Patterns for Service Composition

Matthias Tilsner\textsuperscript{1,2}, Adrian Fiech\textsuperscript{1}, Guangyao Zhan\textsuperscript{1}, Thomas Specht\textsuperscript{2}
\textsuperscript{1}Memorial University of Newfoundland, St. John’s, Canada
\{matthias.tilsner,afietch.gzahn\}@mun.ca
\textsuperscript{2}Hochschule Mannheim, Mannheim, Germany
\{m.tilsner,t.specht\}@hs-mannheim.de

ABSTRACT
The discovery of suitable services is a crucial, but challenging activity during service-oriented engineering. While in many scenarios a single service will satisfy the user’s exigent needs, there are cases where a combination of services might be appropriate. In this work-in-progress paper we identify several composition patterns that assist in the discovery of appropriate services. We outline a pattern based algorithm for service discovery and formalize the solution set.

Keywords: Service-Oriented Architecture, Service Composition, Formal Models, Theory, Software Engineering

1. INTRODUCTION

Over the last few years, service-oriented architectures garnered significant attention amongst software engineers and researchers. The concept of assembling complex applications from smaller, goal-oriented parts promises reduction of engineering effort and, due to reusability, increased return of investment. While we witness the proliferation of simple services (e.g. mashups), the discovery of more complex services and their semantic description are still a sizable challenge.

In our previous work ([4], [9]) we propose an approach to web service repository design that is domain specific and leaves the control over domain ontology firmly in the hands of the designers. In fact, we depart from the common SOA paradigm, where service providers proactively register their services and where service registries accept all submitted services. In our approach, repositories are closed collections of well chosen service specifications. Each specification has an attached (open) collection of service implementations. Service providers can only publish their services in the repository, if such services implement one of the specifications contained there. Concrete services are maintained by service providers on their own servers. The repository assumes no responsibility to host those services. Service specifications contain information about where to locate the corresponding service, how to communicate with the service, the cost of the service, and other relevant information.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2011 ACM 978-1-4503-0626-3/11/05 ...$10.00.

When a service consumer is in need of a particular service, he/she either browses the catalog of existing service specifications or submits the requirement as a query. In such case the repository returns a list of matching service specifications. Often though, there will be no service specification that directly satisfies consumer’s demands. However, it is likely that the desired functionality can be provided by a combination of services that are executed in a specific order. Here, some additional implementation effort on part of the client might be required.

This paper discusses preliminary work that aims to identify patterns of web service composition. We begin with section 2 and formally introduce service repositories. In section 3 we discuss few common patterns of service composition. Section 4 presents our initial approach to formal treatment of solution sets and sketches a pattern-based algorithm for service discovery. A discussion of existing literature follows in section 5. We conclude with section 6.

2. DEFINING REPOSITORIES

This section summarizes our understanding of repositories as described in [9] and [4]. An introduction to the underlying logical concepts can be found in [3].

A service repository is primary a collection of service specifications. Each specification has a corresponding set of services offered by different service providers. Although these actual services will eventually be invoked by service consumers, they play a rather secondary role in our discussions. Notwithstanding, we begin our formal treatment with the definition of a service.

DEFINITION 1 (SERVICE). A service $f$ from input type $\Pi$ to output type $\Omega$ is a binary relation:

$$f \subseteq \Pi \times \Omega$$

Service specifications provide us with information about services’ signature (input and output types) and the pre- and post-conditions that a service must satisfy.

DEFINITION 2 (SPECIFICATION). A service specification is a tuple $(\Pi, \Omega, \phi)$, where $\Pi$ is the input data type and $\Omega$ is the output data type. Pre-condition $\phi$ (with free variable input) and post-condition $\psi$ (with free variables input and output) are well-formed formulas in a first-order language $\mathcal{L}$. $\phi$ asserts some properties of its input value and $\psi$ asserts properties of its output value with respect to its input value.

Given a first-order structure $\mathcal{M} = (\mathcal{D}^{\Pi}, \mathcal{D}^{\Omega}, \mathcal{N}^{\Omega})$, for the chosen language $\mathcal{L}$, each specification defines a set of services.
The domain of discourse $D^n$ is the union of objects in the repository, and $p^n$ provides interpretation to the predicates in $\mathcal{L}$ and $N^n$ assigns meaning to the constants in $\mathcal{L}$. We write $\mathfrak{M} \models \phi(g)$ if a variable assignment $g$ satisfies a formula $\phi$ in $\mathfrak{M}$.

