Synthesis and Partial Execution of Service Processes for Matchmaking [*]

Rajesh Thiagarajan Wolfgang Mayer Markus Stumptner
University of South Australia
Advanced Computing Research Centre
{cisrkt,mayer,mst}@cs.unisa.edu.au

Abstract

Semantic Web technology has been a significant driver of recent research in Service Oriented Architectures to automate tasks such as discovery and reuse of services. However, commercial service providers currently offer their services only as web applications and do not generally offer Semantic Web Service interfaces for querying. In this paper, we consider the synthesis of such web applications into service processes to infer exact capabilities that a service offers. We also present an approach to partially execute services to acquire information required to evaluate specific queries while avoiding permanent effects.

1. Introduction

Semantic Web technology has been a significant driver of recent research with enterprises or end user applications attempting to compose complex interactions from offered services in a variety of application contexts such as brokering, purchasing, supply chain integration, and experiment assembly. Much of the existing research in this domain deals with providing declarative descriptions to services, using these descriptions to automatically discover services for a specific purpose, and automatically compose services to meet goals that cannot be satisfied by individual services.

Most of the existing solutions for automated discovery/composition (Paolucci et al. 2002; Kawamura et al. 2004; Grønmo and Jaeger 2005; Li and Horrocks 2003; Wang and Li 2006; de Bruijn et al. 2005; Patel-Schneider and Horrocks 2007) determine suitability of a service for a purpose by assessing the *conceptual* description of the service. Conceptual descriptions consist of terminological or general information about a service such as 'service that sells airline tickets between Australian cities'. For simplicity, we use natural language to describe requests and profiles. While conceptual information does provide an indication of the functionality of a service, it does not reflect the exact capabilities the service might offer. For example, for a service request like 'service that sell airline tickets between Adelaide and Sydney for less than $200' assessing only the conceptual information of an airline service is insufficient to determine whether the service can meet the request. In contrast, a 'concrete' (or '*instance* level') profile of a service holds information about its exact capabilities that can be used to determine whether it is suitable for a specific request. For example, an airline service might list all available flights. This information is to be used to evaluate whether the service is able to satisfy a specific booking query (like the one above). Existing solutions (such as (Li and Horrocks 2003; Wang and Li 2006)) either ignore the *instance* level details altogether or propose to explicitly model the *instance* level information within the conceptual description. We argue that it is not feasible to predict and model the *instance* level information of a service explicitly under these assumptions. For example, it is not feasible to list all the flights an airline service offers in the *conceptual* description, since available flights may change dynamically and may depend on a number of factors, such as availability. Hence, conceptual profiles typically over-approximate the exact capabilities and may result in a number of false positives.

In our prior work (Thiagarajan and Stumptner 2007; Thiagarajan et al. 2007), a two-stage matchmaking approach to overcome the problem of dealing with *in-*

[*] This work was partially supported by the Australian Research Council (ARC) under grant DP0988961.

stance level details has been presented. In the first stage, a set of *conceptual* profiles that seem to match the request are shortlisted, and subsequently, in the second stage, the *instance* level information about these services are obtained by querying the services directly to assess whether the services indeed meet the requirements. As information at both *conceptual* and *instance* level is used, the matches determined by the two-stage approach supersede those that are derived by pure conceptual matchmaking. The ability to query a service is crucial to realise the two-stage matching scheme. While the existing technologies surrounding the Semantic Web are able to support the creation of such querying facilities, in practice, commercial service providers offer their services only as web applications. It is not common that operations in the application are exposed as web services to directly query them for concrete information. Therefore, from a pragmatic point of view, if such a web application is synthesised as a service process then we may potentially acquire *instance* level information of the service by partially executing the process. For example, major airline companies offer web applications accessible on their websites to search, book and cancel tickets. If such an application is synthesised into a service process then by partially executing the process (for example, complete only the search part) we can acquire *instance* level details to confirm or refute specific queries.

In this paper, we consider the synthesis of web applications into service processes and present a transition selection strategy to control partial execution of a service process to acquire required *instance* level information. Ontologies are used in our approach to disambiguate terms in the requirements specification, *conceptual* profiles, and terminology used in service processes. In our approach, a web application is synthesised as a service process by modelling every operation as a web service call qualified with a set of inputs, outputs, pre- and post-conditions. That is, a web service call in the context of the synthesised process corresponds to a specific operation in the web application. Executing a web service call is analogous to performing the corresponding operation in the web application. The behaviour of a process model is specified using a state chart where every transition represents a web service call. We propose a strategy to search the state chart of a process to determine the set of transitions to be executed so that sufficient *instance* level information about the service is acquired to assess its suitability for a particular purpose.

