DOMAIN SPECIFIC SERVICE REPOSITORY DESIGN

GUANGYAO ZHAN

Domain Specific Service Repository Design

by

© Guangyao Zhan

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Master of Science

Department of Computer Science
Memorial University of Newfoundland

April 2011
St. John's, Newfoundland

Abstract

This thesis describes a formal approach to service repository design, where web services are centrally published by service providers and queried by service consumers. Service behaviors are formally specified and behavioral contracts are utilized to find functional substitutions. If no direct match is found, composed specifications can be used to match the query. A detailed description of an example repository and the design process are presented in the thesis.

Acknowledgements

I would like to devote my sincerest gratitude to my supervisor, Dr. Adrian Fiech, for his encouragement, understanding, and wholehearted support that helped me through all the difficult times. I am also very grateful to my co-supervisor, Prof. Wlodek Zuberek, for his continuous help and support.

I wish to thank Matthias Tilsner, Ph.D candidate at Memorial University and research fellow at Mannheim University of Applied Science, for all the constructive discussions, comments, and feedbacks, which helped a lot for shaping up the thesis. In addition, I am deeply indebted to the entire Tilsner family for their kindness of accommodating me during the first couple of months I was in Germany.

I am heartily thankful to Thomas, Christina, and Max Kühnau for their friendliness and making me feel at home for the time I was living in Mannheim. Life would be much more difficult without their hospitality.

I would like to thank Yilin Liu, Chang Wang, Shuyin Wong, Jifei Ou, Shuo Liu, Tao Wu, and many other friends who offered me their generous help through the years.

I thank Memorial University for the financial support, and Mannheim University of Applied Science for offering me the opportunity to visit and live in Germany for a whole year.

This thesis is dedicated to my parents and my girlfriend for their endless love and understanding, without which this work would not have been possible.

Contents

Chapter 1

Introduction

Recent proliferation of service application programming interfaces (APIs) like those offered by Google, Facebook, or Twitter has had a pronounced impact on web application development. By using these APIs, developers can utilize existing infrastructures of other organizations (often worth millions of dollars) to quickly build and maintain applications. No longer do application developers have to incur heavy capital investment to build the necessary infrastructure.

However, there are hidden risks of relying on these proprietary API's. If application developers do not architect their applications in such a way that it is relatively easy and fast to switch to a completely different set of APIs, they are going to face vender lock-in with all the resulting shortcomings. Additionally, no matter how reliable a third party's service might be, there will always be times when a particular service is not available (due to maintenance, attacks, connectivity issues, etc). At these times service consumers

need the ability to switch quickly and automatically to an alternative service provider.

Application developers have various ways to deal with these reliability issues. For example, one way to build more robust applications is for application developers to create adapter layers intermediating those proprietary APIs so that their applications can ignore the vendor differences, thus making on-the-fly switch to alternatives possible.

But what if service providers actually have somehow standardized APIs? Then there is no need to write adapter layers anymore, or at least the number of adapter layers required will be significantly reduced. It would be even better if all those APIs were not scattered all over the web waiting for you to discover them in a lengthy searching process. Imagine that all those APIs are nicely organized in a centralized place and well documented with necessary usage information.

In this thesis we propose such a centralized place (called a service repository), which holds necessary information for service providers and service consumers. When designing the repository we strive to find a common ground between the requirements from service providers and service consumers. Providers will implement their services according to standard specifications and consumers can search for various implementations of a given service specification. In essence, we are creating a market place where the balance of supply and demand will eventually lead to open and stable API's with sufficient mindshare, so that all parties involved can focus on creating and improving their own products instead of fighting over the interfaces

and vendor lock-in. We believe this is a necessary step towards the ultimate reusability of web applications.

To make such service repository more useful, we strive to provide the ability to automatically query the repository and find all services suitable for use according to some user-specified criteria. Specifically, we want to match syntactic (e.g. input and output types) and semantic (behavioral contracts) interfaces of services in the repository. By providing such ability, we can make dynamic switching of services possible, without requiring service consumers to change their business logic.

The syntactic interfaces are easier to match. There are existing technologies such as Web Services Description Language to describe input and output parameters of services. The semantic interfaces, however, are much more challenging. In this thesis, we will adopt an approach which focuses on treating services as relations and matching the corresponding pre- and post-conditions.

1.1 Service Repository

From our perspective, a service repository is a central place for both service providers and service consumers to publish and search service specifications (descriptions of behaviors and interfaces). Concrete services, which implement these specifications, are maintained by service providers on their own servers. The repository assumes no responsibility to host those services. Service specifications in the repository contain information about where to

locate the corresponding service, what the service actually does with respect to its behaviors, how to communicate with the service to provide input and receive output, how much it costs to invoke the service, and other necessary pieces of information.

A service repository is initially set up by a group of domain experts called repository maintainers. Service providers can only publish their services in the repository if such services implement one of the specifications contained in the repository. Additionally, a service provider might propose to the repository maintainers to include a new specification for which an implementation already exists. It is absolutely up to the repository maintainers to decide if such request will be entertained. If approved, the proposed service specification will be included in the repository. The repository is populated as more and more specifications are added into it and links to their implementations provided.

When a service consumer wants to find a service he needs for his application, he comes to the repository and browses the catalog of existing service specifications. He can also specify the requirements as a query. The repository then tries to match the query with the service specifications in it. After identifying suitable matches, the repository returns to the service consumer a list of matching service specifications. Each of the returned service specifications carries an attached list of references to particular implementations of such specification by different vendors. The consumer can decide then which service to use according to cost, availability, or some other criteria.

Sometimes there will be no service specifications directly matching a consumer's query. However, it is likely that two or more services, if properly composed, will do what is expected. Therefore, the repository should be smart enough to figure out those possible compositions. Due to efficiency concerns, the repository might decide to give up the matching process if no result is found up to a given threshold.

1.2 Organization

This thesis is organized in the following way: in Chapter 2 we describe the motivation of building a service repository. In Chapter 3 we present related work and compare it to our approach. Chapter 4 presents an example scenario where a domain-specific service repository is needed and discusses a framework to design such repository. In Chapter 5 we will complete the design of the example repository in detail and discuss the various design decisions, tradeoffs, and compromises being made. In Chapter 6 we briefly introduce the logic foundation to formalize service repositories. Chapter 7 contains the formalization of all pieces of a service repository and a language based on predicate logic to describe such a system. This is followed in Chapter 8 by a summary of the thesis as well as discussion of future directions to extend our work.

Chapter 2

Motivation

2.1 Status Quo

It is a wild world out there today for service consumers. If an application assembler wants to use any third-party services to provide needed functionality (which he might not have the necessary resources to implement himself), he will have to perform excessive searches for potential providers. If he is lucky, there might be a few providers who offer the services he needs (although some necessary modifications are likely). Then he has to contact the providers to inquire about the price, availability, service level, and other things before he can actually use them. After reaching an agreement, he has to read the detailed documentation of the provider's interfaces and find the right combinations to call. It is almost certain that if he wants to switch to a different provider later (due to poor performance, high price, or other reasons), he will have to repeat the whole process all over again, since the

probability that two providers share similar interfaces is rather slim.

The situation is not any better for service providers, either. There is no standard to follow, which means a service provider has to decide a lot of things on her own. After investing money and resources developing her services and spending a lot on advertising her products on various channels, there might be a rather insignificant demand for her creation. Application developers tend to be reluctant to utilize services because they fear that they will get locked in the provider's system.

This is just a reflection of the current software as a service industry: pretty much like any pre-standardization industry—things have to be custom built. Wheels are constantly being re-invented over and over again.

2.2 The Way Out

Wouldn't it be nice if there exist some standards for both service providers and consumers to follow, and everyone knows where to go for when they need to provide or consume services? Building customized computers seems a lost art nowadays, but it is a good example to illustrate the idea.

2.2.1 Centralized Stores

Before someone builds a computer, he must have some general ideas about what different parts he needs, and he sets a budget to spend. He checks out several big vendors, either online retailers like Newegg and Amazon, or those with physical stores like BestBuy and Future Shop. He rarely goes

to individual manufacturers and orders there, because these big vendors give him many more choices. In fact, major manufacturers accomplish such a large portion of their sales in these big stores that they will assure availability of new products in these big stores as soon as possible. Smaller manufacturers also try their best to deliver their products to these stores. Otherwise the market exposure to their products will be very limited. So in the end most, if not all, suppliers and customers are doing transactions in centralized places.

2.2.2 Standardized Products

When someone chooses a computer component from various alternatives, he can be pretty sure that he can safely change from one manufacturer to another as long as he sticks to the same specification. This is made possible because for each component, there is a set of specifications that every manufacturer has to satisfy in order for their products to play well with other components in the whole system.

As a consumer, one only needs to know what specification to look for, and does not need to worry if one component works well with another, even without real testing. For example, if one needs to buy a hard drive for his computer to be built, he only needs to remember that it must be 3.5" and support SATA II interface. Similarly for things like monitors and graphic cards, a customer only needs to make sure that both support DVI connector, and that the graphic card has enough power to drive the monitor's maximum resolution.

Standard specifications eliminate possible confusion and incompatibility, which are major issues for multiple players in the field to cooperate with each other to produce useful final products.

2.2.3 Assisted Discovery

Sometimes one does not know what specific products exist for her needs, since all she has are really some specifications for potential products. Many of the online stores offer sophisticated systems to search for products that satisfy certain specifications.

For example on Newegg one can choose monitors with specified parameters such as 1920×1080 resolution, a DVI port, an HDMI port, and a screen size of 24". The system will return a list of monitors that have all features that are asked for. There are many more options to specify should one need to nail down to fewer choices.

The ability to quickly discover intended products and filter out unfit ones eliminates the need of manually reviewing each product, lowering the total time to arrive at final selection.

2.3 Service Repository

What do we learn from the example of building a computer? We can identify three key factors:

Centralized stores

Everyone is on the same platform. There is no confusion of where to

sell and buy products.

Standardized products

Due to the existence of standard specifications of products followed by all suppliers, consumers can stick to these specifications and be confident that the products work as expected.

Assisted discovery

There are too many products to choose from. Filtering out unnecessary products speeds up matching of supply and demand, saving time to manually check each product.

Can we somehow apply the same ideas to the domain of software services so that shopping for services is as easy as shopping for products? To begin with, there must exist some kind of centralized places that service providers and consumers both go to when they need to publish or consume services. We call such places *service repositories*.

A service repository is a centralized storage of service specifications. It is not necessary for a service repository to host the actual services in the repository per se, but it is vital that the repository contains the necessary pieces of information to properly describe the functionalities of services, and to locate and invoke those services.

2.3.1 Domain Specific Repositories

There is no silver bullet. We cannot possibly imagine a megashop that fits the needs of everyone, although it might be the case that a supermarket is all

one wants when he just needs to buy some groceries.

The needs of software service consumers vary from one specific domain to another. It is highly unlikely that a single service repository can hold all specifications of everything. Thus, a service repository is designed to be domain-specific. There might be a repository for weather information, another for traffic routing, etc. When a service provider wants to publish some services, she will locate repositories in her target domain. This will limit the choice of channels, but it is necessary to avoid overloading.

2.3.2 Curated Platform

In order to maintain a common set of specifications, a service repository will not blindly accept anything service providers want to add. Otherwise a service provider adds a particular specification for a service, another provider might add a different specification which is essentially the same. Very quickly the repository will be filled up with incompatible specifications that are difficult to use.

We believe a better approach is to have domain experts to set some standards on what to add into the repository and what to exclude. These experts will be responsible to listen to the opinions of various service providers and consumers to decide what service specifications are needed or expected in the repository. A common set of vocabularies will be developed to describe all services in the domain, so that both providers and consumers can properly communicate with each other.

To a large extend, the success of a service repository depends on its

maintainers' ability to analyze the domain and to come up with a good design that satisfies the needs of both service providers and consumers. The maintainers will also supervise and guide the continuous development and refinement of the repository to cope with future changes.

2.3.3 Smart Search

With a proper language to describe the functionalities being devised by domain experts and utilized to specify services, we believe it is necessary for the repository to be able to search for specifications in ways beyond keywords and textual descriptions.

One important aspect of service specification is semantics, that is, what a service does, what it expects from its input, and what it guarantees for its output. By taking semantics into consideration, it is possible to determine many more things. For example, it would be possible to decide if a service can be used as a functionally identical substitution of another service. It would also be possible to decide if two services can be chained together, so that the output of a service is fed into another service to produce desired results. Furthermore, it would be possible to decide if two services, if chained together in the proper way, can be used as a replacement of another service.

With these capabilities, a service repository would allow consumers to find services they need and be confident that the result matches functionally. Consumers thus can dynamically query the repository and automatically switch to alternative services without human intervention should the one they originally use fail. We believe this is the future of services.

Chapter 3

Related Work

The idea of formally describing services to allow automated service discovery and composition is not brand new. Agarwal et al. [1] present a method of using π-calculus and description logic to describe and compose software components. The authors establish a SQL-like language that allows users to semantically describe services and query for them. This generates an elaborate, yet complicated approach for semantic queries. Syntactic information, however, is ignored. While the described method does allow the semantic annotation of syntactic types, true interface matching that would enable direct integration into common technologies is not possible. We explicitly strive for a solution that builds on top of syntactic matching principles.

Liskov and Wing [17] provide an effective base for such a solution. In their work they present a method for describing software components. Using an axiomatic approach, they employ predicates to formulate pre- and post-conditions that specify the behavior of methods and explain how their

definition can be used for subtyping relations. Zaremski and Wing [23] extend on that definition and discuss different alterations. We aim at reusing their definition of subtyping for formulating query matches.

A different option to use for specifying services is provided by Broy et al. [6]. In their work, services are understood as functions on input and output streams in order to facilitate time dependent calculations. This, however, requires the semantic specification using mathematical functions. Since we aim at allowing consumers to query for services, this would require these consumers to specify the desired functionality in the same way.

In a similar manner, Arbab [2] uses time relations in order to describe communication patterns between services. Just like Broy, he models channels between components and the temporal behaviors on these channels. While this temporal argumentation does have its benefits, it is not required in our scenario, since classical services such as web services operate sequentially. Even more, we believe that it hinders functional description of independent components.

Numerous other approaches such as the work done by Elgedawy [9], Li and Horrocks [16], and Pilioura and Tsalgatidou [20] use domain ontologies to describe the semantics of functions. In these ontologies, services are tagged with keywords that describe their semantic meaning. Afterwards, these keywords are connected with one another to allow interpretation of relations between services.

Friesen and Börger [10] use an approach similar to service ontologies where they define a set of goals that allows semantic description of services.

In order to allow service notation and discovery, these goals need to be identified beforehand and stored at a central goal repository. While these approaches provide a very powerful and flexible way for querying and learning, they do not support automated composition of services.

One way to achieve both specification and composition of services is presented by Bailly et al. [3]. Their main achievement is a formal system in which the results of compositions (also referred to as composite builts) again are services, thus they can be treated like any atomic services. However, their solution does not discuss substitutability, which we require in order to achieve successful matchmaking.

The issue of component (and thus service) substitutability is discussed in length by Belguidoum and Dagnat [5]. While they argue extensively about requirements, dependencies, and context sensitivity when substituting, they do not draw a direct connection to semantic and syntactic matchmaking. While their work has given us an informative insight into the topic in general, it is not sufficient enough for our scenario.

An interesting approach is discussed by Lécué and Léger [15] where they try to discover possible compositions of a finite set of services that fulfills a given query. To achieve this, the semantic similarity between input and output data types is calculated. Upon retrieving the query, the distance between the different input and output components of the service in relation to those of the query describes the level of a semantic match. An algorithm that matches the input and output parameters and returns possible matches for a given query is provided, just like a method of achieving service com-

position. Unfortunately, all semantics of a service is solely modeled based on their parameters, rather than taking the behavior into consideration. As a result, this solution does not allow for effective semantic matching, but simply extends syntactic matching.

Helm et al [11] model the semantics of components using contracts that include pre-conditions for methods and a series of state changes that are achieved by invoking them. While this series of state changes does allow service composition quite easily, it requires an extensive understanding of the domain and the internals of a service not only when describing, but also when querying for services. Since we desire a black-box-like nature of services that abstracts away from the internal behavior of components, their solution is not applicable for our scenario.

In their research, Chan and Lyu [7] concentrate purely on Web services, proposing a way of composing web services using their native Web Service Description Language (WSDL) documents and additional interaction information. While WSDL documents model the syntactic interface of services, interaction diagrams are not sufficient to describe semantic behaviors of services. Furthermore, we require a solution that works for generic services, independent of actual implementations rather than relying on specific ones.

Another approach for achieving specification composition is provided by Hemer [12]. He presents a way to describe services with pre- and post-conditions, very similar to Liskov and Wing's approach mentioned above. Based on that, he specifies different combinators that allow users to create combined service specifications. These aim at describing the way the re-

sulting service combination behaves. Unfortunately, this approach requires users to specify the correct assembly using these combinators for each service composition by hand, rather than supporting an automated composition.

A more formal and complex solution is presented by Hoffmann et al. [13]. Using a heuristic approach they propose an algorithm to find appropriate service compositions for a given user requirement, called a *task*. Within this algorithm, a solution is found by describing the pre-condition combined with a set of task-specific constants as an initial state, and then iteratively adding subsequent services to modify that state. Any combination of services that yields a target state as described through the effect-component of the task becomes a valid candidate. The heuristic function, based on AI Planning techniques, optimizes the searching and provides filtering and ordering of results. While this approach does allow a relatively easy identification of compositions that match a given query, it does not support a formal description of such compositions. In contrast, we aim at making composition results reusable as normal services.

Chapter 4

A Sample Repository

In this chapter we will look into the steps to design a service repository. We will introduce an example of flight search applications to illustrate the steps to analyze the problem domain and figure out necessary pieces to complete a service repository.

4.1 Understanding the World

Suppose we are going to build a repository for services related to flight searches, much like similar services offered by airlines and travel agents on their websites. Before we can do anything meaningful, we have to assume some basic understanding of the world we are modeling.

In our example, the world consists of a few countries and each country has a number of cities with one or more airports. For simplicity we assume each airport has only one parent city. There are a few airlines operating flights scheduled on a daily basis among those airports, though no single

airline covers all flights between any pair of airports. In addition, even though an airport is always connected by some flights to another airport, there is no guarantee that direct flights are available between any pair of airports.

All information about our world is available in some databases. For example, each airline has a flight database containing details of all flights operated by this particular airline. A geographic database contains information such as the locations of all countries, cities, airports, and distances between two cities and airports.

There are several stakeholders in this imaginary world. Airlines operate flights and they want people to search for these flights. There are also dedicated companies that provide services to search for flights across multiple airlines. Together they are what we call *service providers*, because in general they have some data that is needed by others and they are interested in becoming suppliers of our repository to allow access to the data for a fee. On the other hand, we have *service consumers* such as travel agencies who need to purchase and consume these services to build final applications for their end users (travelers in this case) to search for flights and plan their trips.