**Definition 3 (Specification Meaning).**

$$\{ \{ \Pi, \Omega, \phi, \psi \} \mid \{ \pi \subseteq \Pi \times \Omega \mid \forall (x, y) \in \pi \} \models 2 \models (\phi \Rightarrow \psi)\}$$

We now define the service repository itself.

**Definition 4 (Repository).** A repository is a tuple $R = (\mathcal{T}, \mathcal{L}, \mathcal{M}, \text{SPEC}_R, \text{serv}_R)$, where $\mathcal{T}$ is the set of data types, $\mathcal{L}$ is a first-order language, $\mathcal{M}$ is a set of axioms over $\mathcal{L}$, and $\mathcal{M}$ is a first-order structure for $\mathcal{L}$ and a model for $\mathcal{T}$. SPEC$_R$ is a collection of service specifications and serv$_R$ is a mapping SPEC$_R$ $\rightarrow$ $\mathcal{P}$($\text{SERV}_R$). Here $\text{SERV}_R$ = $\bigcup_{\text{spec} \in \text{SPEC}_R}$ $\text{SERV}_R$ is the set of concrete services.

Service repositories are domain-specific and the selection of the appropriate first-order language, the set of axioms, first-order structure and admissible types is part of the repository design.

Developers submit queries (specifications) to the repository and receive a set of specifications that ‘match’ (or refine) the query. (Conceptually, if $\text{spec}_1$ refines $\text{spec}_2$ then all services that satisfy $\text{spec}_2$ also satisfy $\text{spec}_1$).

**Definition 5 (Matching).** Let $\text{spec}_1 = (\Pi, \Omega, \phi_1, \psi_1)$ and $\text{spec}_2 = (\Pi, \Omega, \phi_2, \psi_2)$ be two service specifications with the same type signature. We say that $\text{spec}_1$ matches $\text{spec}_2$ ($\text{spec}_1 \sqsubseteq \text{spec}_2$) if and only if

$$\forall \text{input} (\phi_2 \Rightarrow \phi_1) \quad (1)$$

$$\forall \text{input} \forall \text{output} (\phi_2 \land \psi_1 \Rightarrow \psi_2) \quad (2)$$

Justification for this definition is provided in [9].

3. SERVICE ASSEMBLY

In [9] we presented a repository for flight-planning services. This repository contains specifications for diverse services related to flights, airports and travel itineraries in general. Here we briefly describe this repository and use it as a leading example for our discussions about service composition.

In the sample repository we identified CITY, AIRPORT, AIRLINE and FLIGHT amongst our key domain types. Furthermore, we introduced three type constructors LIST, MAP and TUPLE.

The first-order language in our repository contains a number of predicates. Here we present only a small selection: validAirport(a), airportInCity(a, c), flightFromAirport(f, a), flightFromCity(f, c). The theory $\mathcal{T}$ contains, amongst others, the axioms: $\forall f \forall a \forall c$ (flightFromAirport(f, a) $\land$ airportInCity(a, c) $\Rightarrow$ flightFromCity(f, c)).

The model $\mathfrak{M}$ assigns meaning to predicates, e.g. $\text{airportInCity}(\text{airport}, \text{city})$ returns true if $\text{airport}$ is located in the given city. Also, $\text{flightFromAirport}(\text{flight}, \text{airport})$ returns true, if $\text{flight}$ indeed departs from the provided $\text{airport}$.

The set $\text{SPEC}_R$ contains carefully selected specifications, e.g. $\text{getFlightFromAirport} = \{ \text{AIRPORT}, \text{FLIGHT}, \phi, \psi \}$, where $\phi = \text{validAirport}(\text{input})$ and $\psi = \text{getFlightFromAirport}(\text{output}, \text{input})$. serv$_R$ maps then getFlightFromAirport to the set of registered services that satisfy this specification, e.g. a service that returns the next available flight from the provided airport and/or a service that returns the last flight of the day from the selected airport. Other specifications included in the repository are: getAirportForCity, getOperatingAirline, getDepartureAirport, getArrivalAirport, getFlightsBetweenAirports.

