2005. 52

[48] Seyyed Vahid Hashemian and Farhad Mavaddat. A Graph-Based Framework for Composition of Stateless Web Services. In *ECOWS '06: Proceedings of the Fourth IEEE European Conference on Web Services*, pages 75–86. IEEE Computer Society, 2006. 19, 63

[49] Seyyed Vahid Hashemian and Farhad Mavaddat. Composition Algebra: Process Composition Using Algebraic Rules. In *FACS '06: Preliminary Proceedings of The Third International Workshop on Formal Aspects of Component Software*, pages 247–264, 2006. 25, 103

[50] Seyyed Vahid Hashemian and Farhad Mavaddat. Automatic Composition of Stateless Components: A Logical Reasoning Approach. In *FSEN '07: Proceedings of the IPM International Symposium on Fundamentals of Software Engineering*, pages 175–190. Springer Berlin / Heidelberg, 2007. 100

[51] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice/Hall International, 1985. 39, 41, 42, 44

[52] Mat Huizing. Component-Based Development. *Xootic Magazine*, 6(2):5–9, January 1999. 2

[53] Richard Hull and Jianwen Su. Tools for Composite Web Services: A Short Overview. *SIGMOD Record*, 34(2):86–95, 2005. 18

[54] IBM, BEA Systems, Microsoft, SAP AG, Siebel Systems. Business Process Execution Language for Web Services. `http://www.ibm.com/developerworks/library/specification/ws-bpel/`, February 2007. 9, 16, 21

[55] Daniel Jackson. Alloy: A Lightweight Object Modeling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, April 2002. 147

[56] Daniel Jackson. Alloy: A New Technology for Software Modeling. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, page 20. Springer Berlin / Heidelberg, April 2002. 147

[57] Daniel Jackson. Alloy: A Logical Modelling Language. In *ZB '03: Proceedings of the 3rd International Conference of B and Z Users on Formal Specification and Development in Z and B*, page 1. Springer Berlin / Heidelberg, June 2003. 147

[58] Jun-Jang Jeng and Betty H. C. Cheng. Specification Matching for Software Reuse: A Foundation. In *SSR '95: Proceedings of the 1995 Symposium on Software Reusability*, pages 97–105. ACM Press, 1995. 3

[59] Raman Kazhamiakin, Marco Pistore, and Luca Santuari. Analysis of Communication Models in Web Service Compositions. In *WWW '06: Proceedings of the 15th International Conference on World Wide Web*, pages 267–276. ACM Press, 2006. 16

[60] Steven H. Kim. *Knowledge Systems Through Prolog: An Introduction*. Oxford University Press, Inc., 1991. 144

[61] Srividya Kona, Ajay Bansel, and Gopal Gupta. utomatic Composition of Semantic Web Services. In *ICWS '07: Proceedings of the 2007 IEEE International Conference on Web Services*, pages 150–158. IEEE Computer Society, 2007. 21

[62] Jarmo Korhonen, Lasse Pajunen, and Juha Puustjärvi. Automatic Composition of Web Service Workflows Using a Semantic Agent. In *WI '03: Proceedings of the 2003 IEEE/WIC International Conference on Web Intelligence*, pages 566–569. IEEE Computer Society, 2003. 23

[63] A. Kozlenkov, V. Fasoulas, F. Sanchez, G. Spanoudakis, and A. Zisman. A Framework for Architecture-Driven Service Discovery. In *SOSE '06: Proceedings of the 2006 International Workshop on Service-Oriented Software Engineering*, pages 67–73. ACM Press, 2006. 3

[64] Padmanabhan Krishnan and Lai Wang. Supporting Partial Component Matching. In *ICDCIT '04: Proceedings of the First International Conference on Distributed Computing and Internet Technology*, pages 294–303. Springer Berlin / Heidelberg, 2004. 3

[65] Philippe Kruchten. Modeling Component Systems with the Unified Modeling Language. In *Proceedings of the 1st International Workshop in Component-Based Software Engineering*, 1998. 2

[66] Ulrich Küster, Birgitta König-Ries, Mirco Stern, and Michael Klein. DIANE: An Integrated Approach to Automated Service Discovery, Matchmaking and Composition. In *WWW '07: Proceedings of the 16th International Conference on World Wide Web*, pages 1033–1042. ACM Press, 2007. 15