Our job as service repository maintainers is to make the right design decisions, so that the majority of service providers can publish their services in the repository and that service consumers can find matches to most of their reasonable requests.

Depending on how we want to shape the service repository, we have to make decisions about what to include in the repository and what to leave

out. For example we can aim at providing a giant repository to cover every possible service related to flight searching (and possibly some related services such as hotel reservation). Alternatively we can also choose to build a rather small but carefully selected repository that covers only services offered by North American airlines as long as our service consumers are happy with the decision. There is no right or wrong about what to build, but we need to be clear about what we are trying to achieve.

4.2 Determining Data Types

Next we need a language to describe basic concepts in the world. These come in the form of data types, entities with specialized structure that can hold necessary information for concepts we want to express. Throughout this thesis, a data type will be typeset in small caps to distinguish it from normal text.

For example, it is obvious that we need types such as FLIGHT, AIRPORT, TIME in the domain of flight search. We will probably include CITY as well, otherwise it is not feasible to describe services that can search for flights between two cities. This is a design decision, though: many travel agencies are perfectly fine to accept services that can only search for flights between two airports, and for most cities there is only one airport anyway. Travelers usually want to be specific which airport they are going to use should there be multiple choices available.

For each of the data types, we will decide what information it contains.

A FLIGHT will probably contain the flight number (usually specified by an airline code followed by some digits) as its identifier, the departure and destination airports, departure and arrival time, flight duration, etc. The decision of what information to be made available in the data types is completely up to the domain experts.

It should be noted, however, that any decisions we make have consequences. If we make too little information available, it might be difficult or impossible to describe certain things. For example, we might want to omit the departure and arrival cities in FLIGHT since we can consult some database to lookup which cities the departure and arrival airports belong. By doing so we enforce the constraint that services which intend to assert on the departure and arrival cities property must be specified in a way that involves translating airports to cities, which might or might not be a good thing. On the contrary if we make too much information available, things might become too complicated to manage as everyone is forced to check and verify the integrity of input and output to spot potential mistakes. A good design will need to strike a reasonable balance between the two extremes.

Listed below are some of the data types we choose to include in our repository, along with the attributes that belong to them:

1. CITY: name of the city, list of airports in the city
2. AIRPORT: name of the airport, IATA airport code, parent city
3. FLIGHT: flight number, origin, destination, departure time
4. AIRLINE: name of the airline

4.3 Choosing Predicates

The next step of the repository design is to select proper predicates to express the relationship among data types. Predicates describe boolean functions over data types in our repository. We form pre- and post-conditions by combining predicates with logic connectors to assert certain properties of input and output values, which are crucial to describe the services that we want to include in the repository. Each predicate is represented by its name in our repository that we call a predicate symbol. We hope the name and some text description will give an intuitive idea what properties a particular predicate asserts, but in general the exact meaning of a predicate is backed by a concrete reference implementation to avoid ambiguity.

In our example repository of flight search, we are likely to include a predicate airportInCity(AIRPORT, CITY) that asserts that an airport belongs to a particular city in our world. A reference implementation of the predicate will look up some geographic database and check relevant records to see if the relationship actually holds. When we want to describe services that involve an airport and its parent city, we could use this predicate to specify such constraint. Another predicate flightFromAirport(FLIGHT, AIRPORT) might assert that a flight departs from a particular airport. A reference implementation of the predicate could first check which airline is operating this flight, and look up the airline's database to see its departing airport. Keep in mind, though, that reference implementations of predicates are merely ways to clarify the exact meaning of predicates, and we are not concerned about their technical detail when using predicates to describe and reason

about services.

Ideally we want to keep a small set of predicates available in the repository that is powerful enough to describe all services we could reasonably expect to be included in the repository. The reasoning is twofold: first, anyone who masters the small set of predicates should be able to specify all services he wants, and he is also more likely to understand a specification written by others using the same set of predicates that he is familiar with; second, the repository can mechanically reason about predicates more efficiently if the total number of predicates is kept small.

Therefore we believe the success and usefulness of a repository will largely depend on our selection of predicates. If we select too few predicates, certain services that are reasonable to expect will be impossible to specify and thus be excluded from the repository. On the other hand if we select too many predicates, it will be a mental burden for everyone to understand the meaning of service specifications and thus reduce the usefulness of the repository. A set of predicates carefully chosen by skillful domain experts is vital to a repository.

Listed below are some predicates and their meanings that we will use in our sample repository. Notice that we include two similar predicates, flightFromAirport(FLIGHT, AIRPORT) and flightFromCity(FLIGHT, CITY), because we expect that some providers in our flight search repository will likely provide services that deal only with flights departing from airports, while other providers and travel agencies will also need to describe services that deal with flights departing from cities in general. We do see similar

situation in the real world. If we leave any of the two predicates out, it would be difficult, if not impossible, to properly specify services that deal only with airports or cities, depending on which predicate is omitted in the repository.

validCity(CITY**)** : true if the city is indeed a valid city existing in some database; false otherwise.

validAirport(AIRPORT**)** : true if the airport is an operational airport existing in some database; false otherwise.

cityWithAirport(CITY**)** : true if there are one or more airports in the city; false otherwise.

airportInCity(AIRPORT**,** CITY**)** : true if the airport belongs to the city; false otherwise.

flightFromAirport(FLIGHT**,** AIRPORT**)** : true if the flight departs from the airport; false otherwise.

flightFromCity(FLIGHT**,** CITY**)** : true if the flight departs from an airport belonging to the city; false otherwise.

4.4 Adding Specifications

With all the building blocks being designed, now it is time to add some service specifications into our repository. For each service, we are interested in its syntactic as well as semantic interface. The syntactic interface is described

by the input and output type, while the semantic interface is described by a pair of pre-condition and post-condition.

Consider a specification getFlightFromCity for a service that takes an input city and returns a flight departing from the input city. We could specify the service using four parameters: its input type, output type, pre-condition, and post-condition. The definition of the service is given in specification 4.1.

Specification 4.1 getFlightFromCity

CITY $\rightarrow$ FLIGHT

ϕ: cityWithAirport(in)

ψ: flightFromCity(out, in)

Note that the "ϕ" symbol means pre-condition, which is the condition under which the service can be legally invoked. The "ψ" symbol means post-condition, which is the guarantee that the service promises for its result. Two special names *input* and *output* are used to denote the input parameter and output parameter of the actual service.

Now consider two additional service specifications. Service specification getAirportOfCity is defined in specification 4.2, which says that it takes a city and returns a valid, functioning airport that belongs to the input city.

Service specification getFlightFromAirport is defined in specification 4.3, which says it takes a valid, functioning airport and returns a flight departing

Specification 4.2 getAirportOfCity

CITY $\rightarrow$ AIRPORT

ϕ: cityWithAirport(in)

ψ: validAirport(out) $\wedge$ airportInCity(out, in)

from the input airport.

Specification 4.3 getFlightFromAirport

AIRPORT $\rightarrow$ FLIGHT

ϕ: validAirport(in)

ψ: flightFromAirport(out, in)

In our sample repository for the domain of flight searches, we will also need services that can take a pair of origin and destination airports, and return a flight that departs from the origin and reaches the destination as specified in 4.4.

Notice that the input type (AIRPORT, AIRPORT) is a tuple, which in this case is a pair of origin and destination airports. In the pre- and post-conditions we access the individual elements of the tuple by using their zero-based index.

The reader might wonder what happens if there is no direct flight from the origin to the destination? Should we return a connecting flight as a result in this case? This is largely a design decision for the repository designers to

Specification 4.4 searchFlight

(AIRPORT, AIRPORT) $\rightarrow$ FLIGHT

ϕ: validAirport(input(0)) $\wedge$ validAirport(input(1))

ψ: flightFromAirport(output, input(0)) $\wedge$ flightToAirport(output, input(1))

make. It is reasonable to expect the distinction between services returning direct flights only and services that also return connection flights in a real-world use case, and it is the repository designers' job to ensure necessary data types and predicates are in place to facilitate the specifications of such services. For the purpose of demonstration, we choose not to care about such details for now, and as a consequence it is not important of what kind of flights we get. A more detailed design will be presented in subsequent chapters.

The key thing to remember is that a repository does not take arbitrary specifications and put them all in. The catalog of specifications is restricted by the design decisions made by repository designers, and everyone using the repository is subject to such constraints. This more or less corresponds to the real world stores: BestBuy won't stock all electronic products available in the world, but only a selection that the management of BestBuy thinks is a suitable fit due to factors like customer demand, profitability, physical constraint of warehouse spacing, etc. In the software repository world, we will not take all specifications written by everyone, but only a subset that

the repository maintainers think will best match the needs of both service providers and service consumers. At the same time the selection must permit efficient reasoning about these specifications to respond to user queries.

4.5 Choosing Axioms

We specify services by constructing pre- and post-conditions using predicates, and we reason about them by checking the logic relationships of these predicates.

It is trivial to see that we can invoke a service that implements specification 4.2 and supply it with a city to get an airport that serves that city. Next we could invoke a service that implements specification 4.3 with the provided airport as an input. We will get a flight that departs from the airport in the original city. Of course, the reader can infer as a logical consequence that the flight must also depart from the city. This is actually a proper description of a service that implements specification 4.1.

We would like the repository to be aware of this. It requires the repository to know that if a flight departs from an airport and the airport belongs to a city, it follows that the flight must also depart from the city. We call such a statement an axiom in our repository. The said axiom looks like this:

$$\begin{aligned} \text{flightFromAirport(Flight, Airport)} \wedge \text{airportInCity(Airport, City)} \\ \Rightarrow \text{flightFromCity(Flight, City)} \end{aligned} \tag{4.1}$$

Such axioms are crucial in our repository. Without them, the repository will not be able to infer that services that satisfy specifications 4.3 and 4.2 can be combined to provide similar functionality as services that satisfy specification 4.1.

Just like the decision about data types and predicates, the decision of what axioms to include in the repository will significantly influence the ability and usefulness of a repository. In practice, the repository maintainers will have to be extremely careful to choose proper set of axioms due to a couple of reasons: it is easy to carelessly include conflicting axioms that will lead to logical errors; too many axioms will greatly increase the time needed to reach certain conclusions, thus making the repository slow to respond.

4.6 Using the Repository

Now that we have designed a simple repository, we can try to use it from a service consumer's perspective.

Suppose we come to the repository and want to find out if there is any service that takes a city and returns a flight leaving from there. Since the repository is quite small, we can just browse each specification in the repository and check if it does what we want. We simply compare the input and output signature of each specification, and then read its pre-condition and post-condition to make sure its behavior is desired.

Alternatively, we can automatically search the repository (e.g. if it contains too many specifications for manual browsing). In this case, we need

to write a search query to express our intention. We formulate the query by giving four parameters: input type, output type, pre-condition, and post-condition. The query to search for service specifications that can take an input city and return a flight departing from the input city is given in query 4.1.

Query 4.1 Sample query getFlightFromCity

CITY $\rightarrow$ FLIGHT

ϕ: validCity(in) $\wedge$ cityWithAirport(in)

ψ: flightFromCity(out, in)

Careful readers might notice that a query is very similar to a specification. In fact, for all practical purposes, we will treat queries as (somewhat restricted) special forms of specifications. The restriction is that queries cannot have any quantifiers in pre- and post-conditions. We will explain why in later chapters.

Now we try to match the query against service specifications in the repository. At the first glance, it is pretty obvious that the service specification 4.1 matches the query directly, since they are basically the same. What is not so obvious though, is the fact that specification 4.2, if composed with specification 4.3, can satisfy the query too. If a flight is leaving from an airport and the airport belongs to a city, it is trivial to conclude that the flight is also leaving from the city. This substitution is illustrated in figure 4.1. The

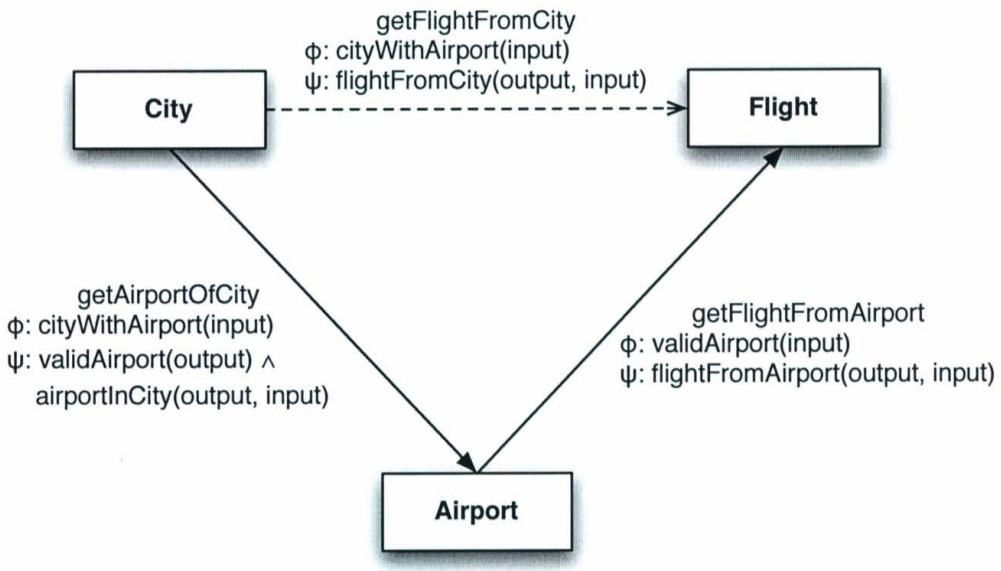

Figure 4.1: Substitution of services

reasoning is possible with the help of axiom 4.1.

Chapter 5

Design

In chapter 4 we demonstrated the framework to design a very simple service repository for flight search. In this chapter we will explore the repository in detail and design additional data types, predicates, service specifications, and axioms to make the repository more complete.

5.1 Roles

There are four major groups of stakeholders in a service repository:

Repository maintainers are people who design and maintain the repository. Throughout this chapter we will mainly look at various design decisions from the repository maintainers' perspective.

Service providers are people who implement and operate various concrete services according to specifications listed in the service repository.

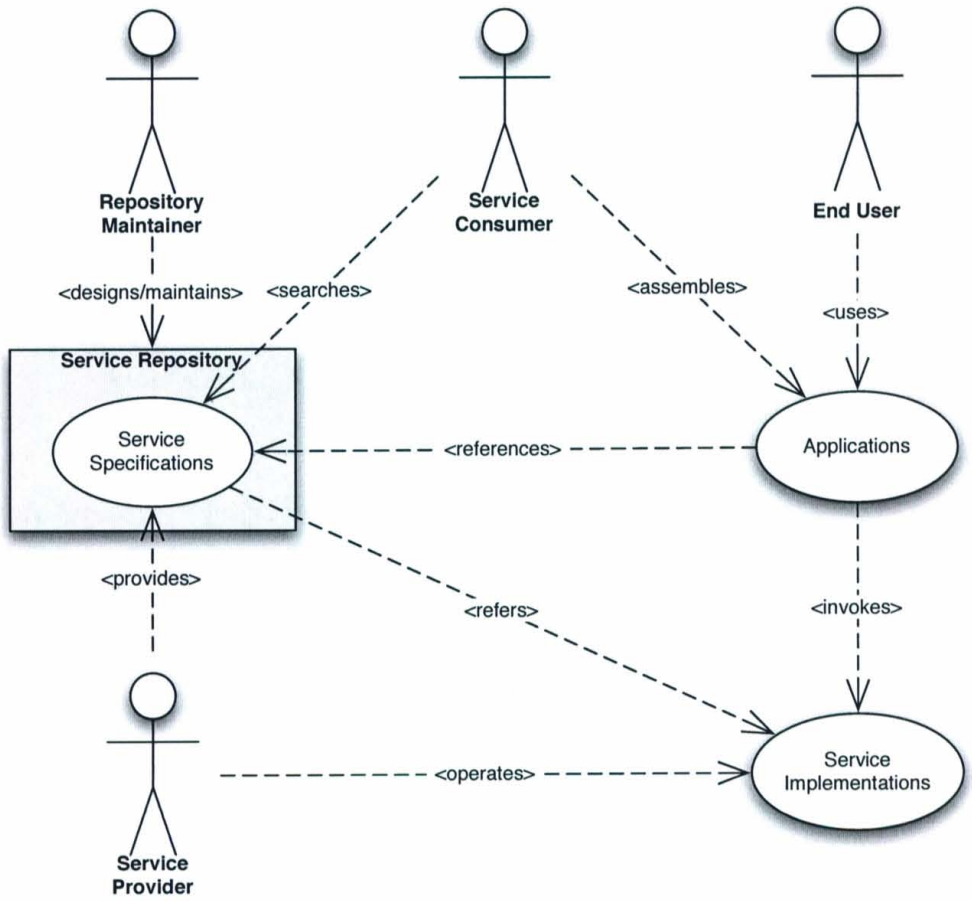

Figure 5.1: Four roles of a service repository

They hope that their services will be bought and utilized by service consumers.

Service consumers are people who have specific requirements for certain services and are willing to purchase them. They assemble from these services applications aimed at the end user.

End users are people who will ultimately use the applications built by service consumers upon services found through a service repository. End users will not directly interact with the service repository, and from their perspective the service repository and service providers do not even exist.

In the example domain of flight searches, the thesis author will play the role of repository maintainer. He will walk the reader through the process of designing and maintaining a service repository. Airlines and various other companies will be service providers who implement and advertise services that can search for available flights between airports. Travel agents are the primary service consumers in our example. They purchase and consume flight searching services to build applications that end users (travelers in this case) use to find and select routes that best fit their schedule.

5.2 Requirement Analysis

In this section, we will analyze the needs of travelers to see what kind of functionalities are required in the flight planning applications that will be

built by travel agents.

The most fundamental need of travelers is that when they want to fly from an origin to a destination at a given time, they need to know what flights are available. Travelers will then pick one of the flights based on various factors such as departure time, flight duration, arrival time, airlines, airfares, meals and services offered onboard, etc. For demonstration purposes and to keep things simple, we will be concerned only with a few of these factors such as airlines, locations and time. We will leave out other factors such as airfares, meals, etc.

A typical flight search usually results in a quite large set of possible flights. Many travelers will require more advanced functionalities to search for flights to narrow down the scope. For example, some travelers are more sensitive to time, and they will specify an exact time range that they would like to depart from or arrive at a particular airport. Other travelers might be less sensitive to time but more sensitive to price. They tend to choose a larger range of time in the hope that cheaper flights will be found and returned as a result. In reality many travel agents offer an option usually labeled "I'm flexible with time" to address this particular need.

In addition, many airlines operate frequent flyers programs to encourage customer loyalty by rewarding travelers if they fly more often with one airline. As a result many travelers tend to favor some airlines to accumulate their miles. Some airlines form alliances to acknowledge participating members' frequent flyers programs, therefore it is also necessary for travelers to search for flights with their preferred airline alliances.