The list here is not exhaustive. We trust that the selected specification names convey their meaning accurately and invite the reader to write the details.

A developer of a flight reservation portal could assemble such web application by utilizing several services from our repository. After identifying the desired functionality, the developer sends a specification query to the repository. The result is a solution set with ‘matching’ specifications and instructions about selection of services. These services, when combined, will perform the desired task.

We examine now how such solutions could look like. This in turn leads us to the identification of certain service composition patterns.

**Atomic Pattern.**

The simplest result would be the specification query itself, provided such specification is contained in the repository. If a service consumer sends the query getFlightFromAirport to our repository, then the result should certainly contain the specification getFlightFromAirport, as getFlightFromAirport $\epsilon$ SPEC$_R$. The instructions would advise the user to simply select one of the services in serv$_R$(getFlightFromAirport). This leads us to the atomic pattern represented in the Figure 1. In a nutshell, specifications (or more precise services described by specifications) can be viewed as black-box components that take some input data and return some output. Figure 1 displays such specification where the circles represent input and output data and the rectangle stands for the service executed.

**Sequential Pattern.**

For many queries, however, there might be no single specification that ‘matches’ the request. A combination of services that implement several specifications might do the trick though. Imagine that the application assembler is searching for a service that takes a city and returns a flight departing that city (to somewhere). We leave it up to the reader to write the specification of such service. Let’s assume now that the repository does not contain a direct match for such specification. On the other hand, the repository contains the following two specifications: one that describes services that take a city and return corresponding airport; the second one describes services that take an airport and return some flight from this airport. It is rather straightforward to accomplish our goal by utilizing services belonging to these two specifications. Choose a service in the first specification, invoke it on a city and pass the result to a service associated with the second specification. Figure 2 depicts the composition.

**Multiplication Pattern.**

For another scenario consider a developer that queries for service specifications that allow him to retrieve the operating airlines for a list of flights (e.g. Air Canada may operate
the flight LH 234, if Lufthansa and Air Canada have code sharing agreement for this flight). The repository, however, only contains a specification \textit{airlineOfFlight} that returns the airline of a single flight. While this specification by itself does not satisfy the query, it is easy to see how services associated with it can be helpful. The result to the query could contain the specification \textit{airlineOfFlight} and instructions to choose some service from \textit{serve\_q}(\textit{airlineOfFlight}), apply it to every element of the input list and then to assemble all the sub-results to the output list. Figure 3 illustrates the composition.

**Parallel Pattern.**

Similar to \textit{Multiplication Pattern} is another scenario. Consider a request for a service that takes a flight and returns the tuple with the flight’s departure and arrival airports. Now assume that our repository only contains specifications \textit{getDepartureAirport} and \textit{getArrivalAirport}, but not the desired specification \textit{getConnectedAirports}. We still can help our developer. The desired behavior can be easily achieved by executing services that satisfy these two specifications in parallel. We merge then the outputs. Figure 4 presents the composition.

**Combining Patterns.**

Consider now a situation where the developer queries for a service \textit{getAlternativeFlight} that returns a substitute flight between two airports. The repository, however, only contains specification \textit{flightBetweenAirports} and the previously described \textit{getDepartureAirport} and \textit{getArrivalAirport}. While neither of these specifications satisfies the query, it is possible to achieve the desired functionality by invoking \textit{getDepartureAirport} and \textit{getArrivalAirport} on the input flight and then using the result as the input for \textit{getFlightBetweenAirports}. Figure 5 portrays the composition.

Conceptually we have applied the \textit{Parallel Pattern} (darker rectangle) followed by the \textit{Sequential Pattern} (lighter rectangle) with appropriate instructions to apply the pattern. The result, together with the \textit{getFlightBetweenAirports} specification is an ‘input’ to the Sequential Pattern (again with appropriate instructions). The user can rest assured that by following the instructions, he/she will assemble a service that satisfies the query.

The list of patterns identified so far is by far not exhaustive. It should, however, give the reader an insight into specification/service composition.