[67] Ulrich Küster, Mirco Stern, and Birgitta König-Ries. A Classification of Issues and Approaches in Automatic Service Composition. In *WESC' 05: Proceedings of the First International Workshop on Engineering Service Compositions*, pages 25–34, 2005. 4, 5, 18

[68] Mikko Laukkanen and Heikki Helin. Composing Workflows of Semantic Web Services. In *WSABE '03: Proceedings of the Workshop on Web-Services and Agent-based Engineering*, 2003. 21

[69] Benchaphon Limthanmaphon and Yanchun Zhang. Web Service Composition with Case-Based Reasoning. In *ADC '03: Proceedings of the 14th Australasian Database Conference*, pages 201–208. Australian Computer Society, Inc., 2003. 22

[70] Xiaocheng Luan, Yun Peng, and Timothy Finin. Quantitative Agent Service Matching. In *WI '04: Proceedings of the 2004 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 334–340. IEEE Computer Society, 2004. 3

[71] Zakaria Maamar, Soraya Kouadri Mostéfaoui, and Hamdi Yahyaoui. A Web Services Composition Approach based on Software Agents and Context. In *SAC' 04: Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1619–1623. ACM Press, 2004. 22

[72] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic Schema Matching with Cupid. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 49–58. Morgan Kaufmann Publishers Inc., 2001. 23

[73] J. McCarthy and P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. pages 26–45, 1987. 16

[74] Kirby McInnis. Component-Based Design and Reuse. `http://www.cbd-hq.com/PDFs/cbdhq_990715km_CBDreuse.pdf`, 1999. 2

[75] Brahim Medjahed, Athman Bouguettaya, and Ahmed K. Elmagarmid. Composing Web services on the Semantic Web. *The VLDB Journal*, 12(4):333–351, 2003. 17

[76] Nikola Milanovic and Miroslaw Malek. Current Solutions for Web Service Composition. *IEEE Internet Computing*, 8(6):51–59, 2004. 18

[77] Robin Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989. 39, 40, 41, 44

[78] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. 17

[79] Srini Narayanan and Sheila A. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *WWW '02: Proceedings of the 11th International Conference on World Wide Web*, pages 77–88. ACM Press, 2002. 16

[80] OASIS. Reference Model for Service Oriented Architecture 1.0. `http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf`, August 2006. 10

[81] Seog-Chan Oh, Byung-Won On, Eric J. Larson, and Dongwon Lee. BF*: Web Services Discovery and Composition as Graph Search Problem. In *EEE '05: Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service*, pages 784–786. IEEE Computer Society, 2005. 20

[82] Claus Pahl. An Ontology for Software Component Matching. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(2):169–178, 2007. 3

[83] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia P. Sycara. Semantic Matching of Web Services Capabilities. In *ISWC '02: Proceedings of the First International Semantic Web Conference on the Semantic Web*, pages 333–347. Springer-Verlag, 2002. 18

[84] Joachim Peer. A PDDL-Based Tool for Automatic Web Service Composition. In *PPSWR' 04: Proceedings of Second International Workshop on Principles and Practice of Semantic Web Reasoning*, pages 149–163, 2004. 20

[85] Carl Adam Petri. Kommunikation mit Automaten (English Translation). *New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377*, 1. 5

[86] Carl Adam Petri. *Kommunikation mit Automaten*. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. 5

[87] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math, 2001. 1, 2

[88] Erhard Rahm and Philip A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal*, 10(4):334–350, 2001. 23

[89] Jinghai Rao, Peep Küngas, and Mihhail Matskin. Application of Linear Logic to Web Service Composition. In *ICWS '03: Proceedings of the 2003 International Conference on Web Services*, pages 3–9. CSREA Press, 2003. 10, 20, 24

[90] Jinghai Rao and Xiaomeng Su. A Survey of Automated Web Service Composition Methods. In *SWSWPC '04: Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition*, pages 43–54. Springer Berlin / Heidelberg, 2004. 18

[91] Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001. 16

[92] Cristina Schmidt and Manish Parashar. A Peer-to-Peer Approach to Web Service Discovery. *World Wide Web*, 7(2):211–229, 2004. 3

[93] Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. The Semantic Web Revisited. *IEEE Intelligent Systems*, 21(3):96–101, 2006. 12