Even the simple concept of origin and destination differs significantly for many travelers. Many big cities have multiple airports with different connections to other parts of the world. Some travelers might prefer a particular airport due to reasons like convenience of local transportation from and to that airport. Others might not care about this difference because they can drive to any airports in a city and they are willing to take any flights that depart from a city. Same applies to the destination when some people would prefer certain airports to land while others are indifferent.

Many travelers need to fly to some place, stay there for a while, and then continue to fly to other places. These so-called "multi-destination flights" exist because airlines usually offer discounts if travelers fly longer distances with them. Plus, should travelers miss any intermediate flights, airlines are more inclined to figure out a backup plan if they know the travelers are going to fly with them many more times down the road. It is also more convenient for travelers to specify their intended routes all at once instead of breaking the whole trip into small segments and search for each of the segment individually. As a result, travelers tend to prefer multi-destination flights to enjoy more convenience, better deals and services.

Clearly there are large variances of functionalities required. As we strive to build a rather complete service repository for the domain, we would like to accommodate the majority of services that are necessary to build applications that can support these functionalities. On the other hand, we are not committed to cover every single aspect of all possibilities. We are quite happy to leave out services that are rarely used or requested. In such

cases service consumers will need to produce their own customized code to implement functionalities that are not present in the repository. The control of the repository remains in the maintainers' hand.

5.3 Domain Model

Based on the requirements analyzed in the previous section, we can identify many of the services that will be needed by service consumers to build final applications. Before we start to analyze what services to include, a common set of vocabularies needs to be agreed upon to describe various concepts in our problem domain.

In our example domain, we will primarily deal with the following entities:

Flights are of primary interest in the problem domain. A flight is a segment flown between two airports operated by an *airline*. It has a flight number, origin, destination and departure/arrival times. Flights can be either non-stop or direct.

Itineraries are ordered collections of flights. Later on we will define (with the help of predicates) valid itineraries, which have certain restrictions with respect to the timing of the flights in the itinerary. Itineraries can represent non-stop, direct, connecting, round-trip or multi-destination flights.

Airlines are carriers of commercial flights. Some airlines collaborate with other airlines to form *airline alliances* to provide better services for

customers of affiliating airlines. For simplicity, we assume each airline can be a member in at most one airline alliance.

Airline alliances are groups of airlines that have partnership with each other. Many airline alliances have shared frequent flyer programs among participating airlines for customers to consolidate their mile credits. For this reason, many travelers prefer to fly with airlines of the same airline alliance.

Airports are places where aircrafts take off and land. They are considered physical points in our domain that are connected by flights. Airports are also connected to nearby cities by local transportation. We assume each airport has exactly one parent city.

Cities are places where travelers reside or want to have access to. For most cities there is usually only one airport that is considered accessible due to reasons such as local transportation constraints or affiliation relationships. However, some cities might have multiple airports nearby and thus travelers are free to choose which airport to use.

Time is an instant in a continuous flow. Flights are assumed to take off and land at specific points in time.

Time interval is a range of time between two instants. When travelers search for potential flights, they usually specify vague terms like *tomorrow*, *next morning*, or *on Tuesday*. These terms need to be translated into time intervals to make sense. For example, *tomorrow* is really an

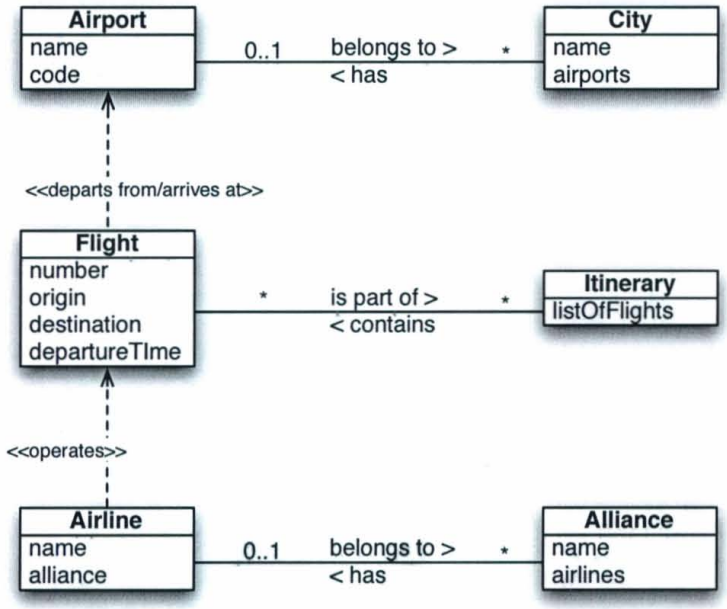

Figure 5.2: Model of the sample domain of flight search

interval between midnight today and the instant 24 hours after that. Sometimes travelers will search for flights departing at, say, 10 A.M tomorrow. In these cases it appears that they are specifying a time point instead of an interval. In reality, though, it is highly unlikely these travelers will consider only flights departing exactly at 10 A.M. the next day. More often the expected behavior is to search for flights departing during a time interval that is around 10 A.M. the next day, plus or minus a threshold (for example half an hour).

5.4 Data Types

To formally capture the essence of the domain objects mentioned in the previous section, we design the necessary data types to hold information about these objects. We will not be concerned about exact representation of the data types required since existing technologies such as WSDL [8] already handle this aspect pretty well, but we do need to have names and concrete references. Specifically, we will have the following data types in our repository:

FLIGHT contains information such as flight number, carrier, departure time, flight duration, etc.

ITINERARY contains an ordered list of flights that constitutes a single trip. It is the result of flight searches.

AIRLINE contains information such as the name of the airline, the alliances it participates in, etc.

ALLIANCE contains information such as the name of the alliance, member airlines.

AIRPORT contains information such as the name, 3-letter IATA airport code, parent city, etc.

CITY contains information such as the name of the city, its affiliating airports, etc.

TIME contains information to pin down an instant of time.

INTERVAL contains a starting point and an end point of time to fully specify a time interval.

5.5 Signatures of Required Services

Now we turn the attention to the question: what services are likely to be requested by service consumers in order to build applications?

5.5.1 Search for One-Way Itineraries

The most straightforward service in demand is one that simply searches for one-way itineraries given an origin airport, a destination airport, and a time interval of departure time, as in signature 5.1.

Signature 5.1 searchOnewayItineraries

(AIRPORT, AIRPORT, INTERVAL) → [ITINERARY]

Searches for one-way itineraries between two airports.

The first element of the input is the origin airport, the second element the destination airport. The third element specifies the interval of time during which the traveler intends to depart.

Output is a list of candidate itineraries.

Note that the input and output of services are considered single values. When there are multiple parameters, they need to be packed into tuples.

In signature 5.1 the input type is a tuple with three elements, with the first AIRPORT representing the origin airport, the second AIRPORT representing the destination airport, and the third INTERVAL representing the time interval of intended departure time. Collections of the same type are represented by a parameterized list in a pair of square brackets, as is shown by the output type, which is a list of itineraries.

We also need services that search for only direct or non-stop flights as many travelers do not want intermediate stopovers if possible or at least minimal transfers. Thus we have two additional signatures to address the need.

Signature 5.2 searchDirectItineraries

(AIRPORT, AIRPORT, INTERVAL) $\rightarrow$ [ITINERARY]

Searches for direct itineraries between two airports.

The first element of the input is the origin airport, the second element the destination airport. The third element specifies the interval of intended departure time.

Output is a list of candidate direct itineraries. Each of the returned itineraries will consist of exactly one direct flight.

Signature 5.3 searchNonstopItineraries

(AIRPORT, AIRPORT, INTERVAL) → [ITINERARY]

Searches for non-stop itineraries between two airports.

The first element of the input is the origin airport, the second element the destination airport. The third element specifies the interval of intended departure time.

Output is a list of candidate non-stop itineraries. Each of the returned itineraries will consist of exactly one non-stop flight.

5.5.2 Search for Multi-Destination Itineraries

In theory, service consumers could use the service that searches for one-way itineraries to build their own version of multi-destination itineraries search functionality by continuously searching for flights given a list of origins and destinations.

In reality, however, a single service that can directly search for multi-destination itineraries makes a lot of sense due to the fact that repetitively calling remote services will incur much higher overhead and latency. It is much faster to implement and run such functionality by a single service provider with lower overhead. Therefore we decide to include signature 5.4.

5.5.3 Search for Round-Trip Itineraries

We could also provide signatures for services to search for round-trip itineraries as in signature 5.5. However, we choose not to include it because it is a spe-

Signature 5.4 searchMultidestItineraries

[(AIRPORT, AIRPORT, INTERVAL)] → [ITINERARY]

Searches for multi-destination itinerary given a list of tuples of origin airport, destination airport, and departure time interval.

Input is a list of tuples, where in each tuple the first element is the origin airport, the second element the destination airport. The last element is the time interval of the intended departure time. Each tuple in the input list must follow the previous tuple in time and not overlap with each other.

Return a list of multi-destination itineraries.

cial case of signature 5.4 with the input list containing two tuples, the first from origin airport to destination airport, the second back from destination to origin. Should the demand for such services grow higher later, we can then add signature 5.5 to make it more convenient to use. We leave it out now for simplicity.

5.5.4 Search for Itineraries by Particular Airlines

For each of the service previously listed, we need extended versions of them with additional input parameter to specify a particular airline or airline alliances so that service consumers can use to implement advanced functionalities for end users to nail down the results. There are several different ways to do this. The most straightforward one would be to design an interface

Signature 5.5 searchRoundtripItineraries

(AIRPORT, AIRPORT, INTERVAL, INTERVAL) → [ITINERARY]

Searches for round-trip itineraries between two airports.

The first element of input is the origin airport, the second the destination airport. The third element is the time interval of intended departure time of the outward flight from origin to destination, and the last element is the time interval of intended departure time of the return flight from destination to origin.

Return a list of round-trip itineraries.

taking an additional airline parameter and another taking an airline alliance parameter. However this would require two additional interfaces for each service we designed before. A better way would be to have an interface that takes an additional list of airlines. This way it is possible to specify just one airline, several airlines, or all airlines in an airline alliance when the service consumers invoke some helper service to lookup the member airlines of an airline alliance. Therefore we decide to go with the more powerful and flexible design.

For example, signature 5.1 is extended to signature 5.6 with an additional element in the input type which is the preferred list of airlines. We extend signature 5.2, 5.3, and 5.4 respectively to get signature 5.7, 5.8, and 5.9.

Signature 5.6 searchOnewayItinerariesByAirlines

(AIRPORT, AIRPORT, INTERVAL, [AIRLINE]) → [ITINERARY]

Signature 5.7 searchDirectItinerariesByAirlines

(AIRPORT, AIRPORT, INTERVAL, [AIRLINE]) → [ITINERARY]

Signature 5.8 searchNonstopItinerariesByAirlines

(AIRPORT, AIRPORT, INTERVAL, [AIRLINE]) → [ITINERARY]

Signature 5.9 searchMultidestItinerariesByAirlines

[(AIRPORT, AIRPORT, INTERVAL, [AIRLINE])] → [ITINERARY]

5.5.5 Auxiliary Services

In addition to the basic search functionalities, our repository will also need some auxiliary utilities to help service consumers to implement their applications.

We expect most of the service providers in our domain deal primarily with searching functionalities given two airports, because there is less ambiguity and most scheduling is done based on airports anyway. However, many end users (travelers) will likely specify origins or destinations using names of cities instead of airports. Therefore service consumers (application developers) will need to translate the names of cities to airports using additional geographic information. They are likely to demand services that can do this translation for them.

One of the helper utilities will search for airports accessible from a given city. The syntactic interface is defined in signature 5.10.

Signature 5.10 lookupAirports

CITY → [AIRPORT]

Take a city and return a list of affiliated airports.

The reverse functionality to look up the parent city of an airport is necessary, too. Its syntactic interface is defined in signature 5.11. Note that we assume every airport has exactly one parent city in the repository.

Signature 5.11 lookupCity

AIRPORT → CITY

Take an airport and return its parent city.

We also need two additional services to lookup the affiliating relationship between airlines and airline alliances as shown in signature 5.12 and 5.13.

Signature 5.12 lookupAirlineAlliance

AIRLINE → ALLIANCE

Take an airline and return the alliance it belongs to.

Signature 5.13 lookupMemberAirlines

ALLIANCE → [AIRLINE]

Take an airline alliance and return a list of member airlines which join the alliance.

5.6 Additional Predicates

In order to properly describe properties of data types, we need many additional predicates. One of the design decisions is that we will expose to service providers and consumers only a language based on propositional logic. The specifications will be easier to understand, to reason about, and to verify by a machine. Specifically, we will restrain ourselves from using universal and existential quantifiers. We believe the omission of quantifiers will make it easier for people not very familiar with logic to correctly use repositories, as quantifiers tend to make it rather complicated to understand and write specifications. (Although in the underlying formal system we must allow quantifiers to deal with specification composition—see next chapter)

The consequence of the decision is that we will have multiple predicates to describe similar conceptual ideas. We do not lose much of the expressiveness by eliminating quantifiers since we can design extra predicates that fulfill the objective of using quantifiers. For example, let's consider a service specification that describes direct itinerary searches. A list of itineraries should be returned. We want to assert that for each of the itineraries in the output, the first flight must depart from the origin airport. We introduce

a dedicated predicate called allItinerariesDepartFromAirport([ITINERARY], AIRPORT). It takes a list of itineraries and an airport and asserts that for each itinerary in the list, the first flight of that itinerary departs from the same origin airport. Similarly we will have a predicate allItinerariesArriveAtAirport([ITINERARY], AIRPORT) that asserts all itineraries in a list arrive at the same destination airport. If we were to use a universal quantifier, the expression would take the form $(\forall i \in is : \text{itineraryArrivesAtAirport}(i, a))$, where *is* is a set of itineraries.

The other necessary predicates will be explained in subsequent sections when they appear for the first time.

5.7 Complete Specifications

We now add the necessary pre- and post-conditions to the signatures described before.

5.7.1 Search for One-Way Itineraries

Signature 5.2 assumes three input parameters: the origin airport, the destination airport and the departure time interval. It returns a list of one-way Itineraries. We include the proper pre- and post-conditions to describe its semantic interface and we get specification 5.1 as a result.

The predicate validAirport(AIRPORT) asserts that an airport is considered valid, i.e. operational in our imaginary world. The predicate allItineraries-DepartureTimeWithin([ITINERARY], INTERVAL) asserts that in a list of

Specification 5.1 searchOnewayItineraries

(AIRPORT, AIRPORT, INTERVAL) $\rightarrow$ [ITINERARY]

ϕ: validAirport(in(0)) $\wedge$ validAirport(in(1))

ψ: allItinerariesDepartFromAirport(out, in(0)) $\wedge$
allItinerariesArriveAtAirport(out, in(1)) $\wedge$
allItinerariesDepartureTimeWithin(out, in(2)) $\wedge$
completeListOfOnewayItineraries(out, in)

itineraries all initial flights depart within a given time interval. The exact meanings of these predicates are left to reference implementations, which are not the primary concerns of this thesis.

Two special names, *in* and *out*, are used to designate the input and output value of services being specified. Notice that to access each element of the input tuple, a zero-based indexed scheme is used, therefore in(0) means the first element of the input tuple, while in(1) refers to the second element.

At the first glance it may not be apparent why we have completeListOfOnewayItineraries(out, in) in the post-condition. It may seem that the previous three predicates are sufficient to describe the output which is a list of flights. After careful consideration, though, it can be found that a lazy service that always returns an empty list will also satisfy the post-condition. We want to exclude these lazy services by putting more restrictions on the output. We would like to guarantee that services which properly implement this specification should indeed return all (or at least most) itineraries leaving

from the origin airport to the destination airport during the specified window of departure time. For this reason we added the predicate completeListOfOnewayItineraries([ITINERARY], (AIRPORT, AIRPORT, INTERVAL)). Its reference implementation would check if the output list does indeed include every itinerary (but nothing else) that satisfies the input condition. There is a catch however: the definition of a *complete list* of itineraries is rather vague. A Toronto-Frankfurt itinerary might fly a traveler from Toronto to Vancouver first, then to Beijing and finally to Frankfurt. Should such itinerary belongs to the list of results if the traveler searches for flights from Toronto to Frankfurt? Technically it should, but it does not make much sense for sane people. So when explaining the meaning of the predicate to users (service providers and consumers) we would impose some reasonable restrictions on the reference implementation. These restrictions might be rather elaborate (e.g. return true only if the list contains itineraries with no more than three connections and where the longest itinerary takes maximum double the miles than the shortest one and where the longest itinerary cannot take more than 12 hours more than the shortest). The nice thing about it is that when using the predicate in specifications and queries we just need to remember completeListOfOnewayItineraries([ITINERARY], (AIRPORT, AIRPORT, INTERVAL)). Even for outsiders, the predicate name carries a substantial amount of information.

The similar signature 5.2 for searching direct itineraries between two airports is completed with semantic interface as listed in specification 5.2. Signature 5.3 receives similar treatment, resulting in specification 5.3.

Specification 5.2 searchDirectItineraries

(AIRPORT, AIRPORT, INTERVAL) $\rightarrow$ [ITINERARY]

ϕ: validAirport(in(0)) $\wedge$ validAirpoort(in(1))

ψ: allItinerariesDepartFromAirport(out, in(0)) $\wedge$
allItinerariesArriveAtAirport(out, in(1)) $\wedge$
allItinerariesDepartureTimeWithin(out, in(2)) $\wedge$
completeListOfDirectItineraries(out, in)

Specification 5.3 searchNonstopItineraries

(AIRPORT, AIRPORT, INTERVAL) $\rightarrow$ [ITINERARY]

ϕ: validAirport(in(0)) $\wedge$ validAirport(in(1))

ψ: allItinerariesDepartFromAirport(out, in(0)) $\wedge$
allItinerariesArriveAtAirport(out, in(1)) $\wedge$
allItinerariesDepartureTimeWithin(out, in(2)) $\wedge$
completeListOfNonstopItineraries(out, in)

5.7.2 Search for Multi-Destination Itineraries

Services that search for multi-destination itineraries turn out to be rather difficult to specify due to the variable length of input sequence. The challenge is that we can not easily describe the properties of elements in the variable-length input sequence, as our current approach is geared to static descriptions. To compensate our inability to specify the dynamic details,

we rely on elaborate predicates such as validMultidestSequence([(AIRPORT, AIRPORT, INTERVAL)]). There are a few things this predicate asserts, namely,

1. The origin airport in each tuple of the sequence must be valid.
2. The destination airport in each tuple of the sequence must be valid.
3. The time interval in each tuple of the sequence must follow the previous interval without any overlapping.