We envision domain specific service repositories with a finite number of included specifications. Although repository designers will anticipate the most common service requests, they cannot envision all of them. Still, the aim is to satisfy as large number of queries as possible. This is where the patterns come into play.

4. **QUERIES AND SOLUTIONS**

In this section we formally describe the solution set for a given query. Later on we sketch a pattern-based algorithm for service discovery.

As discussed in the previous section, application assemblers submit search queries to the service repository. Such queries are in fact service specifications, which describe the desired behavior of the needed service. Our repository contains a collection of selected specifications and therefore the straightforward result set would include all specifications in the repository that match the submitted specification. A brute force algorithm would simply verify the refinement property for every \textit{spec} \in \textit{SPEC}_R. In [9] we implemented such algorithm (also for sequential composition) in Prolog. What’s left, is the selection of particular service from \textit{serve\_q}(\textit{spec}) for any of the returned specifications. The service consumer will make the choice based on reliability, reputation, performance and/or price.

If no refinement specification is available in \textit{SPEC}_R or if more complex solution alternatives exists, we can build solutions sets based on the composition patterns.

**Definition 6 (Solution Set).** Let \textit{spec} \in \textit{SPEC}_R. We recursively define solutions to queries:

\[
\text{Solution} ::= \text{ATOM}(\text{spec}) \\
| \text{SEQ}(\text{Solution} \times \text{Solution}) \\
| \text{MUL}(\text{Solution}) \\
| \text{PAR}(\text{Solution} \times \text{Solution})
\]

\[
\text{Solution\_Set} ::= \text{SET}(\text{Solution})
\]
Here $ATOM(s)$ represents the atomic solution (consisting of a specification in $SPEC_R$), whereas the other solutions represent application of patterns. The composition in Figure 5 can be described as $SEQ(PAR/ATOM(getDepartureAirport), \ ATOM(getArrivalAirport), ATOM(getFlightBetweenAirports))$.

Intuitively, for each solution $sol$ we have a specification $sol_spec(sol)$ (not necessarily in $SPEC_R$) with associated set of services (which might have to be ‘manually’ crafted from services in $SERV_R$). For example, $spec$ itself is the specification for $ATOM(spec)$, whereas for $SEQ(sol_1, sol_2)$ we have the more complex $sol_spec(sol_1) \circ sol_spec(sol_2)$. (See [9] for details.)

The complex services are constructed using operations defined below.

**Definition 7 (Service operations).** Let $f \subseteq A \times B$, $g \subseteq B \times C$ and $h \subseteq A \times C$ be services. We define the following operations on services:

- $g \circ f \equiv \{(a, c) \in A \times C \mid \exists b \in B : (a, b) \in f \land (b, c) \in g\}$
- $f \odot h \equiv \{(a, b, c) \in A \times (B \times C) \mid (a, b) \in f \land (a, c) \in h\}$
- $map(f) = \{(a_1, \ldots, a_n) \mid \forall 1 \leq i \leq n : (a_i, b_i) \in f\}$

Below we define the join method, which provides us with the set of (assembled) services that correspond to each Solution.

**Definition 8 (Service assembly).** The function $join : Solution \rightarrow \mathcal{P}(SERV)$ is defined as:

- $join(ATOM(spec)) = [spec]$
- $join(SEQ(sol_1, sol_2)) = \cup_{s_1 \in join(sol_1)} \cup_{s_2 \in join(sol_2)} s_2 \circ s_1$
- $join(MUL(sol)) = \cup_{s \in join(sol)} map(s)$
- $join(PAR(sol_1, sol_2)) = \cup_{s_1 \in join(sol_1)} \cup_{s_2 \in join(sol_2)} s_1 \otimes s_2$

The function $join$ can be viewed as a set of instructions on how to build complex services from services registered in $\mathcal{R}$.