[94] Zhongnan Shen and Jianwen Su. On Completeness of Web Service Compositions. In *ICWS' 07: Proceedings of the 2007 IEEE International Conference on Web Services*, pages 800–807, 2007. 17

[95] Dong-Hoon Shin and Kyong-Ho Lee. An Automated Composition of Information Web Services based on Functional Semantics. In *WSCA '07: Proceedings of the IEEE Workshop on Web Service Composition and Adaptation*, pages 300–307. IEEE Computer Society, 2007. 19, 99

[96] Electra Tamani and Paraskevas Evripidou. A Pragmatic Methodology to Web Service Discovery. In *ICWS '07: Proceedings of the 2007 IEEE International Conference on Web Services*, pages 1168–1171. IEEE Computer Society, 2007. 23

[97] Xianfei Tang, Changjun Jiang, and Zhijun Ding. Automatic Web Service Composition Based on Logical Inference of Horn Clauses in Petri Net Models. In *ICWS '07: Proceedings of the 2007 IEEE International Conference on Web Services*, pages 1162–1163. IEEE Computer Society, 2007. 21

[98] Maurice ter Beek, Antonio Bucchiarone, and Stefania Gnesi. A Survey on Service Composition Approaches: From Industrial Standards to Formal Methods. Technical report, 2006. 18

[99] Paolo Traverso and Marco Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *ISWC '04: Proceedings of the Third International Semantic Web Conference*, pages 380–394. Springer Berlin / Heidelberg, 2004. 22

[100] R. J. van Glabbeek. Notes on the Methodology of CCS and CSP. *Theoretical Computer Science*, 177(2):329–349, 1997. 37, 40

[101] W3C. Namespaces in XML 1.0 (Second Edition). `http://www.w3.org/TR/xml-names/`, August 2006. 11

[102] W3C. OWL Web Ontology Language. `http://www.w3.org/TR/owl-features/`, February 2004. 12

[103] W3C. RDF Primer. `http://www.w3.org/TR/rdf-primer/`, February 2004. 12

[104] W3C. Web Services Architecture. `http://www.w3.org/TR/ws-arch/`, February 2004. 10

[105] W3C. Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. `http://www.w3.org/2002/ws/desc/wsdl20-primer/`, June 2007. 152

[106] W3C. Web Services Description Language (WSDL). `http://www.w3.org/TR/wsdl/`, March 2001. 9

[107] Peter Z. Yeh, Bruce Porter, and Ken Barker. Using Transformations to Improve Semantic Matching. In *K-CAP '03: Proceedings of the 2nd International Conference on Knowledge Capture*, pages 180–189. ACM Press, 2003. 23

[108] Ustun Yildiz and Claude Godart. Information Flow Control with Decentralized Service Compositions. In *ICWS '07: Proceedings of the 2007 IEEE International Conference on Web Services*, pages 9–17. IEEE Computer Society, 2007. 22

[109] Jianjun Yu, Shengmin Guo, Hao Su, Hui Zhang, and Ke Xu. A Kernel-Based Structure Matching for Web Services Search. In *WWW '07: Proceedings of the 16th International Conference on World Wide Web*, pages 1249–1250. ACM Press, 2007. 3

[110] Tao Yu and Kwei-Jay Lin. Service Selection Algorithms for Web Services with End-to-End QoS Constraints. In *CEC '04: Proceedings of the IEEE International Conference on E-Commerce Technology (CEC'04)*, pages 129–136. IEEE Computer Society, 2004. 24

[111] Amy Moormann Zaremski and Jeannette M. Wing. Signature Matching: A Key to Reuse. In *SIGSOFT '93: Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 182–190. ACM Press, 1993. 3

[112] Amy Moormann Zaremski and Jeannette M. Wing. Signature Matching: A Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(2):146–170, 1995. 3

[113] Amy Moormann Zaremski and Jeannette M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(4):333–369, 1997. 3

[114] Liangzhao Zeng, Boualem Benatallah, Marlon Dumas, Jayant Kalagnanam, and Quan Z. Sheng. Quality Driven Web Services Composition. In *WWW '03: Proceedings of the 12th International Conference on World Wide Web*, pages 411–421. ACM Press, 2003. 24

[115] Ruoyan Zhang, Ismailcem Budak Arpinar, and Boanerges Aleman-Meza. Automatic Composition of Semantic Web Services. In *ICWS '03: Proceedings of the 2003 International Conference on Web Services*, pages 38–41, 2003. 19