In the post-condition, we use the predicate completeListOfMultidestItineraries([ITINERARY], (AIRPORT, AIRPORT, INTERVAL)) to assert a list of itineraries are all indeed multi-destination itineraries and it contains all the sensible results that satisfy the input condition. Predicate allMutlidestItinerariesSatisfy([ITINERARY], [(AIRPORT, AIRPORT, INTERVAL)]) asserts that each of the multi-destination itinerary in the list designated by the first parameter satisfies the constraints of sequence in the second parameter. The constraints include

1. Each segment of a multi-destination itinerary must depart from origin airport for destination airport within the given time interval listed in the sequence.
2. The segments of each multi-destination itinerary must be in the same order as listed in the sequence.

The specification of services that search for multi-destination itineraries is defined in specification 5.4 using signature 5.4.

Specification 5.4 searchMultidestItineraries

[(AIRPORT, AIRPORT, INTERVAL)] → [ITINERARY]

ϕ: validMultidestSequence(in)

ψ: completeListOfMultidestItineraries(out, in) ∧
allMultidestItinerariesSatisfy(out, in)

As we see in the previous examples, there are varying degrees of granularity when choosing predicates. On one hand, we can have very fine-grained predicates that describe limited aspects of properties so that we can easily compose many of them to describe more complex relationships; on the other hand we can have coarse-grained predicates that by themselves describe rather elaborate properties of complicated structures. The choice of what to use depends on many factors and the repository maintainers have to make good decisions.

5.7.3 Search for Round-Trip Itineraries

We mentioned earlier that we are not going to include specifications for services that search for round-trip itineraries. Nonetheless, it is a good exercise to try to formally define the specification in case we need it later due to popular demand.

To search for round-trip itineraries we will need to supply two departure time intervals, one for the outward flight and one for the return flight. There is an additional constraint that the departure time of the return flight should

be later than the arrival time of the outward flight, otherwise the traveler will not be able to take the flights given our current understanding of time in physics (no time traveling!). A new predicate isLaterThan(INTERVAL, INTERVAL) is introduced to express the relationship that the first interval of time follows the second without overlapping.

Round-trip itineraries are specified separately by its outward and return flights. We want the outward flight to leave from the origin for the destination during the first departure time interval, and the return flight to leave from the destination back for the origin during the second departure time interval. Predicate allOutwardFlightsDepartFrom([ITINERARY], AIRPORT) asserts that the outward flights of a list of round-trip itineraries in the first parameter has the same origin airport as specified by the second parameter. Similarly, predicate allOutwardFlightsArriveAt([ITINERARY], AIRPORT) asserts that the outward flight of a list of round-trip itineraries arrives at a given airport. Predicate allOutwardFlightsDepartureTimeWithin([ITINERARY], INTERVAL) asserts the departure time of the outward flight of the list of round-trip itineraries should lie in the interval indicated by the second parameter of the predicate. Three more similar predicates are introduced to assert the properties of the return flights. The complete definition is illustrated in specification 5.5 using signature 5.5.

5.7.4 Search for Itineraries by Airlines

Remember for each specification listed previously, we also have an extended version that takes an additional list of airlines in the input to specify preferred

Specification 5.5 searchRoundtripItineraries

(AIRPORT, AIRPORT, INTERVAL, INTERVAL) → [ITINERARY]

ϕ: validAirport(in(0)) ∧ validAirport(in(1)) ∧ isLaterThan(in(3), in(2))

ψ: allOutwardFlightsDepartFrom(out, in(0)) ∧
allOutwardFlightsArriveAt(out, in(1)) ∧
allOutwardFlightsDepartureTimeWithin(out, in(2)) ∧
allReturnFlightsDepartFrom(out, in(1)) ∧
allReturnFlightsArriveAt(out, in(0)) ∧
allReturnFlightsDepartureTimeWithin(out, in(3)) ∧
completeListOfRoundtripItineraries(out, in)

airlines. We will write down their specifications here.

For example, the extended version of specification 5.1 is given in specification 5.6. The additional predicate validAirlines([AIRLINE]) asserts the list of airline instances must be valid, and in the post-condition predicate allItinerariesOperatedByAirlines([ITINERARY], [AIRLINE]) guarantees that each itinerary in the result list must be operated by one of the airlines given in the input.

Similar treatments are applied to all specifications listed previously and we get additional specification 5.7, 5.8, and 5.9.

Specification 5.6 searchOnewayItinerariesByAirlines

(AIRPORT, AIRPORT, INTERVAL, [AIRLINE]) → [ITINERARY]

ϕ: validAirport(in(0)) ∧ validAirport(in(1)) ∧ validAirlines(in(3))

ψ: allItinerariesDepartFromAirport(out, in(0)) ∧
allItinerariesArriveAtAirport(out, in(1)) ∧
allItinerariesDepartureTimeWithin(out, in(2)) ∧
allItinerariesOperatedByAirlines(out, in(3)) ∧
completeListOfOnewayItinerariesByAirlines(out, in)

Specification 5.7 searchDirectItinerariesByAirlines

(AIRPORT, AIRPORT, INTERVAL, [AIRLINE]) → [ITINERARY]

ϕ: validAirport(in(0)) ∧ validAirpoort(in(1)) ∧ validAirlines(in(3))

ψ: allItinerariesDepartFromAirport(out, in(0)) ∧
allItinerariesArriveAtAirport(out, in(1)) ∧
allItinerariesDepartureTimeWithin(out, in(2)) ∧
allItinerariesOperatedByAirlines(out, in(3)) ∧
completeListOfDirectItinerariesByAirlines(out, in)

5.7.5 Auxiliary Specifications

The auxiliary utility to lookup accessible airports from a city given in signature 5.10 assumes a valid city (predicate validCity(CITY)) with at least one airport (predicate cityWithAirport(CITY)) as input, and returns a list

Specification 5.8 searchNonstopItinerariesByAirlines

(AIRPORT, AIRPORT, INTERVAL, [AIRLINE]) → [ITINERARY]

ϕ: validAirport(in(0)) ∧ validAirport(in(1)) ∧ validAirlines(in(3))

ψ: allItinerariesDepartFromAirport(out, in(0)) ∧
allItinerariesArriveAtAirport(out, in(1)) ∧
allItinerariesDepartureTimeWithin(out, in(2)) ∧
allItinerariesOperatedByAirlines(out, in(3)) ∧
completeListOfNonstopItinerariesByAirlines(out, in)

Specification 5.9 searchMultidestItinerariesByAirlines

[(AIRPORT, AIRPORT, INTERVAL, [AIRLINE])] → [ITINERARY]

ϕ: validMultidestSequence(in) ∧ validAirlines(in(3))

ψ: allMultidestItinerariesSatisfy(out, in) ∧
allItinerariesOperatedByAirlines(out, in(3)) ∧
completeListOfMultidestItinerariesByAirlines(out, in)

of airports that belong to the input city. The predicate allAirportsInCity([AIRPORT], CITY) asserts that a list of airports in its first parameter belong to the given city indicated by its second parameter. The complete specification is provided in 5.10.

Likewise, the reverse to lookup the parent city of an airport given in signature 5.11 is extended to specification 5.11 with semantic information. The

Specification 5.10 lookupAirports

CITY $\rightarrow$ [AIRPORT]

ϕ: validCity(in) $\wedge$ cityWithAirport(in)

ψ: allValidAirports(out) $\wedge$ allAirportsInCity(out, in)

predicate airportInCity(AIRPORT, CITY) is similar to the previous predicate allAirportsInCity([AIRPORT], CITY), but deals with a single airport as its first parameter instead of a list of airports. It asserts the airport is in the city given by the second parameter.

Specification 5.11 lookupCity

AIRPORT $\rightarrow$ CITY

ϕ: validAirport(in)

ψ: validCity(out) $\wedge$ airportInCity(input, output)

Signature 5.12 and 5.13 are extended to specification 5.12 and 5.13 respectively. Predicate validAirline(AIRLINE) asserts the input airline must be valid, and predicate airlineInAlliance(AIRLINE, ALLIANCE) asserts the airline given by the first parameter is a member of the airline alliance given by the second parameter. Similarly, predicate allAirlinesInAlliance([AIRLINE], ALLIANCE) asserts all airlines given by the first parameter are members of the airline alliance given by the second parameter.

Specification 5.12 lookupAirlineAlliance

AIRLINE → ALLIANCE

ϕ: validAirline(in)

ψ: airlineInAlliance(input, output)

Specification 5.13 lookupMemberAirlines

ALLIANCE → [AIRLINE]

ϕ: validAlliance(in)

ψ: allAirlinesInAllince(out, in)

5.8 Axioms and Composing Services

We include in our repository service specification 5.14 which for any given flight returns the destination airport. Furthermore we include in the repository service specification 5.15, which returns the city in which the input airport is located.

Specification 5.14 getDestinationAirport

FLIGHT → AIRPORT

ϕ: validFlight(in)

ψ: flightToAirport(input, output)

Specification 5.15 getParentCity

AIRPORT → CITY

ϕ: validAirport(in)

ψ: airportInCity(input, output)

Now let's consider the situation where a service consumer requires a service that takes a flight and returns the arrival city. The specification is provided in 5.16.

Specification 5.16 getDestinationCity

FLIGHT → CITY

ϕ: validFlight(in)

ψ: flightToCity(input, output)

Our repository does not include such specification. Still, it is obvious to the human observer that we could combine a service that implements specification 5.14 with a service that implements specification 5.15. First we determine the destination airport for the flight and then we use the second service to obtain the corresponding city. When we combine the two specifications we get specification 5.17 (see section 7.5 for details about composition).

If this new specification matches specification 5.16, our repository could return the composition path of specification 5.14 and 5.15 as a result to a

Specification 5.17 composedSpecification

FLIGHT $\rightarrow$ CITY

ϕ: validFlight(in)

ψ: $\exists$tmp (validFlight(in) $\wedge$ flightToAirport(input, tmp) $\wedge$
validAirport(tmp) $\wedge$ airportInCity(tmp, output))

query to search for specification 5.16 and service consumer who queried for specification 5.16 could safely combine the mentioned services.

Unfortunately, the underlying logic system is not able to deduce this dependency. To address this issue, we include in the repository a set of axioms. Such axioms encode a valid human knowledge about the domain. In our example the axiom

$$\text{flightToAirport(flight, airport)} \wedge \text{airportInCity(airport, city)} \Rightarrow \text{flightToCity(flight, city)}$$

will do the trick and enable the derivation of the desired matching relation. Note that for this to work we need to assert that

$$\exists\text{tmp (validFlight(in)} \wedge \text{flightToAirport(input, tmp)} \wedge\text{validAirport(tmp)} \wedge \text{airportInCity(tmp, output))} \Rightarrow \text{flightToAirport(flight, airport)} \wedge \text{airportInCity(airport, city))}$$

More on this in Chapter 7.

5.9 Complex Composition of Services

Some compositions of specifications are not that simple, though. For example, currently we do not have specifications for services that search for flights between two cities. It is possible to combine services that search for flights between airports and services that lookup affiliated airports of a given city to create such functionality.

The approach is to first lookup all affiliated airports of the origin city and all affiliated airports of the destination city. For each pair in the Cartesian product of the list of origin airports and the list of destination airports, invoke the services that search for flights between the pair of airports and get a list of possible itineraries. Finally concatenate the resulting lists of itineraries for each pair of airports to produce a complete list of itineraries that depart from the origin city and arrive at the destination city.

Ideally we would like the repository to automatically figure out the possibility of combining multiple specifications to generate new ones. The challenge is that the repository is not smart enough to map individual elements of a tuple to meet the requirement of another specification reliably without human guidance. Therefore we need to encode the necessary information to help the repository in the form of additional specifications, predicates, and axioms.

Using the previous example, the first step is to make the repository know that it can transform the input tuple (CITY, CITY, INTERVAL) into a new tuple ([AIRPORT], [AIRPORT], INTERVAL) as defined in specification 5.18. The transformation should preserve as much information as possible from

the original input in the transformed result, which is expressed in the form of post-condition of the specification.

Specification 5.18 cityCityIntervalToAirportsAirportsInterval

(CITY, CITY, INTERVAL) $\rightarrow$ ([AIRPORT], [AIRPORT], INTERVAL)

ϕ: validCity(in(0)) $\wedge$ cityWithAirport(in(0)) $\wedge$ validCity(in(1)) $\wedge$ cityWithAirport(in(1))

ψ: allValidAirports(out(0)) $\wedge$ allAirportsInCity(out(0), in(0)) $\wedge$ allValidAirports(out(1)) $\wedge$ allAirportsInCity(out(1), in(1)) $\wedge$ sameInterval(out(2), in(2))

In specification 5.18 we put the necessary predicates to guarantee that the returned list of origin airports must all belong to the origin city, the returned list of destination airports must all belong to the destination city, and the departure time interval must stay untouched.

The second step is that we have to take the transformed input and produce the necessary result. The specification is defined in 5.19. Notice that since now we have more than one airport as the origin and destination, we have to guarantee that the itineraries returned as a result must depart from one of the origin airports and arrive at one of the destination airports. Two new predicates, allItinerariesDepartFromAmong([ITINERARY], [AIRPORT]) and allItinerariesArriveAtAmong([ITINERARY], [AIRPORT]), are introduced for this purpose.

Specification 5.19 searchItinerariesAirportsToAirports

([AIRPORT], [AIRPORT], INTERVAL) → [ITINERARY]

ϕ: allValidAirports(in(0)) ∧ allValidAirports(in(1))

ψ: allItinerariesDepartFromAmong(out, in(0)) ∧
allItinerariesArriveAtAmong(out, in(1)) ∧
allItinerariesDepartureTimeWithin(out, in(2))

The actual implementation of specification 5.19 can be relayed back to service consumers as instructions on how to invoke the services since their task is to assemble ready-made service components to make final applications. The repository will provide them with necessary pieces to get the job done, but not necessarily babysit them all the time.

5.9.1 Composed Specification

With the extra specifications in hand, it is now possible to combine them as the output type of specification 5.18 matches the input type of 5.19, and the post-condition of specification 5.18 implies the pre-condition of specification 5.19. As a result of the composition we will get a new specification as in 5.20.

The special name *tmp* in the post-condition designates the intermediate output from specification 5.18 and used as the input to specification 5.19. It is introduced because we would like to preserve as much as possible the guarantees of the original specifications. More on this in section 7.5.

Specification 5.20 composedSearch

(CITY, CITY, INTERVAL) $\rightarrow$ ([AIRPORT], [AIRPORT], INTERVAL) $\rightarrow$ [ITINERARY]

ϕ: validCity(in(0)) $\wedge$ cityWithAirport(in(0)) $\wedge$ validCity(in(1)) $\wedge$ cityWithAirport(in(1))

ψ: $\exists$tmp (allValidAirports(tmp(0)) $\wedge$ allAirportsInCity(tmp(0), in(0)) $\wedge$ allValidAirports(tmp(1)) $\wedge$ allAirportsInCity(tmp(1), in(1)) $\wedge$ sameInterval(tmp(2), in(2)) $\wedge$ allItinerariesDepartFromAmong(out, tmp(0)) $\wedge$ allItinerariesArriveAtAmong(out, tmp(1)) $\wedge$ allItinerariesDepartureTimeWithin(out, tmp(2)))

5.9.2 Additional Axioms

We also need additional axioms to help reasoning about the composed specification. It is obvious that, if a list of itineraries all depart from some airports, and all these airports belong to a particular city, we can conclude that the itineraries must depart from the given city as a result. The following axiom restates this:

$$\begin{aligned} &\text{allItinerariesDepartFromAmong(Itineraries, Airports)} \\ &\quad \wedge \text{allAirportsInCity(Airports, City)} \\ &\Rightarrow \text{allItinerariesDepartFromCity(Itineraries, City)} \end{aligned} \tag{5.1}$$

The same logic applies when a list of itineraries all arrive at some airports which all belong to a city, these itineraries therefore all arrive at the particular

city. Thus we have the following axiom:

$$\begin{aligned} &\text{allItinerariesArriveAtAmong(Itineraries, Airports)} \\ &\quad \wedge \text{allAirportsInCity(Airports, City)} \\ &\Rightarrow \text{allItinerariesArriveAtCity(Itineraries, City)} \end{aligned} \tag{5.2}$$

5.9.3 Instructions versus Service Implementations

The more complicated form of composition introduced in this section illustrates the limitation of completely automated composition done by the repository: the repository is only smart enough to do the simplest form of *chaining* without additional human knowledge. Advanced composition like the one shown in this section requires human insight to guide the repository. The extra specifications designed to help the repository to reason about composition might not necessarily have concrete service implementations. They might be instructions that service consumers need to read and understand that they have to write some clue code in order to properly compose services. This is less a concern in our current design, though.

Chapter 6

Logic Foundation

In this chapter we will introduce the logic foundation of the language we will use to describe service repositories. Our treatment is based on standard first-order logic and we summarize the main concepts and results throughout this chapter. The treatment is primarily based on [4] with minor modifications to fit our context.

6.1 First-Order Language

We define the syntax of a first-order language in this section.

6.1.1 Constants

Constants are names to some objects we are concerned. A constant refers to a concrete object. The same object can have multiple constants as its name. For example, we use the constant "Toronto" to refer to the city with that name

in Canada, and we use the constant "AC698" to refer to a flight operated by Air Canada.

6.1.2 Predicate Symbols

Predicate symbols are used to express some property of objects and relations between objects. For example, a predicate can express the relation that flight AC698 departs from Toronto: depart(AC698, Toronto). In this example, flight AC698 and Toronto are called the *arguments* of the predicate "depart". The number of arguments of a predicate is called its *arity*. In the previous example, predicate "depart" has two arguments, so it has arity of two. A predicate with arity of n is said to be n-ary. There are some special names: *unary* for $n = 1$, *binary* for $n = 2$, and *trinary* for $n = 3$.

The order of a predicate's arguments is important. The expression depart(AC698, Toronto) has a different meaning than depart(Toronto, AC698). The latter does not make sense if the predicate defines a relation that its first argument departs from the second argument.

6.1.3 Atomic Sentences

An *atomic sentence* is a formed by a predicate followed by the same number of arguments as its arity. Sentences make *claims*, which can have a *truth value* of either *true* or *false*. We will explain more about this in the subsequent section.

6.1.4 Logic Connectives

We use *logic connectives* to join simpler sentences to form more complex sentences. The five logic connectives we will use include *conjunction* (symbol $\wedge$), *disjunction* (symbol $\vee$), *negation* (symbol $\neg$), material conditional (symbol $\Rightarrow$), and material biconditional (symbol $\Leftrightarrow$).

The negation symbol $\neg$ expresses the opposite case of the sentence followed by it. For example, we want to express the case that flight AC698 does *not* depart from Toronto. We could write $\neg$depart(AC698, Toronto). The sentence has the negated truth value as depart(AC698, Toronto).

The conjunction symbol $\wedge$ joins two sentences. The result is true only if both sentences are true. For example, we want to express the case that flight AC698 departs from Toronto *and* that flight AC150 departs from Vancouver. We write depart(AC698, Toronto) $\wedge$ depart(AC150, Vancouver).