In the following we sketch an algorithm that computes a solution set for a query (which is a specification).

```python
1 def findAllSolutions(base: Set[Solution],
2     query: Query): Set[Solution] = {
3     val solutions = EmptySet
4     for (s in base)
5         if (sol_spec(s) matches query)
6             solutions = solutions + s
7     }
8     if (!reachedBreakpoint(base)) {
9         for (s1, s2 in base) {
10             base = base + SEQ(s1, s2)
11                 + MUL(s1) + PAR(s1, s2)
12         }
13     solutions = solutions
14         ++ findAllSolutions(base, query)
15     }
16     return solutions
17 }
18 def atomBase: Set[Solution] = {
19     val base = EmptySet
20     for (spec in SPEC) {
21         base + ATOM(spec)
22     }
23     return base
24 }
25 val allSolutions = findAllSolutions(atomBase(), query)
```

The algorithm first adds to the solution set all specifications in $SPEC_R$ that match the query. Thereafter, solutions based on patterns are constructed recursively. When doing this, the algorithm checks if the associated specification matches the query. The method $reachedBreakpoint(base)$ restricts the depth of recursion. We believe that solutions with more than 3 or 4 pattern levels are impractical.

**Claim 1.** Let $spec \in SPEC$ be a query for the findAllSolutions algorithm. Then for all $sol \in findAllSolutions(_, spec)$, $sol_spec(sol) \subseteq spec$. Also, $join(sol) \subseteq [spec]$.

5. RELATED WORK

The issue of service specification, discovery and matching has been explored by multiple research projects. However, only few examine the topic of complex service composition. Many current approaches (e.g., [7]) assume that any composition is manually designed by a software architect. We, on the other side, strive for an automated solution that allows us to apply the pattern-based algorithm.

One of the earlier publications on composition of software components is [1]. In their work the authors explore the composition of independent components by connecting their input and output channel. While it does provide a solid basis for further thought process, the paper neither discusses more complex composition models, nor is a formal specification language provided.

One way to formally model the composition of services is presented in [2]. The main contribution is a formal system in which the results of composition (also referred to as composite built) are services, and can thus be treated like any atomic service. However, their solution does not discuss substitutability, which we require in order to achieve successful matchmaking. Furthermore, the presented solution supports only sequential service chaining. More complex compositions are not discussed.

Another approach towards specification composition is provided in [5]. There too, services are described with pre- and post-conditions, similar to our approach. The author specifies different combinators that allow users to create combined service specifications. 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.

[3] presents a method for discovery of Web Service compositions. Within multiple iterations the authors examine different options on how to get from the desired input to the desired output. The different paths are then compared as routes. A software component examines the different options and evaluates the best route. However, by only checking how to receive the desired output for a given input, the approach merely processes syntactic information while any semantic meaning is ignored.

A more formal and complex solution is presented in [6]. Using a heuristic approach the authors present an algorithm that identifies suitable service compositions for given user requirements, called Tasks. 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 in sequence yields the target state, described through the effect-component of the task, is a valid candidate. Unfortunately, sequential execution of services is
the only method of composition supported by this approach. [8] provides a model for semantically composing services both in sequence and in parallel. By doing so the authors present one of the few solutions that recognize the benefit of having more complex composition structures. Furthermore, they provide an approach that allows formal description of component semantics. However, their solution does not incorporate other composition patterns.

To the best of our knowledge, no research done so far provides an in-depth discussion of complex service compositions, nor does any provide a complete formal model for describing composition results. We hope to fill this gap with our investigation.

6. DISCUSSION AND FUTURE WORK

The results presented in this paper are still preliminary. We describe small patterns that model atomic structures of complex compositions and show how these can be combined to construct larger systems. The sample of patterns presented here is by far not exhaustive. An in-depth examination of successful service-oriented architectures is required to identify the predominant composition structures. We also expect that extensive interviews with our industry partners (actual service providers) from the COCTAIL project will bring us closer to understand the main service composition patterns.

We believe that the development of formal models for specification and service composition are key aspects of our future research. The semi-formal presentation in this paper is just the beginning.

The crude pattern-based matching algorithm presented here is meant to provide the basis for further investigation. More efficient algorithms must be developed. Case studies with large repositories will be conducted. We are designing domain-specific repositories for a number of business areas. The ‘similarity’ of specifications (vs. exact match) is another direction we are pursuing.

Acknowledgment

This work was partially supported by the German Federal Ministry of Education and Research (BMBF) under the COCTAIL project (grant FKZ 01BS0823) and by the Natural Sciences and Engineering Research Council of Canada (grant OGP0170497).

7. REFERENCES