The disjunction symbol $\vee$ joins two sentences. The result is true if either sentence is true (they could both be true). For example, depart(AC698, Toronto) $\vee$ depart(AC150, Vancouver) means that either flight AC698 departs from Toronto, or flight AC150 departs from Vancouver.

The *material conditional* symbol $\Rightarrow$ is used to combine two sentences. The sentence $\psi \Rightarrow \phi$ is equivalent to $\neg\psi \vee \phi$.

The *material biconditional* symbol $\Leftrightarrow$ is used to combine two sentences. The sentence $\psi \Leftrightarrow \phi$ is equivalent to $(\psi \Rightarrow \phi) \wedge (\phi \Rightarrow \psi)$.

6.1.5 Variables and Quantifiers

Variables are placeholder symbols that can appear as arguments of predicates. However, they do not refer to concrete objects. They indicate relationships between *quantifiers* and the arguments of predicates.

We use the universal quantifier (symbol $\forall$) to express universal claims that usually contain words like *every*, *all*, and *each*. The universal quantifier is always used together with a variable to *bind* it. For example, $\forall x$ means "for every object x". The expression $\forall x$ depart(x, Toronto) says that every object x departs from Toronto.

Similarly, we use the existential quantifier (symbol $\exists$) to express existential claims that usually contain words like *some*, *there is one*, and *at least one*. The existential quantifier is always used together with a variable to bind it. For example, $\exists x$ means "there is an object x". The expression $\exists x$ depart(x, Toronto) says that there is an object x that departs from Toronto.

6.1.6 Formulas

An expression of a predicate followed by its arguments (either constants or variables) is called an *atomic well-formed formula*. Later we will just say *atomic formulas* for short. For example, the expression depart(AC698, x) is an atomic formula where x is a variable.

Complex formulas can be built by combining simpler formulas with logic connectives. If ψ and ϕ are both formulas, so is

- $\neg\psi$

- $\psi \wedge \phi$
- $\psi \vee \phi$
- $\psi \Rightarrow \phi$
- $\psi \Leftrightarrow \phi$
- $\forall x \psi$, any occurrence of x in ψ is said to be bound
- $\exists x \psi$, any occurrence of x in ψ is said to be bound

The precedence of logic connectives is, $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, from highest to lowest. $\forall$ and $\exists$ quantifiers bind the variable in the formula following them. Parentheses are necessary if the formula is not an atomic one.

Variables appearing in formulas without corresponding quantifiers to bind them are said to be *free* or *unbound*. Otherwise, they are *bound*. Sentences are formulas with no free variables.

The expression $\psi[v \leftarrow v']$ is the formula ψ with all free occurrences of variable v replaced by a new variable v'.

6.2 First-Order Structures

So far we have defined the syntax of first-order languages. We now consider their semantics. We introduce the notion of *first-order structures* to give meanings to constants and predicates in first-order languages. A *first-order structure* contains a set of objects and a set of predicates that determine the truth values of sentences in a first-order language.

Definition 1. *Let $\mathfrak{L}$ be a first-order language. A first-order structure $\mathfrak{M}$ for $\mathfrak{L}$ is a tuple $\langle D^{\mathfrak{M}}, P^{\mathfrak{M}}, N^{\mathfrak{M}} \rangle$ where $D^{\mathfrak{M}}$ is a nonempty set of objects called the domain of discourse; $P^{\mathfrak{M}}$ is a function from predicates in $\mathfrak{L}$ to their extensions; $N^{\mathfrak{M}}$ is a naming function from constants in $\mathfrak{L}$ to objects in $D^{\mathfrak{M}}$.*

Variables, universal quantifiers, and existential quantifiers range over objects in $D^{\mathfrak{M}}$. The naming function $N^{\mathfrak{M}}$ assigns objects in $D^{\mathfrak{M}}$ to constants in $\mathfrak{L}$. If c is a constant in $\mathfrak{L}$, $N^{\mathfrak{M}}(c)$ is the object in $D^{\mathfrak{M}}$ that it refers to.

An n-ary predicate p is represented in $\mathfrak{M}$ by its *extension* $p^{\mathfrak{M}}$, a subset of the set of n-tuple $\langle t_1, t_2, t_3, ..., t_n \rangle$ for $t_1, t_2, t_3, ..., t_n \in D^{\mathfrak{M}}$. Each tuple in $p^{\mathfrak{M}}$ is a fact that the relation described by p holds for the objects in that tuple. For a predicate p, its extension $p^{\mathfrak{M}}$ is given by $P^{\mathfrak{M}}(p)$.

6.2.1 Variable Assignment

In a first-order language, we need to deal with variables appearing in formulas.

Definition 2 (Variable assignments). *Let $\mathfrak{M}$ be a first-order structure with a domain of discourse $D^{\mathfrak{M}}$. A variable assignment is a (possibly partial) function defined on a set of variables and taking values in $D^{\mathfrak{M}}$.*

We use the notation $[v_i \mapsto o_i]$ to denote a variable assignment which assigns object o_i to variable v_i. For example, $[v_1 \mapsto o_1, v_2 \mapsto o_2]$ assigns objects o_1 and o_2 in $D^{\mathfrak{M}}$ to variables v_1 and v_2, respectively.

Definition 3 (Appropriate variable assignments). *Let $\mathfrak{M}$ be a first-order structure for a first-oder language $\mathfrak{L}$ with a domain of discourse $D^{\mathfrak{M}}$. A variable assign-*

ment is appropriate for a formula ψ in $\mathfrak{L}$ if it assigns objects in $D^{\mathfrak{M}}$ to each free variable of ψ.

Given a variable assignment $g = [v_1 \mapsto o_1, ..., v_k \mapsto o_k ..., v_n \mapsto o_n]$, the expression $g(v_k)$ gives the corresponding object o_k in the assignment. g can be *modified* using the notation $g[v_t/o_t]$. The modification changes the assignment such that the domain of g is extended by v_t and v_t is assigned object o_t. The rest of the assignment stays unchanged. There is a special assignment called the *empty variable assignment* that does not assign objects to any variables.

6.2.2 Satisfaction and Truth

Definition 4 (Satisfaction). *Let $\mathfrak{M}$ be a first-order structure $\langle D^{\mathfrak{M}}, P^{\mathfrak{M}}, N^{\mathfrak{M}} \rangle$ for a first-order language $\mathfrak{L}$. Let ψ, ϕ, and ω be formulas in $\mathfrak{L}$. Let g be a variable assignment in $\mathfrak{M}$ that is appropriate for ψ.*

- *If ψ is an atomic formula $p(t_1, t_2, ..., t_n)$ where p is an n-ary predicate, then g satisfies ψ iff $\left\langle [\![t_1]\!]_g^{\mathfrak{M}}, [\![t_2]\!]_g^{\mathfrak{M}}, ..., [\![t_n]\!]_g^{\mathfrak{M}} \right\rangle \in p^{\mathfrak{M}}$ where $[\![t_k]\!]_g^{\mathfrak{M}}$ is $g(t_k)$ if t_k is a variable in $\mathfrak{L}$, or $N^{\mathfrak{M}}(t_k)$ if t_k is a constant in $\mathfrak{L}$ for $k \in \{1, 2, ..., n\}$.*
- *If ψ is $\neg\phi$, then g satisfies ψ iff g does not satisfy ϕ.*
- *If ψ is $\phi \wedge \omega$, then g satisfies ψ iff g satisfies both ϕ and ω.*
- *If ψ is $\phi \vee \omega$, then g satisfies ψ iff g satisfies either ϕ or ω, or both.*
- *If ψ is $\phi \Rightarrow \omega$, then g satisfies ψ iff g does not satisfy ϕ or g satisfies ω or both.*

- *If ψ is $\phi \Leftrightarrow \omega$, then g satisfies ψ iff g satisfies both ϕ and ω or neither.*
- *If ψ is $\forall x\phi$, then g satisfies ψ iff for every $o \in D^{\mathfrak{M}}$, $g[x/o]$ satisfies ϕ.*
- *If ψ is $\exists x\phi$, then g satisfies ψ iff for some $o \in D^{\mathfrak{M}}$, $g[x/o]$ satisfies ϕ.*

If a variable assignment g satisfies a formula ψ in $\mathfrak{M}$, we write $\mathfrak{M} \models \psi[g]$.

Definition 5 (Truth). *Let $\mathfrak{M}$ be a first-order structure for a first-order language $\mathfrak{L}$. A sentence ψ is true in $\mathfrak{M}$ iff the empty variable assignment satisfies ψ in $\mathfrak{M}$. Otherwise ψ is false. We write $\mathfrak{M} \models \psi$ if ψ is true in $\mathfrak{M}$.*

Definition 6 (Theory and Model). *A set of sentences $\mathcal{T}$ in a given first-order language $\mathfrak{L}$ is a called a theory. A structure $\mathfrak{M}$ that satisfies every sentence in $\mathcal{T}$ is called a model for $\mathcal{T}$, denoted by $\mathfrak{M} \models \mathcal{T}$.*

Definition 7 (First-Order Consequences). *A sentence ψ is a* first-order consequence *of a theory $\mathcal{T}$ iff for every model $\mathfrak{M}$ of $\mathcal{T}$, $\mathfrak{M} \models \psi$.*

6.3 Deductive System

Given a theory $\mathcal{T}$ and a sentence ψ, we want to know if there is a proof of ψ from the premise $\mathcal{T}$. We write $\mathcal{T} \vdash \psi$ if there is a proof. We arrive at proofs with the help of a deductive system. There are many kinds of deductive systems. We use the deductive system $\mathcal{F}$ in [4] with the following rules:

Axiom

$$\mathcal{T} \vdash \phi, \text{ for all } \phi \in \mathcal{T}$$

Conjunction Introduction

$$\frac{\mathcal{T} \vdash \psi \quad \mathcal{T} \vdash \phi}{\mathcal{T} \vdash \psi \wedge \phi}$$

Conjunction Elimination

$$\frac{\mathcal{T} \vdash \psi \wedge \phi}{\mathcal{T} \vdash \psi} \quad \frac{\mathcal{T} \vdash \psi \wedge \phi}{\mathcal{T} \vdash \phi}$$

Disjunction Introduction

$$\frac{\mathcal{T} \vdash \psi}{\mathcal{T} \vdash \psi \vee \phi} \quad \frac{\mathcal{T} \vdash \phi}{\mathcal{T} \vdash \psi \vee \phi}$$

Disjunction Elimination

$$\frac{\mathcal{T} \vdash \psi \vee \phi \quad \mathcal{T} \vdash \psi \Rightarrow \omega \quad \mathcal{T} \vdash \phi \Rightarrow \omega}{\mathcal{T} \vdash \omega}$$

Negation Introduction

$$\frac{\mathcal{T} \vdash \psi \Rightarrow (\phi \wedge \neg\phi)}{\mathcal{T} \vdash \neg\psi}$$

Negation Elimination

$$\frac{\mathcal{T} \vdash \neg\neg\psi}{\mathcal{T} \vdash \psi}$$

Conditional Introduction

$$\frac{\mathcal{T} \cup \{\phi\} \vdash \psi}{\mathcal{T} \vdash (\phi \Rightarrow \psi)}$$

Conditional Elimination

$$\frac{\mathcal{T} \vdash \phi \Rightarrow \psi, \quad \mathcal{T} \vdash \phi}{\mathcal{T} \vdash \psi}$$

Universal Introduction

$$\frac{\mathcal{T} \vdash \psi(c)}{\mathcal{T} \vdash \forall x \psi(x)}$$

c is an arbitrary constant that does not occur outside the subproof of $\psi(c)$ where it is introduced.

Universal Elimination

$$\frac{\mathcal{T} \vdash \forall x \psi(x)}{\mathcal{T} \vdash \psi(c)}$$

c is an arbitrary constant.

Existential Introduction

$$\frac{\mathcal{T} \vdash \psi(c)}{\mathcal{T} \vdash \exists x \psi(x)}$$

c is an arbitrary constant.

Existential Elimination

$$\frac{\mathcal{T} \vdash \exists x \psi(x) \quad \mathcal{T} \cup \{\psi(c)\} \vdash \phi}{\mathcal{T} \vdash \phi}$$

c is an arbitrary constant that does not occur outside the subproof of ϕ where it is introduced.

Theorem 1 (Soundness of $\mathcal{F}$). *If $\mathcal{T} \vdash \psi$, ψ is a first-order consequence of $\mathcal{T}$.*

The theorem states that if we can prove a sentence ψ from a set of sentences $\mathcal{T}$ using deductive system $\mathcal{F}$, then ψ must be true in every structure $\mathfrak{M}$ in which sentences in $\mathcal{T}$ are true.

Theorem 2 (Completeness of $\mathcal{F}$). *If ψ is a first-order consequence of $\mathcal{T}$, then $\mathcal{T} \vdash \psi$.*

The theorem states that if ψ is true in every structure $\mathfrak{M}$ in which sentences in $\mathcal{T}$ are true, then we can prove ψ from premise $\mathcal{T}$ using deductive system $\mathcal{F}$.

Proofs of the two theorems listed above can be found in [4].

Chapter 7

Repository Formalization

In the previous chapter we defined first-order languages and structures. In this chapter we will use them to formally define various pieces of service repositories.

A service repository is a collection of service specifications. Each specification has a corresponding set of services offered by different service providers that implement it. Although these actual services will eventually be invoked by service consumers, they play a rather secondary role in our discussions. Service specifications provide us with information about services' signature (input and output types) and the pre- and post-conditions that a service must satisfy. We need a collection of data types admissible in a repository and a collection of predicates over these data types that we use to formulate pre- and post-conditions. The choice of data types and predicates is part of the repository design.

Service repositories are domain-specific. When we formalize a service

repository, we always consider a particular set of objects in a domain and a particular set of relations between these objects. Using the terminology introduced in the previous chapter, we have a fixed first-order structure $\mathfrak{M} = \langle D^{\mathfrak{M}}, P^{\mathfrak{M}}, N^{\mathfrak{M}} \rangle$ for a service repository with a fixed first-order language $\mathfrak{L}$. It should be noted that all semantic definitions introduced below will implicitly assume such a structure $\mathfrak{M}$.

7.1 Data Types

Data types are one of the fundamental building blocks of service repositories. A repository will have a finite set of basic data types that can be used to express the most basic concepts. These basic data types include primitive data types such as STRING, INTEGER, FLOAT, BOOLEAN, and various others like the ones usually found in a mainstream programming language, as well as domain specific data types identified during the design phase.

Collectively, these basic data types are denoted by the symbol $\mathbb{T}^B$. We construct the set of all possible data types in a service repository, denoted by the symbol $\mathbb{T}$, with the help of a few collection type constructors such as LIST, MAP, and TUPLE.

Definition 8. *Let $\mathbb{T}^B$ be a finite set of basic data type names. We inductively define*

$\mathbb{T}$, *the set of repository data type names, as*

$$\mathbb{T}^B \subseteq \mathbb{T}$$

$$\forall t \in \mathbb{T} : \textsc{List}[t] \in \mathbb{T}$$

$$\forall t_1, t_2 \in \mathbb{T} : \textsc{Map}[t_1, t_2] \in \mathbb{T}$$

$$\forall t_1, t_2 \in \mathbb{T} : \textsc{Tuple}[t_1, t_2] \in \mathbb{T}$$

Each repository data type specifies a particular kind of objects. Any object that follows the specification is regarded as an *instance* of the type. The sum of those instances define the semantics of any data type $t \in \mathbb{T}$. The union of all instances of each data type constitutes the domain of discourse $D^{\mathfrak{M}}$. In other words, the semantic meaning of a data type in a service repository is a subset of the corresponding domain of discourse $D^{\mathfrak{M}}$.

For each type $t \in \mathbb{T}$, there is a corresponding predicate τ_t that has a single variable. $\tau_t(o)$ is true if o is of type t.

7.1.1 Implementation of Data Types

All data types are backed up by concrete implementations in a given host language chosen by service providers and service consumers. The choice of host languages is not the concern of the service repository, though, since the repository will contain only names referring to these data types.

Nevertheless, to clarify our intentions in this chapter we will choose Scala as a host language. Scala [19] is a statically-typed language built on top of Java Virtual Machine with a clear and concise syntax.

In our case, each data type corresponds to a Scala class. We only need the class to have necessary attributes to contain information about a particular instance of a data type. Because these data instances are immutable, we do not need the class to contain other methods. In other words, think of these classes more like data records. We define an operator to show the relationship between a data type name and its reference implementation.

Let $T \in \mathbb{T}$ be a data type. The expression $\textcircled{S}(T)$ means the reference implementation of T in the chosen host language. For example, in our chosen host language Scala, the implementation of the data type AIRPORT is backed up by a class called Airport, with the necessary attributes to hold information related to the data type. The attribute *name* stores the name of the airport, while the attribute *code* stores the unique three-letter IATA airport code of the airport.

$$\textcircled{S}(\text{AIRPORT}) = \text{class Airport(name: String, code: String)}$$

The semantic meaning of a data type can be understood as the collection of all instances of the reference implementation in the host language.

Definition 9. *Let $T \in \mathbb{T}$ be a data type. The meaning of T is defined by all instances of a corresponding reference implementation of T in a host language.*

$$[\![T]\!] = \{o \mid o \text{ is an instance of } \textcircled{S}(T)\}$$

For example, the following piece of Scala code demonstrates the creation of three instances of the AIRPORT type. The collection of all instances of

airports in our imaginary world defines the meaning of the type AIRPORT.

```
1 val airport1 = new Airport("Toronto Pearson International", "YYZ")
2 val airport2 = new Airport("Vancouver International", "YVR")
3 val airport3 = new Airport("St. John's International, ", "YYT")
4 val airport4 = new Airport("Ottawa International", "FRA") // ?
```

Note in the previous example, the last instance is considered problematic because in the real world Ottawa International Airport has the IATA code YOW while the code FRA belongs to Frankfurt International Airport in Germany. Still, airport4 belongs to $[\![\text{AIRPORT}]\!]$. Whether an instance is considered valid in our context depends on predicates.

7.2 Services and Service Specifications

The actual services that providers host and consumers eventually invoke are not our primary concern in a service repository since we are interested only in their specification. Nevertheless, we define them formally in order to have a context to describe service specifications later.

Definition 10 (Service). *A service f in model $\mathfrak{M}$ is a binary relation on the domain of discourse $D^{\mathfrak{M}}$.*

$$f \subseteq D^{\mathfrak{M}} \times D^{\mathfrak{M}}$$

Definition 11 (Total Service). *A service f is total iff the corresponding binary relation is left-total, i.e.*

$$\forall x \in D^{\mathfrak{M}}\ \exists y \in D^{\mathfrak{M}}\ \langle x, y \rangle \in f$$

In practice, most services are defined only on some subsets of $D^{\mathfrak{M}}$. The behavior of a service taking an input out of range is undetermined. The most common way to describe the range of input a service accepts and the range of output a service produces is to specify the input and output type.

Definition 12. *The combination of input and output type of a service is called the service's signature, or its syntactic interface.*

For example, a service that takes an airport and returns a flight departing from that airport will have the signature

$$\text{getFlight} : \text{AIRPORT} \times \text{FLIGHT}$$

where AIRPORT is its input type and FLIGHT is its output type.

We describe a set of services sharing the same signature, pre-condition, and post-condition with a service specification.

Definition 13 (Service Specifications). *Let $\mathfrak{L}$ be a first-order language. An $\mathfrak{L}$-service specification is a pair of formulas (ϕ, ψ) where ϕ contains one free variable in, and ψ contains two free variables in and out.*

Let $\mathbb{T}$ be a set of types. We write $\langle \pi, \omega, \phi, \psi \rangle$ as a syntactic sugar for an $\mathfrak{L}$-specification $(\tau_\pi(in) \wedge \phi(in), \tau_\pi(in) \wedge \tau_\omega(out) \wedge \psi(in, out))$, where $\pi \in \mathbb{T}$ is the input type, $\omega \in \mathbb{T}$ is the output type.

If it is clear from the context which $\mathfrak{L}$ we are talking about, we will just say service specifications. The previous example service that returns a flight departing from an airport can be described by the following specification $(\tau_{\text{AIRPORT}}(in) \wedge \text{validAirport}(in), \tau_{\text{FLIGHT}}(out) \wedge \text{validFlight}(out) \wedge$

flightFromAirport(out, in)). An equivalent notation is $\langle$AIRPORT, FLIGHT, validAirport(in), validFlight$(out) \wedge$ flightFromAirport$(out, in)\rangle$.

The meaning of a service specification is the collection of actual services that can be described by the specification.

Definition 14. *Let $\mathfrak{L}$ be a first-order language. Let $\mathfrak{M}$ be a model for $\mathfrak{L}$ with a domain of discourse $D^{\mathfrak{M}}$. The meaning of an $\mathfrak{L}$-specification (ϕ, ψ) in $\mathfrak{M}$ is defined by the set of services that implement the specification.*

$$[\![(\phi, \psi)]\!]_{\mathfrak{M}} = \{f \subseteq D^{\mathfrak{M}} \times D^{\mathfrak{M}} \mid \forall \langle x, y\rangle \in f\, \mathfrak{M} \models (\phi \Rightarrow \psi)[in \mapsto x, out \mapsto y]\}$$

(see definition 4 for the meaning of $\mathfrak{M} \models \psi[g]$.)

7.3 Service Repositories

Definition 15. *Let $\mathfrak{L}$ be a first-order language. $S^{\mathfrak{L}}$ is the set of all possible $\mathfrak{L}$-service specifications, and $S^{\mathfrak{L}}_f \subset S^{\mathfrak{L}}$ is the set of all possible quantifier-free (no $\forall$ or $\exists$) $\mathfrak{L}$-service specifications.*

A service repository consists primarily of service specifications. There are infinite number of quantifier-free service specifications in a first-order language. A service repository can contain only a finite number of them. In addition, each specification in a repository must point to a number of concrete services that implement it.

Definition 16. *A repository $\mathcal{R}$ is a tuple $\langle \mathbb{T}, \mathfrak{L}, \mathcal{T}, \mathfrak{M}, S^{\mathfrak{L}}_{\mathcal{R}}, m_{\mathcal{R}}\rangle$, where $\mathbb{T}$ is the set of data types, $\mathfrak{L}$ is a first-order language, $\mathcal{T}$ is an $\mathfrak{L}$-theory, $\mathfrak{M}$ is a first-order*

structure for $\mathfrak{L}$ such that $\mathfrak{M} \models \mathcal{T}$, $S^{\mathfrak{L}}_{\mathcal{R}} \subset S^{\mathfrak{L}}_{f}$ is a finite set of quantifier-free $\mathfrak{L}$-service specifications and $m_{\mathcal{R}}$ is a mapping from $S^{\mathfrak{L}}_{\mathcal{R}}$ to the power set of $V_{\mathcal{R}}$ such that $m_{\mathcal{R}}(s) \subset [\![s]\!]$. Here $V_{\mathcal{R}} = \bigcup_{s \in S^{\mathfrak{L}}_{\mathcal{R}}} [\![s]\!]$ is the set of concrete services.

7.4 Matching of Service Specifications

It is common to have services that satisfy multiple service specifications. As a special case, we are interested in the situation where all services satisfying a particular service specification happen to satisfy another service specification, because then we can treat the former specification as if it were the latter specification. This is vital for querying a service repository later when we introduce queries.

Definition 17 (Matching). *Let (ϕ_1, ψ_1) and (ϕ_2, ψ_2) be two $\mathfrak{L}$-service specifications. We say that (ϕ_1, ψ_1) matches (ϕ_2, ψ_2) under theory $\mathcal{T}$, denoted by*

$$(\phi_1, \psi_1) \sqsubseteq_{\mathcal{T}} (\phi_2, \psi_2)$$

if and only if

$$\mathcal{T} \vdash \forall in\ (\phi_2 \Rightarrow \phi_1) \tag{7.1}$$

$$\mathcal{T} \vdash \forall in\ \forall out\ (\phi_2 \wedge \psi_1 \Rightarrow \psi_2) \tag{7.2}$$

The reader might wonder why we have such conditions as in (7.1) and (7.2). At first glance, the two conditions seem rather counter-intuitive. The idea is that if an input is valid under ϕ_2 and it is to be fed into services

described by (ϕ_2, ψ_2), it should also be valid under ϕ_1 so that we can feed the value into services described by (ϕ_1, ψ_1). Similarly, if an input and an output value of services described by (ϕ_1, ψ_1) satisfy ψ_1, we should be able to use the result in any context a result from services described by (ϕ_2, ψ_2) is expected. Only in this way can we say that services described by (ϕ_1, ψ_1) are also described by (ϕ_2, ψ_2).

One extra technicality in condition (7.2) is the presence of ϕ_2, which seems to be unnecessary at first. The reason it is there is to preserve maximum possibility of matching. It is best explained by an example.

Suppose we have a specification $s_1 = (\phi_1, \psi_1)$ for services that take an integer input value and produce an integer output value such that the input value is dividable by two, and the output value is two times the input value. We write $\phi_1 = \text{DivByTwo}(in)$ and $\psi_1 = \text{TwoTimes}(out, in)$.

Now suppose we have another specification $s_2 = (\phi_2, \psi_2)$ for services that take an integer input value and produce an integer output value such that the input value is dividable by six, and the output value is dividable by four. We write $\phi_2 = \text{DivBySix}(in)$ and $\psi_2 = \text{DivByFour}(out)$.

Human experts have no problem deducing that $s_1 \sqsubseteq_{\mathcal{T}} s_2$ based on the fact that

$$\text{DivBySix}(in) \Rightarrow \text{DivByTwo}(in) \tag{7.3}$$

and

$$\text{DivBySix}(in) \wedge \text{TwoTimes}(out, in) \Rightarrow \text{DivByFour}(out) \tag{7.4}$$

We can encode the human knowledge of the two facts and add them as

axioms to the system. With the help of condition (7.2) we can deduce that $s_1 \sqsubseteq_{\mathcal{T}} s_2$.

If, however, condition (7.2) comes in the form

$$\mathcal{T} \vdash \forall in\, \forall out\, (\psi_1 \Rightarrow \psi_2)$$

even with the added axioms the system will not be able to deduce that $s_1 \sqsubseteq_{\mathcal{T}} s_2$.

Proposition 1 (Reflexivity of $\sqsubseteq_{\mathcal{T}}$). *Let $s = (\phi, \psi)$ be an $\mathfrak{L}$-specification and let $\mathcal{T}$ be a theory in $\mathfrak{L}$. Then $s \sqsubseteq_{\mathcal{T}} s$.*

Proof. This is very straightforward:

$$\mathcal{T} \vdash \forall in(\phi \Rightarrow \phi)$$
$$\mathcal{T} \vdash \forall in \forall out(\phi \wedge \psi \Rightarrow \psi)$$

□

Proposition 2 (Transitivity of $\sqsubseteq_{\mathcal{T}}$). *Let s_1, s_2, and s_3 be three $\mathfrak{L}$-specifications and let $\mathcal{T}$ be a theory in $\mathfrak{L}$. If $s_1 \sqsubseteq_{\mathcal{T}} s_2$ and $s_2 \sqsubseteq_{\mathcal{T}} s_3$, then $s_1 \sqsubseteq_{\mathcal{T}} s_3$*

Proof. Let

$$s_1 = (\phi_1, \psi_1)$$
$$s_2 = (\phi_2, \psi_2)$$
$$s_3 = (\phi_3, \psi_3)$$

Because $s_1 \sqsubseteq_{\mathcal{T}} s_2$ and $s_2 \sqsubseteq_{\mathcal{T}} s_3$, we have

$$\mathcal{T} \vdash \forall in(\phi_2 \Rightarrow \phi_1) \tag{7.5}$$

$$\mathcal{T} \vdash \forall in \forall out(\phi_2 \wedge \psi_1 \Rightarrow \psi_2) \tag{7.6}$$

$$\mathcal{T} \vdash \forall in(\phi_3 \Rightarrow \phi_2) \tag{7.7}$$

$$\mathcal{T} \vdash \forall in \forall out(\phi_3 \wedge \psi_2 \Rightarrow \psi_3) \tag{7.8}$$

By (7.5) and (7.7)

$$\mathcal{T} \vdash \forall in(\phi_3 \Rightarrow \phi_1)$$

By (7.7)

$$\mathcal{T} \vdash \forall in \forall out((\phi_3 \wedge \psi_1) \Rightarrow (\phi_2 \wedge \psi_1))$$

By (7.6)

$$\mathcal{T} \vdash \forall in \forall out((\phi_3 \wedge \psi_1) \Rightarrow \psi_2)$$

By (7.8)

$$\mathcal{T} \vdash \forall in \forall out((\phi_3 \wedge \psi_1) \Rightarrow \psi_3)$$

Therefore, $s_1 \sqsubseteq_{\mathcal{T}} s_3$. □

7.4.1 Matching of Specifications and Subset of Services

Our original intention to define the matching relationship of two service specifications is to clarify the subset relationship of the underlying services they describe. We give a brief proof that the idea holds.

Theorem 3. *Let s_1 and s_2 be two $\mathfrak{L}$-service specifications. Let $\mathcal{T}$ be a theory in $\mathfrak{L}$ and $\mathfrak{M}$ be a structure for $\mathfrak{L}$ such that $\mathfrak{M} \models \mathcal{T}$.*

$$(s_1 \sqsubseteq_{\mathcal{T}} s_2) \Rightarrow ([\![s_1]\!]_{\mathfrak{M}} \subseteq [\![s_2]\!]_{\mathfrak{M}})$$

Proof. Let

$$s_1 = (\phi_1, \psi_1)$$
$$s_2 = (\phi_2, \psi_2)$$

By condition (7.1) and (7.2), we know

$$\mathcal{T} \vdash \forall in(\phi_2 \Rightarrow \phi_1)$$
$$\mathcal{T} \vdash \forall in \forall out(\phi_2 \wedge \psi_1 \Rightarrow \psi_2)$$

Since $\mathfrak{M} \models \mathcal{T}$, by theorem 1 it follows that

$$\mathfrak{M} \models \forall in(\phi_2 \Rightarrow \phi_1) \tag{7.9}$$

$$\mathfrak{M} \models \forall in \forall out(\phi_2 \wedge \psi_1 \Rightarrow \psi_2) \tag{7.10}$$

The two sets of services represented by the two specifications, respectively, are

$$[\![(\phi_1, \psi_1)]\!]_{\mathfrak{M}} = \{f \in D^{\mathfrak{M}} \times D^{\mathfrak{M}} \mid \forall \langle x, y\rangle \in f\ (\mathfrak{M} \models (\phi_1 \Rightarrow \psi_1)[in \mapsto x, out \mapsto y])\}$$

$$[\![(\phi_2, \psi_2)]\!]_{\mathfrak{M}} = \{f \in D^{\mathfrak{M}} \times D^{\mathfrak{M}} \mid \forall \langle x, y\rangle \in f\ (\mathfrak{M} \models (\phi_2 \Rightarrow \psi_2)[in \mapsto x, out \mapsto y])\}$$

Now we need to show that an arbitrary service $f \in [\![s_1]\!]_{\mathfrak{M}}$, also $f \in [\![s_2]\!]_{\mathfrak{M}}$ holds. We do it by showing that for all $\langle x, y\rangle \in f$

$$(\mathfrak{M} \models (\phi_1 \Rightarrow \psi_1)[in \mapsto x, out \mapsto y])$$
$$\Rightarrow (\mathfrak{M} \models (\phi_2 \Rightarrow \psi_2)[in \mapsto x, out \mapsto y])$$

which is equivalent to

$$
\begin{aligned}
& \forall \langle x, y \rangle \in f\, \mathfrak{M} \models ((\phi_1 \Rightarrow \psi_1) \Rightarrow (\phi_2 \Rightarrow \psi_2))[in \mapsto x, out \mapsto y] \\
\overset{\text{Shunting}}{\iff} \quad & \mathfrak{M} \models ((\phi_1 \Rightarrow \psi_1) \wedge \phi_2 \Rightarrow \psi_2)[in \mapsto x, out \mapsto y] \\
\overset{\text{Def. of} \Rightarrow}{\iff} \quad & \mathfrak{M} \models (\neg((\neg\phi_1 \vee \psi_1) \wedge \phi_2) \vee \psi_2)[in \mapsto x, out \mapsto y] \\
\overset{\text{Distribution}}{\iff} \quad & \mathfrak{M} \models (\neg((\neg\phi_1 \wedge \phi_2) \vee (\psi_1 \wedge \phi_2)) \vee \psi_2)[in \mapsto x, out \mapsto y] \\
\overset{\text{De Morgan}}{\iff} \quad & \mathfrak{M} \models (\neg(\neg\phi_1 \wedge \phi_2) \wedge \neg(\psi_1 \wedge \phi_2)) \vee \psi_2)[in \mapsto x, out \mapsto y] \\
\overset{\text{De Morgan}}{\iff} \quad & \mathfrak{M} \models (((\phi_1 \vee \neg\phi_2) \wedge \neg(\psi_1 \wedge \phi_2)) \vee \psi_2)[in \mapsto x, out \mapsto y] \\
\overset{\text{Distribution}}{\iff} \quad & \mathfrak{M} \models ((\phi_1 \vee \neg\phi_2) \vee \psi_2) \wedge (\neg(\psi_1 \wedge \phi_2) \vee \psi_2)[in \mapsto x, out \mapsto y] \\
\overset{\text{Def. of} \Rightarrow}{\iff} \quad & \mathfrak{M} \models ((\phi_2 \Rightarrow \phi_1) \vee \psi_2) \wedge (\psi_1 \wedge \phi_2 \Rightarrow \psi_2)[in \mapsto x, out \mapsto y]
\end{aligned}
$$

which is the case by (7.9) and (7.10).

□

Corollary 1 (Semantic equivalence). *Let s_1 and s_2 be two $\mathfrak{L}$-specifications. Let $\mathcal{T}$ be a theory in $\mathfrak{L}$ and $\mathfrak{M}$ be a structure for $\mathfrak{L}$ such that $\mathfrak{M} \models \mathcal{T}$. If $s_1 \sqsubseteq_{\mathcal{T}} s_2$ and $s_2 \sqsubseteq_{\mathcal{T}} s_1$, then $[\![s_1]\!]_{\mathfrak{M}} = [\![s_2]\!]_{\mathfrak{M}}$.*

Proof. By theorem 3, if $s_1 \sqsubseteq_{\mathcal{T}} s_2$ and $s_2 \sqsubseteq_{\mathcal{T}} s_1$, then $[\![s_1]\!]_{\mathfrak{M}} \subseteq [\![s_2]\!]_{\mathfrak{M}}$ and $[\![s_2]\!]_{\mathfrak{M}} \subseteq [\![s_1]\!]_{\mathfrak{M}}$. Therefore $[\![s_1]\!]_{\mathfrak{M}} = [\![s_2]\!]_{\mathfrak{M}}$. □

7.5 Composition

Many services can be chained together to perform more complex operations. We desire a way to describe the behavior of these chained services by combining their specifications.

Definition 18 (Composition of Specifications). *Let $\mathcal{T}$ be a first-order theory in a first-order language $\mathfrak{L}$. Let (ϕ_1, ψ_1) and (ϕ_2, ψ_2) be two $\mathfrak{L}$-service specifications with free variables in_1, out_1 respectively in_2, out_2. We say that (ϕ_1, ψ_1) can be composed with (ϕ_2, ψ_2) under $\mathcal{T}$, denoted by*

$$(\phi_2, \psi_2) \circ_{\mathcal{T}} (\phi_1, \psi_1)$$

if and only if

$$\mathcal{T} \vdash \forall in_1 \forall out_1 (\phi_1 \wedge \psi_1 \Rightarrow (\phi_2[in_2 \leftarrow out_1])) \tag{7.11}$$

The result of the composition is a new specification

$$(\phi_2, \psi_2) \circ_{\mathcal{T}} (\phi_1, \psi_1) = (\phi_1, \exists tmp\ (\psi_1[out_1 \leftarrow tmp] \wedge \psi_2[in_2 \leftarrow tmp]))$$

with the free variables in_1 and out_2. The variable tmp must be chosen so that it does not cause name collisions with existing variables in the formula.

It is pretty obvious that equation (7.11) must hold. We cannot combine two specifications if the result from the first specification does not satisfy the precondition of the second.

The resulting composed specification, however, deserves more attention. The reader might ask why we could not simply use ψ_2 as the post-condition for the composed specification. The answer is that ψ_2 is a formula about the relationship between the input and output of the second specification, while the desired post-condition of the composed specification must describe

the relationship between the output of the second specification with respect to the input of the first specification. This is where the mysterious intermediate temporary value *tmp* comes into play: it is the output of the first component of the composition and the input of the second component of the composition.

Proposition 3 (Monotonicity of composition). *Let $\mathfrak{L}$ be a first-order language and $\mathcal{T}$ be a theory in $\mathfrak{L}$. Let s_1, s_2, s_3 and s_4 be $\mathfrak{L}$-specifications. Furthermore let $s_1 \sqsubseteq_{\mathcal{T}} s_3$, $s_2 \sqsubseteq_{\mathcal{T}} s_4$, $s_2 \circ_{\mathcal{T}} s_1$ and $s_4 \circ_{\mathcal{T}} s_3$. Then $s_2 \circ_{\mathcal{T}} s_1 \sqsubseteq_{\mathcal{T}} s_4 \circ_{\mathcal{T}} s_3$.*

Proof. Let $s_1 = (\phi_1, \psi_1)$, $s_2 = (\phi_2, \psi_2)$, $s_3 = (\phi_3, \psi_3)$, $s_4 = (\phi_4, \psi_4)$.

Because $s_1 \sqsubseteq_{\mathcal{T}} s_3$ and $s_2 \sqsubseteq_{\mathcal{T}} s_4$, we have

$$\mathcal{T} \vdash \forall in(\phi_3 \Rightarrow \phi_1) \tag{7.12}$$

$$\mathcal{T} \vdash \forall in \forall out(\phi_3 \wedge \psi_1 \Rightarrow \psi_3) \tag{7.13}$$

$$\mathcal{T} \vdash \forall in(\phi_4 \Rightarrow \phi_2) \tag{7.14}$$

$$\mathcal{T} \vdash \forall in \forall out(\phi_4 \wedge \psi_2 \Rightarrow \psi_4) \tag{7.15}$$

In addition, because $s_2 \circ_{\mathcal{T}} s_1$ and $s_4 \circ_{\mathcal{T}} s_3$, we have

$$\mathcal{T} \vdash \forall in \forall out(\phi_1 \wedge \psi_1 \Rightarrow (\phi_2[in \leftarrow out])) \tag{7.16}$$

$$\mathcal{T} \vdash \forall in \forall out(\phi_3 \wedge \psi_3 \Rightarrow (\phi_4[in \leftarrow out])) \tag{7.17}$$

The results of the two compositions are

$$s_2 \circ_{\mathcal{T}} s_1 = (\phi_1, \exists tmp\ \psi_1[out \leftarrow tmp] \wedge \psi_2[in \leftarrow tmp]) \tag{7.18}$$

$$s_4 \circ_{\mathcal{T}} s_3 = (\phi_3, \exists tmp\ \psi_3[out \leftarrow tmp] \wedge \psi_4[in \leftarrow tmp]) \tag{7.19}$$

We need to show that

$$\mathcal{T} \vdash \forall in(\phi_3 \Rightarrow \phi_1) \tag{7.20}$$

which is given by (7.12), and

$$\begin{aligned} \mathcal{T} \vdash \forall in \forall out(\phi_3 \wedge \exists tmp(\psi_1[out \leftarrow tmp] \wedge \psi_2[in \leftarrow tmp]) \\ \Rightarrow \exists tmp(\psi_3[out \leftarrow tmp] \wedge \psi_4[in \leftarrow tmp])) \end{aligned} \tag{7.21}$$

which is shown by the steps below:

$$\begin{aligned} & \mathcal{T} \vdash \forall in \forall out(\phi_3 \wedge \exists tmp(\psi_1[out \leftarrow tmp] \wedge \psi_2[in \leftarrow tmp])) \\ \Longrightarrow\ & \mathcal{T} \vdash \forall in \forall out \exists tmp(\phi_3 \wedge \psi_1[out \leftarrow tmp] \wedge \psi_2[in \leftarrow tmp]) \\ \Longrightarrow\ & \mathcal{T} \vdash \forall in \forall out \exists tmp(\phi_3 \wedge \phi_3 \wedge \psi_1[out \leftarrow tmp] \wedge \psi_2[in \leftarrow tmp]) \\ \overset{\text{by (7.13)}}{\Longrightarrow}\ & \mathcal{T} \vdash \forall in \forall out \exists tmp(\phi_3 \wedge \psi_3[out \leftarrow tmp] \wedge \psi_2[in \leftarrow tmp]) \\ \Longrightarrow\ & \mathcal{T} \vdash \forall in \forall out \exists tmp(\phi_3 \wedge \psi_3[out \leftarrow tmp] \wedge \psi_3[out \leftarrow tmp] \wedge \psi_2[in \leftarrow tmp]) \\ \overset{\text{by (7.17)}}{\Longrightarrow}\ & \mathcal{T} \vdash \forall in \forall out \exists tmp(\phi_4[in \leftarrow tmp] \wedge \psi_3[out \leftarrow tmp] \wedge \psi_2[in \leftarrow tmp]) \\ \overset{\text{by (7.15)}}{\Longrightarrow}\ & \mathcal{T} \vdash \forall in \forall out \exists tmp(\psi_3[out \leftarrow tmp] \wedge \psi_4[in \leftarrow tmp]) \end{aligned}$$

□

Definition 19 (Composition of Services). *Let $\mathfrak{M}$ be a model with an domain of*

discourse $D^{\mathfrak{M}}$. *Let* $f, g \subseteq D^{\mathfrak{M}} \times D^{\mathfrak{M}}$ *be two services.*

The composition of f *and* g, *denoted by* $g \circ f$, *is a new service*

$$g \circ f = \{\langle x, z\rangle \in D^{\mathfrak{M}} \times D^{\mathfrak{M}} |\ \exists y \in D^{\mathfrak{M}}\ (\langle x, y\rangle \in f \wedge \langle y, z\rangle \in g)\}$$

Theorem 4. *Let* $\mathcal{T}$ *be a first-order theory in* $\mathfrak{L}$ *and let* $\mathfrak{M}$ *be a structure for* $\mathfrak{L}$ *such that* $\mathfrak{M} \models \mathcal{T}$. *Moreover let* $s_2 \circ_{\mathcal{T}} s_1$. *If* $f_1 \in [\![s_1]\!]_{\mathfrak{M}}$ *and* $f_2 \in [\![s_2]\!]_{\mathfrak{M}}$, *then* $f_2 \circ f_1 \in [\![s_2 \circ_{\mathcal{T}} s_1]\!]_{\mathfrak{M}}$.

Proof. Let $s_1 = (\phi_1, \psi_1)$, $s_2 = (\phi_2, \psi_2)$ with free variables in_1, out_1 respectively in_2, out_2. By definition of $[\![s_2 \circ_{\mathcal{T}} s_1]\!]$, we need to show that

$$\begin{aligned}\forall \langle x, z\rangle \in f_2 \circ f_1\ \mathfrak{M} \models (\phi_1 \Rightarrow (\exists tmp\ \psi_1[out_1 \leftarrow tmp] \\ \wedge \psi_2[in_2 \leftarrow tmp]))[in_1 \mapsto x, out_2 \mapsto z]\end{aligned}$$

which is equivalent to

$$\begin{aligned}\forall \langle x, z\rangle \in f_2 \circ f_1\ \mathfrak{M} \models \exists tmp\ (\phi_1 \Rightarrow (\psi_1[out_1 \leftarrow tmp] \\ \wedge \psi_2[in_2 \leftarrow tmp]))[in_1 \mapsto x, out_2 \mapsto z]\end{aligned} \tag{7.22}$$

Later on we will show that

$$\begin{aligned}\forall \langle x, z\rangle \in f_2 \circ f_1\ \mathfrak{M} \models (\exists tmp\ (\phi_1 \Rightarrow \psi_1[out_1 \leftarrow tmp]) \\ \wedge(\phi_2[in_2 \leftarrow tmp] \Rightarrow \psi_2[in_2 \leftarrow tmp]))[in_1 \mapsto x, out_2 \mapsto z]\end{aligned} \tag{7.23}$$

and for all $x, z \in D^{\mathfrak{M}}$

$$\mathfrak{M} \models (\exists tmp(\phi_1 \wedge \psi_1[out_1 \leftarrow tmp] \Rightarrow \phi_2[in_2 \leftarrow tmp]))[in_1 \mapsto x, out_2 \mapsto z] \tag{7.24}$$

Therefore

$$\begin{aligned} \forall \langle x, z \rangle \in f_2 \circ f_1 \, \mathfrak{M} \models (\exists tmp \, ((\phi_1 \Rightarrow \psi_1[out_1 \leftarrow tmp]) \wedge \\ (\phi_2[in_2 \leftarrow tmp] \Rightarrow \psi_2[in_2 \leftarrow tmp]) \wedge (\phi_1 \wedge \psi_1[out_1 \leftarrow tmp] \\ \Rightarrow \phi_2[in_2 \leftarrow tmp])))[in_1 \mapsto x, out_2 \mapsto z] \end{aligned} \tag{7.25}$$

Let now: $P = \phi_1$, $Q = \psi_1[out_1 \leftarrow tmp]$, $R = \phi_2[in_2 \leftarrow tmp]$, $S = \psi_2[in_2 \leftarrow tmp]$. We have

$$\begin{aligned} &(P \Rightarrow Q) \wedge (R \Rightarrow S) \wedge (P \wedge Q \Rightarrow R) \\ \Rightarrow &(P \Rightarrow Q) \wedge (P \wedge Q \Rightarrow S) \\ \Rightarrow &(P \Rightarrow Q \wedge S) \end{aligned}$$

Therefore, (7.25) implies (7.22) and we are done.

In the following subproofs we show that (7.23) and (7.24) hold. We begin with (7.24).

Since $s_2 \circ_{\mathcal{T}} s_1$, by condition (7.11) we know that

$$\mathcal{T} \vdash \forall in_1 \forall out_1 (\phi_1 \wedge \psi_1 \Rightarrow \phi_2[in_2 \leftarrow out_1])$$

By renaming variables we get:

$$\mathcal{T} \vdash \forall in_1 \forall tmp(\phi_1 \wedge \psi_1[out_1 \leftarrow tmp] \Rightarrow \phi_2[in_2 \leftarrow tmp])$$

Since $\mathfrak{M} \models \mathcal{T}$, by theorem 1 we have

$$\mathfrak{M} \models \forall in_1 \forall tmp(\phi_1 \wedge \psi_1[out_1 \leftarrow tmp] \Rightarrow \phi_2[in_2 \leftarrow tmp])$$

and therefore for all $x, y \in D^{\mathfrak{M}}$

$$\mathfrak{M} \models (\phi_1 \wedge \psi_1[out_1 \leftarrow tmp] \Rightarrow \phi_2[in_2 \leftarrow tmp])[in_1 \mapsto x, tmp \mapsto y]$$

hence $\forall x \in D^{\mathfrak{M}}$

$$\mathfrak{M} \models \exists tmp(\phi_1 \wedge \psi_1[out_1 \leftarrow tmp] \Rightarrow \phi_2[in_2 \leftarrow tmp])[in_1 \mapsto x]$$

and because out_2 does not occur in the above formula, we can extend the assignment so that $\forall x, z \in D^{\mathfrak{M}}$

$$\mathfrak{M} \models \exists tmp(\phi_1 \wedge \psi_1[out_1 \leftarrow tmp] \Rightarrow \phi_2[in_2 \leftarrow tmp])[in_1 \mapsto x, out_2 \mapsto z]$$

which is (7.24).

Now we prove (7.23). The composed specification is

$$s_2 \circ_{\mathcal{T}} s_1 = (\phi_1, (\exists tmp\ \psi_1[out_1 \leftarrow tmp] \wedge \psi_2[in_2 \leftarrow tmp]))$$

Because $f_1 \in [\![s_1]\!]_{\mathfrak{M}}$ and $f_2 \in [\![s_2]\!]_{\mathfrak{M}}$, we know that

$$\forall \langle x, y \rangle \in f_1 \; \mathfrak{M} \models (\phi_1 \Rightarrow \psi_1)[in_1 \mapsto x, out_1 \mapsto y] \tag{7.26}$$

$$\forall \langle y, z \rangle \in f_2 \; \mathfrak{M} \models (\phi_2 \Rightarrow \psi_2)[in_2 \mapsto y, out_2 \mapsto z] \tag{7.27}$$

By definition 19 we also know that

$$\forall \langle x, z \rangle \in f_2 \circ f_1 \; \exists y \in D^{\mathfrak{M}} \; (\langle x, y \rangle \in f_1 \wedge \langle y, z \rangle \in f_2) \tag{7.28}$$

From (7.26), (7.27), and (7.28) we have

$$\begin{aligned} \forall \langle x, z \rangle \in f_2 \circ f_1 \; \exists y \in D^{\mathfrak{M}} ((\mathfrak{M} \models (\phi_1 \Rightarrow \psi_1)[in_1 \mapsto x, out_1 \mapsto y]) \\ \wedge (\mathfrak{M} \models (\phi_2 \Rightarrow \psi_2)[in_2 \mapsto y, out_2 \mapsto z])) \end{aligned} \tag{7.29}$$

We extend the assignments by adding variables that do not occur in the formulas so that

$$\begin{aligned} &\forall \langle x, z \rangle \in f_2 \circ f_1 \; \exists y \in D^{\mathfrak{M}} \\ &(\mathfrak{M} \models (\phi_1 \Rightarrow \psi_1)[in_1 \mapsto x, out_1 \mapsto y, in_2 \mapsto y, out_2 \mapsto z]) \\ &\wedge (\mathfrak{M} \models (\phi_2 \Rightarrow \psi_2)[in_1 \mapsto x, out_1 \mapsto y, in_2 \mapsto y, out_2 \mapsto z] \end{aligned}$$

Now combine the two parts

$$\begin{aligned} &\forall \langle x, z \rangle \in f_2 \circ f_1 \; \exists y \in D^{\mathfrak{M}} \\ &\mathfrak{M} \models ((\phi_1 \Rightarrow \psi_1) \wedge (\phi_2 \Rightarrow \psi_2))[in_1 \mapsto x, out_1 \mapsto y, in_2 \mapsto y, out_2 \mapsto z] \end{aligned}$$

Then introduce a new variable tmp to replace out_1 and in_2

$$\begin{aligned} \forall \langle x, z \rangle \in f_2 \circ f_1 \; \exists y \in D^{\mathfrak{M}} \; \mathfrak{M} \models ((\phi_1 \Rightarrow \psi_1[out_1 \leftarrow tmp]) \\ \wedge(\phi_2[in_2 \leftarrow tmp] \Rightarrow \psi_2[in_2 \leftarrow tmp]))[in_1 \mapsto x, tmp \mapsto y, out_2 \mapsto z] \end{aligned} \tag{7.30}$$

And finally we have

$$\begin{aligned} \forall \langle x, z \rangle \in f_2 \circ f_1 \; \mathfrak{M} \models (\exists tmp \; (\phi_1 \Rightarrow \psi_1)[out_1 \leftarrow tmp] \\ \wedge(\phi_2 \Rightarrow \psi_2)[in_2 \leftarrow tmp])[in_1 \mapsto x, out_2 \mapsto z] \end{aligned}$$

which is (7.23).

□

7.6 Queries

Let $\mathcal{R}$ be our repository. A query q for $\mathcal{R}$ is a quantifier-free service specification in $S_f^{\mathfrak{L}}$ (see definition 15 for the meaning of $S_f^{\mathfrak{L}}$). A query is submitted to a service repository in order to find possible matches. Theorem 3 states that if we have a specification that matches a query, all services that satisfy the specification also satisfy the query. Our matching algorithm introduced further down will follow this idea.

7.6.1 Querying Results

Service consumers submit queries to a service repository to retrieve matching service specifications so that they can in turn find actual services that

implement these service specifications. Therefore the result of submitting a query to a service repository should contain service specifications that directly match the query, as well as chains of service specifications such that the composition of each specification in a chain in the given order matches the query.

Definition 20. *Let $\mathcal{R}$ be a service repository. With $[s_1, s_2, ..., s_n]_{\circ_\mathcal{T}}$ we describe a sequence of specifications in $S^{\mathfrak{L}}_{\mathcal{R}}$ such that $(s_n \circ_\mathcal{T} (... \circ_\mathcal{T} (s_2 \circ_\mathcal{T} s_1)))$ holds. We write $c :: s_{n+1}$ for the new sequence $[s_1, s_2, ..., s_n, s_{n+1}]_{\circ_\mathcal{T}}$. The function $Compose([s_1, s_2, ..., s_n]_{\circ_\mathcal{T}})$ sequentially composes all specifications in the sequence into the specification $(s_n \circ_\mathcal{T} (... \circ_\mathcal{T} (s_2 \circ_\mathcal{T} s_1)))$, whereby $Compose([s]) = s$.*

Definition 21 (Querying Results). *Let $\mathcal{R}$ be a service repository. The result of submitting a query q to $\mathcal{R}$ is a set of sequences of composable specifications in $S^{\mathfrak{L}}_{\mathcal{R}}$. For each sequence c in the result set, it must hold that $Compose(c) \sqsubseteq_\mathcal{T} q$.*

Theorem 3 and 4 assure us that if for any of the returned sequences we assemble a new service by combining services in the contained specifications, that aggregate service will satisfy our query.

Service consumers then use metadata contained in the matching specifications to get links to and invoke the actual service implementations hosted by service providers. The exact sequence of interactions with a service repository and actual services hosted by service providers after matching is beyond the scope of this thesis, and thus will not be discussed here.

7.6.2 Matching Algorithm

Upon receiving a query q , the repository $\mathcal{R}$ will first select service specifications that directly match the query. Thereafter we will try to find all sequences of specifications such that their composition matches the query. An algorithm is provided below.

Algorithm 1 Matching Algorithm

Base := $\{[s] \mid s \in S^{\mathfrak{C}}_{\mathcal{R}}\}$

Result := $\emptyset$

while (predefined threshold not reached) **do**

Result := Result $\cup \{c \mid (c \in \text{Base}) \wedge (\text{Compose}(c) \sqsubseteq_{\mathcal{T}} q)\}$

Base := $\{c :: s \mid (c \in \text{Base}) \wedge (s \in S^{\mathfrak{C}}_{\mathcal{R}}) \wedge (s \circ_{\mathcal{T}} \text{Compose}(c))\}$

end while

return Result

A predefined threshold should be checked during each iteration of the loop. For example, we can restrict that the maximum length of a composition sequence to be no longer than three. Other constraints based on different criteria can be utilized as the repository designers see fit.

The efficiency of the matching algorithm is not a concern in this thesis. Complexity and potential optimizations will be dealt with in future works.

7.7 Decidability

We use a first-order language to formalize our repository. In order to query the repository, we repeatedly check if a particular specification matches another one. To accomplish this, we need to verify that we have proofs for sentences of the form (7.1) and (7.2). The question arises if there is an effective way to prove a sentence from a theory. We introduce the notion of *decidability* here.

Definition 22 (Decidability). *Let $\mathcal{T}$ be a theory (a set of sentences) in a first-order language $\mathfrak{L}$. A formula $\psi \in \mathfrak{L}$ is decidable if there is an effective method to determine if $\mathcal{T} \vdash \psi$.*

First-order sentences in general are not decidable. [21] proved that a fragment of first-order logic formulas, called the Bernays-Schönfinkel-Ramsey (BSR) class of formulas, is decidable. Formulas in BSR class are those prefixed by zero or more existential quantifiers followed by zero or more universal quantifiers ($\exists^*\forall^*$) when written in prenex normal form without any function symbols or equality.

The algorithm from the previous section relies on our ability to establish the truth of certain sentences (when performing matching). With the following proposition we show that this task is indeed decidable.

Theorem 5. *For a given repository $\mathcal{R}$ let (ϕ_q, ψ_q) be a query and let $c = [s_1, s_2, ..., s_n]_{\circ_\mathcal{T}}$. Then Compose$(c) \sqsubseteq_\mathcal{T} (\phi_q, \psi_q)$ is decidable.*

Proof. The proof is by induction on the length of c. Recall that $s_1, s_2, ..., s_n, \phi_q$, and ψ_q are all quantifier-free formulas.

BASE CASE: $c = [(\phi, \psi)]$. To verify that $(\phi, \psi) \sqsubseteq_{\mathcal{T}} (\phi_q, \psi_q)$ we need to check (7.1) and (7.2):

$$\mathcal{T} \vdash \forall in\ (\phi_q \Rightarrow \phi)$$
$$\mathcal{T} \vdash \forall in\ \forall out\ (\phi_q \wedge \psi \Rightarrow \psi_q).$$

Since $\phi, \psi, \phi_q, \psi_q$ are all quantifier-free, the formulas are in BSR class. Hence the problem is decidable.

INDUCTIVE STEP: Let $c = [(\phi_1, \psi_1), (\phi_2, \psi_2), ..., (\phi_n, \psi_n)]_{\circ_{\mathcal{T}}}$ and let $(\phi, \psi) = \text{Compose}([(\phi_1, \psi_1), ..., (\phi_{n-1}, \psi_{n-1})]_{\circ_{\mathcal{T}}})$. By inductive hypothesis $(\phi, \psi) \sqsubseteq_{\mathcal{T}} (\phi_q, \psi_q)$ is decidable, therefore $\forall in\ (\phi_q \Rightarrow \phi)$ and $\forall in\ \forall out\ (\phi_q \wedge \psi \Rightarrow \psi_q)$ are decidable. We have to show that $(\phi_n, \psi_n) \circ_{\mathcal{T}} (\phi, \psi) \sqsubseteq_{\mathcal{T}} (\phi_q, \psi_q)$ is decidable. Now, we must be able to determine if

$$(\phi_n, \psi_n) \circ_{\mathcal{T}} (\phi, \psi) = (\phi, \exists tmp(\psi[out \leftarrow tmp] \wedge \psi_n[in \leftarrow tmp]))$$

To accomplish this, we need to verify

$$\mathcal{T} \vdash \forall in(\phi_q \Rightarrow \phi)$$

which is decidable by the inductive hypothesis, and

$$\mathcal{T} \vdash \forall in \forall out(\phi_q \wedge \exists tmp(\psi[out \leftarrow tmp] \wedge \psi_n[in \leftarrow tmp]) \Rightarrow \psi_q)$$

which is equivalent to

$$\mathcal{T} \vdash \forall in \forall out(\exists tmp(\phi_q \wedge \psi[out \leftarrow tmp] \wedge \psi_n[in \leftarrow tmp]) \Rightarrow \psi_q)$$

since tmp does not appear in ϕ_q, which is again equivalent to

$$\mathcal{T} \vdash \forall in \forall out \forall tmp(\phi_q \wedge \psi[out \leftarrow tmp] \wedge \psi_n[in \leftarrow tmp] \Rightarrow \psi_q)$$

by the fact that $\mathcal{T} \vdash (\exists x P(x)) \Rightarrow Q$ is equivalent to $\mathcal{T} \vdash \forall x(P(x) \Rightarrow Q)$ provided that x does not appear free in Q.

We now move $\forall tmp$ to the beginning so we have

$$\mathcal{T} \vdash \forall tmp \forall in \forall out(\phi_q \wedge \psi[out \leftarrow tmp] \wedge \psi_n[in \leftarrow tmp] \Rightarrow \psi_q)$$

which follows from

$$\mathcal{T} \vdash \forall tmp \forall in \forall out(\phi_q \wedge \psi[out \leftarrow tmp] \Rightarrow \psi_q)$$

which is decidable by the inductive hypothesis, and

$$\mathcal{T} \vdash \forall tmp \forall in \forall out(\psi_n[in \leftarrow tmp] \Rightarrow \psi_q)$$

which is also decidable since it is already in BSR class (both ψ_n and ψ_q are quantifier-free.

Therefore Compose$(c) \sqsubseteq_{\mathcal{T}} (\phi_q, \psi_q)$ is decidable.

□

Chapter 8

Summary

In this thesis we have shown an approach to organize services in a centralized location called a service repository in the hope of simplifying the process of publishing, discovering and reusing existing services offered by service providers. We have argued the rationale of adopting such service repositories and their benefits. We have presented the various components needed to build a service repository. The detailed steps to design a sample service repository in the domain of flight search have been discussed intensively to show many of the tradeoffs and compromises being considered when designing a service repository. We have also formalized each component of a service repository using a language based on first-order logic.

8.1 Future Work

What we have presented so far in this thesis is just the initial step towards the ultimate goal of designing a feature-rich system to describe services that

would allow advanced querying and automatic composing existing services to produce new ones to meet the demand of service consumers.

Nevertheless, there are many severe limitations in our current design and lots of necessary pieces are missing to allow practical usage. We will briefly discuss many of our concerns and possible directions for future work to extend our current approach.

8.1.1 Hierarchy of Data Types

In our original vision of a service repository we have thought about the hierarchy of data types in the hope that the subtyping relationship of existing data types might be beneficiary to provide additional information about services, and that information could possibly direct querying and composition of services. Later when we tried to actually design a service repository we skipped this part to keep it really simple so we could have a better understanding of service repositories themselves. A future direction would be to consider how to exploit the information available from the hierarchy of data types to infer possible matches of service specifications and compositions.

8.1.2 Generic Predicates

In the current design of service repositories, a predicate called flightFromAirport that asserts the relationship that a FLIGHT departs from an AIRPORT is treated completely different from a similar predicate called flightFromCity that asserts the relationship that a FLIGHT departs from a CITY, even though

both predicates describe conceptually the same idea. This is primarily due to the limitation of the type system. As a consequence, the number of the predicates involved in a service repository is rather large. This makes learning, using, and reasoning about predicates much more challenging.

We envision a future direction to extend the work being done is to merge a family of predicates that assert the same property over different data types into one generic predicate. Instead of having two predicates, flightFromAirport(FLIGHT, AIRPORT) and flightFromCity(FLIGHT, CITY), we would like to combine them into a single predicate, flightFrom(FLIGHT, ?), which expresses the idea that the FLIGHT departs from some place as specified in the second argument. An even more aggressive approach would be to simply design a generic predicate called from(?, ?) which asserts that the first argument leaves from the second argument. Such a design might require a major overhaul of the underlying system.

8.1.3 Services with Side Effects

So far we have assumed the services being described have no side effects. In reality this is a very limiting assumption. Many services exist for the sole purpose of generating side effects. Even services that can be made side-effect free might be implemented to have side effects for various reasons such as efficiency or convenience. Therefore it is absolutely necessary to investigate methods to describe services with side effects.

8.1.4 Ranking and Metadata of Services

We have expressed at the beginning of the thesis that we expect services come with metadata other than its signature and semantic behaviors, such as cost, reliability, etc, so that service consumers can compare and choose when multiple services meet a query simultaneously. This is a very practical concern in real world, even though it has less effect on the design of service repositories. The reason we ignored this aspect in this thesis is that we believe it is orthogonal to the designing and functioning of service repositories in general. Furthermore, we believe the utilization of these metadata would differ greatly in different domains as well as different usage patterns: service consumers in mission-critical domains might be much more concerned about reliability of services, while those in highly-competitive markets might be more price-sensitive. We imagine there would be various different ways to utilize these metadata on a per-domain basis.

8.1.5 Automated Verification of Pre-/Post-Conditions

The language we designed to describe the functionalities of services is only used in the service repository as part of service specifications. It would be greatly helpful if there were a way to automatically verify if a service indeed matches its pre- and post-condition. Obviously it is impractical to test every combination of input and output of a service and see if the values match the assertions in the pre-condition and the post-condition. Nevertheless it would cover a lot of potential bugs for the purpose of unit testing. Therefore

a future direction to make a service repository more useful in real world is to provide standardized unit test packages in different target languages which are extracted from the pre-conditions and post-conditions available in the repository.

Bibliography

[1] S. Agarwal and R. Studer. Automatic matchmaking of web services. In *Proceedings of the IEEE International Conference on Web Services*, pages 45–54, 2006.

[2] F. Arbab. Abstract behavior types: A foundation model for components and their composition. In *Science of Computer Programming*, pages 33–70. Springer Verlag, 2003.

[3] A. Bailly, M. Clerbout, and I. Simplot-Ryl. Component composition preserving behavioral contracts based on communication traces. *Theoretical Computer Science*, 363(2):108–123, 2006.

[4] J. Barwise and J. Etchemendy. *Language, Proof and Logic*. CSLI Publications, Stanford University, 2002.

[5] M. Belguidoum and F. Dagnat. Formalization of component substitutability. *Electronic Notes in Theoretical Computer Science*, 215:75–92, 2008.

[6] M. Broy, I. H. Krüger, and M. Meisinger. A formal model of services. *ACM on Software Engineering Methodology*, 16(1):5, 2007.

[7] P. P. W. Chan and M. R. Lyu. Dynamic web service composition: A new approach in building reliable web service. In *Proceedings of the International Conference on Advanced Information Networking and Applications*, pages 20–25, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[8] W. W. W. Consortium. Web services description language. Website, March 2001. Retrieved from `http://www.w3.org/TR/wsdl` on October 12th 2010.

[9] I. Elgedawy. A conceptual framework for web services semantic discovery. In *On The Move to Meaningful Internet Systems*, volume 2889 of

Lecture Notes in Computer Science, pages 1004–1016. Springer Berlin / Heidelberg, 2003.

[10] A. Friesen and E. Börger. A high-level specification for semantic web service discovery services. In *Workshop Proceedings of the Sixth International Conference on Web Engineering*, page 16, New York, NY, USA, 2006. ACM.

[11] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. *SIGPLAN Notices*, 25(10):169–180, 1990.

[12] D. Hemer. A formal approach to component adaptation and composition. In *Proceedings of the Twenty-eighth Australasian Conference on Computer Science*, pages 259–266, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[13] J. Hoffmann, I. Weber, J. Scicluna, T. Kaczmarek, and A. Ankolekar. Combining scalability and expressivity in the automatic composition of semantic web services. In *ICWE '08: Proceedings of the 2008 Eighth International Conference on Web Engineering*, pages 98–107, Washington, DC, USA, 2008. IEEE Computer Society.

[14] S. C. Kleene. *Mathematical Logic*. Wiley, New York, 1967.

[15] F. Lécué and A. Léger. A formal model for web service composition. In *Proceeding of the 2006 Conference on Leading the Web in Concurrent Engineering*, pages 37–46, Amsterdam, The Netherlands, The Netherlands, 2006. IOS Press.

[16] L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *Proceedings of the 12th International Conference on the World Wide Web*, pages 331–339. ACM Press, 2003.

[17] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16:1811–1841, 1994.

[18] Object Management Group. Corba 3.0 - IDL syntax and semantics chapter, July 2002. Retrieved from `http://www.omg.org/cgi-bin/doc?formal/02-06-07.pdf` on October 12th, 2010.

[19] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Press, Mountain View, California, 2007.

[20] T. Pilioura and A. Tsalgatidou. Unified publication and discovery of semantic web services. *ACM Transactions on the Web*, 3(3):1–44, 2009.

[21] F. P. Ramsey. On a problem in formal logic. In *Proceedings of the London Mathematical Society*, pages 264–286. London Mathematical Society, 1930.

[22] S. Willmott, H. Ronsdorf, and K. H. Krempels. Publish and search versus registries for semantic web service discovery. In *International Conference on Web Intelligence*, pages 491–494, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[23] A. M. Zaremski and J. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.

Appendix A

List of Data Types

1. Flight
2. Itinerary
3. Airline
4. Alliance
5. Airport
6. City
7. Time
8. Interval

Appendix B

List of Predicates

1. validAirport(AIRPORT)
2. allItinerariesDepartFromAirport([ITINERARY], AIRPORT)
3. allItinerariesArriveAtAirport([ITINERARY], AIRPORT)
4. allItinerariesDepartureTimeWithin([ITINERARY], INTERVAL)
5. completeListOfOnewayItineraries([ITINERARY], (AIRPORT, AIRPORT, INTERVAL))
6. completeListOfDirectItineraries([ITINERARY], (AIRPORT, AIRPORT, INTERVAL))
7. completeListOfNonstopItineraries([ITINERARY], (AIRPORT, AIRPORT, INTERVAL))
8. validMultidestSequence([(AIRPORT, AIRPORT, INTERVAL)])
9. completeListOfMultidestItineraries([ITINERARY], (AIRPORT, AIRPORT, INTERVAL))
10. allMutlidestItinerariesSatisfy([ITINERARY], [(AIRPORT, AIRPORT, INTERVAL)])
11. allItinerariesOperatedByAirlines([ITINERARY], [AIRLINE])
12. validAirlines([AIRLINE])
13. allItinerariesOperatedByAirlines([ITINERARY], [AIRLINE])

14. completeListOfOnewayItinerariesByAirlines([ITINERARY], (AIRPORT, AIRPORT, INTERVAL, [AIRLINE]))
15. completeListOfDirectItinerariesByAirlines([ITINERARY], (AIRPORT, AIRPORT, INTERVAL, [AIRLINE]))
16. completeListOfNonstopItinerariesByAirlines([ITINERARY], (AIRPORT, AIRPORT, INTERVAL, [AIRLINE]))
17. completeListOfMultidestItinerariesByAirlines([ITINERARY], (AIRPORT, AIRPORT, INTERVAL, [AIRLINE]))
18. validFlight(FLIGHT)
19. flightToAirport(FLIGHT, AIRPORT)
20. flightToCity(FLIGHT, CITY)
21. validCity(CITY)
22. cityWithAirport(CITY)
23. allAirportsInCity([AIRPORT], CITY)
24. airportInCity(AIRPORT, CITY)
25. allAirportsInCity([AIRPORT], CITY)
26. validAirline(AIRLINE)
27. airlineInAlliance(AIRLINE, ALLIANCE)
28. allAirlinesInAllince([AIRLINE], ALLIANCE)
29. allValidAirports([AIRPORT])
30. sameInterval(INTERVAL, INTERVAL)

Appendix C

List of Specifications

Specification C.1 searchOnewayItineraries

(AIRPORT, AIRPORT, INTERVAL) $\rightarrow$ [ITINERARY]

ϕ: validAirport(in(0)) $\wedge$ validAirport(in(1))

ψ: allItinerariesDepartFromAirport(out, in(0)) $\wedge$
allItinerariesArriveAtAirport(out, in(1)) $\wedge$
allItinerariesDepartureTimeWithin(out, in(2)) $\wedge$
completeListOfOnewayItineraries(out, in)

Specification C.2 searchDirectItineraries

(AIRPORT, AIRPORT, INTERVAL) $\rightarrow$ [ITINERARY]

ϕ: validAirport(in(0)) $\wedge$ validAirpoort(in(1))

ψ: allItinerariesDepartFromAirport(out, in(0)) $\wedge$
allItinerariesArriveAtAirport(out, in(1)) $\wedge$
allItinerariesDepartureTimeWithin(out, in(2)) $\wedge$
completeListOfDirectItineraries(out, in)

Specification C.3 searchNonstopItineraries

(AIRPORT, AIRPORT, INTERVAL) $\rightarrow$ [ITINERARY]

ϕ: validAirport(in(0)) $\wedge$ validAirport(in(1))

ψ: allItinerariesDepartFromAirport(out, in(0)) $\wedge$
allItinerariesArriveAtAirport(out, in(1)) $\wedge$
allItinerariesDepartureTimeWithin(out, in(2)) $\wedge$
completeListOfNonstopItineraries(out, in)

Specification C.4 searchMultidestItineraries

[(AIRPORT, AIRPORT, INTERVAL)] $\rightarrow$ [ITINERARY]

ϕ: validMultidestSequence(in)

ψ: completeListOfMultidestItineraries(out, in) $\wedge$
allMultidestItinerariesSatisfy(out, in)

Specification C.5 searchOnewayItinerariesByAirlines

(AIRPORT, AIRPORT, INTERVAL, [AIRLINE]) → [ITINERARY]

ϕ: validAirport(in(0)) ∧ validAirport(in(1)) ∧ validAirlines(in(3))

ψ: allItinerariesDepartFromAirport(out, in(0)) ∧
allItinerariesArriveAtAirport(out, in(1)) ∧
allItinerariesDepartureTimeWithin(out, in(2)) ∧
allItinerariesOperatedByAirlines(out, in(3)) ∧
completeListOfOnewayItinerariesByAirlines(out, in)

Specification C.6 searchDirectItinerariesByAirlines

(AIRPORT, AIRPORT, INTERVAL, [AIRLINE]) → [ITINERARY]

ϕ: validAirport(in(0)) ∧ validAirpoort(in(1)) ∧ validAirlines(in(3))

ψ: allItinerariesDepartFromAirport(out, in(0)) ∧
allItinerariesArriveAtAirport(out, in(1)) ∧
allItinerariesDepartureTimeWithin(out, in(2)) ∧
allItinerariesOperatedByAirlines(out, in(3)) ∧
completeListOfDirectItinerariesByAirlines(out, in)

Specification C.7 searchNonstopItinerariesByAirlines

(AIRPORT, AIRPORT, INTERVAL, [AIRLINE]) → [ITINERARY]

ϕ: validAirport(in(0)) ∧ validAirport(in(1)) ∧ validAirlines(in(3))

ψ: allItinerariesDepartFromAirport(out, in(0)) ∧
allItinerariesArriveAtAirport(out, in(1)) ∧
allItinerariesDepartureTimeWithin(out, in(2)) ∧
allItinerariesOperatedByAirlines(out, in(3)) ∧
completeListOfNonstopItinerariesByAirlines(out, in)

Specification C.8 searchMultidestItinerariesByAirlines

[(AIRPORT, AIRPORT, INTERVAL, [AIRLINE])] → [ITINERARY]

ϕ: validMultidestSequence(in) ∧ validAirlines(in(3))

ψ: allMultidestItinerariesSatisfy(out, in) ∧
allItinerariesOperatedByAirlines(out, in(3)) ∧
completeListOfMultidestItinerariesByAirlines(out, in)

Specification C.9 getDestinationAirport

FLIGHT → AIRPORT

ϕ: validFlight(in)

ψ: flightToAirport(in, out)

Specification C.10 getParentCity

AIRPORT → CITY

ϕ: validAirport(in)

ψ: airportInCity(in, out)

Specification C.11 getDestinationCity

FLIGHT → CITY

ϕ: validFlight(in)

ψ: flightToCity(in, out)

Specification C.12 lookupAirports

CITY → [AIRPORT]

ϕ: validCity(in) $\wedge$ cityWithAirport(in)

ψ: allValidAirports(out) $\wedge$ allAirportsInCity(out, in)

Specification C.13 lookupCity

AIRPORT → CITY

ϕ: validAirport(in)

ψ: validCity(out) $\wedge$ airportInCity(in, out)

Specification C.14 lookupAirlineAlliance

AIRLINE $\rightarrow$ ALLIANCE

ϕ: validAirline(in)

ψ: airlineInAlliance(in, out)

Specification C.15 lookupMemberAirlines

ALLIANCE $\rightarrow$ [AIRLINE]

ϕ: validAlliance(in)

ψ: allAirlinesInAllince(out, in)