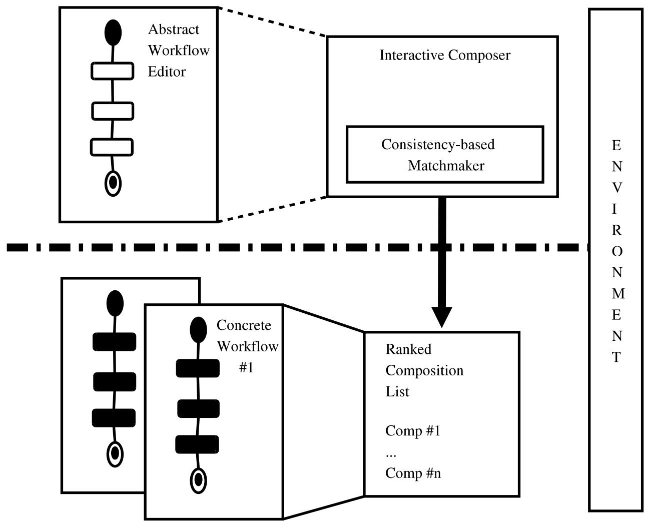


Figure 1. Model Driven Workflow Composition

The modeling approach follows the lines of the Model-Driven Architecture (MDA) paradigm, where platform specific models (PSM - e.g., program code) are largely hidden from view by providing a Platform Independent Model (PIM) that is used to describe the application and is then, if possible automatically, transformed into a PSM for execution. Development and maintenance are mainly performed at the PIM level, resulting in reduced effort compared to explicit PSM level implementation. In our case, processes are specified in terms of high level conceptual models (e.g,. statecharts) and the process synthesis and evaluation process consists of the transformation and interpretation of these high level models. We demonstrate our approach by evaluating the suitability of an airline service process for a travel booking query using our consistency-based matchmaking scheme presented in (Thiagarajan and Stumptner 2007; Thiagarajan et al. 2007).

The rest of the paper is structured as follows. We give a brief introduction to our two state consistency-base matchmaking scheme in Section 2. Synthesis of web applications into service processes is considered in Section 3. In Section 4, we present our strategy to determine best set of state transitions that provide *instance* level details of a service. A review of the existing related work is provided in Section 5 followed by the summary on contributions and future work in Section 6.

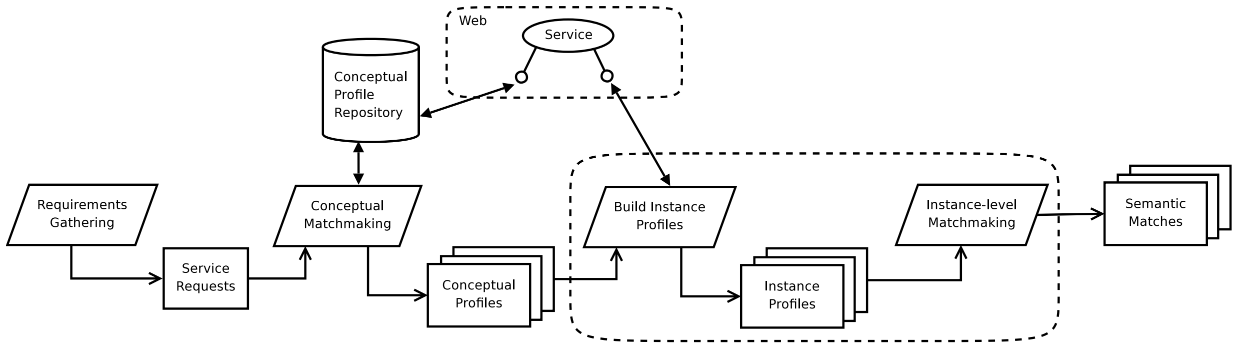


Figure 2. Consistency-based Matchmaking Phases

2. Service Composition with two-stage Matchmaking

In our MDA-based composition approach (Thiagarajan et al. 2007), an abstract workflow describing a user's business processes and abstract requirements in an implementation-independent fashion is refined into a concrete workflow that is suitable for automatically synthesising execution. The composition process is shown in Figure 1. In our approach, UML activity diagrams are extended to allow users to specify the requirements of a composition. The abstract workflow is refined into a concrete workflow such that all the requirements in the abstract workflow are satisfied. The refinement is carried out by our two-staged consistency-based matchmaking scheme.

2.1 Consistency-based Matchmaking

Our consistency-based matchmaking approach (Thiagarajan and Stumptner 2007) consists of two stages (see Figure 2). In the first stage, conceptual profiles that are consistent with the requirements from the request are selected from a repository such as the one shown in Figure 3. In our approach, a *conceptual* profile is a declarative specification that consists of a set of attributes, data types of the attributes, and a set of general constraints. For example, the *conceptual* profile of *BudgetAirlines* service in Figure 3 consists of the following attributes $\{\langle category, SalesCategory\rangle, \ldots, \langle source, City\rangle, \ldots \langle travelDate, Date\rangle\}$, and the following constraints[1] $\{category.oclIsTypeOf(TicketCategory), price \leq 150\}$. The *conceptual* profiles are used to limit the search process to a few likely candidates that seem to match the request. In the second stage, the instance profile of each candidate service identified in the previous step is built by querying the respective services for concrete information and this *instance* information is used to to assess if the service can satisfy the request.

As shown in (Thiagarajan and Stumptner 2007; Thiagarajan et al. 2007), the services returned by the consistency-based matchmaker have the potential to supersede the ones that are derived by pure conceptual matching. Since information at both *conceptual* and *instance* level are used, we are able to significantly reduce the number of false positive matches, thereby increasing the chances of successfully completing the tasks requested by the user. The strength of our matchmaker lies in the ability to query, represent and utilize the *instance* level information directly from the services.

[1] UML and Object Constraints Language(OCL) are the languages used in our approach to author profiles. A discussion about the choice of languages is presented in Section 3.3

3. Synthesis of Web Applications as Service Processes

It is common that commercial service providers such as Airline companies expose their services in the form of web applications. An airline ticket booking application may have the following sequence of operations search for flights, review flights, select flights, and pay for the tickets. These individual operations are generally hidden within the web application and the order of execution of the operations is determined by user interaction. Even though existing research in the web services arena has resulted in a number of platform technologies (such as BPEL and WSDL) that are able to support the migration of these operations as services, most providers expose their services only in the form

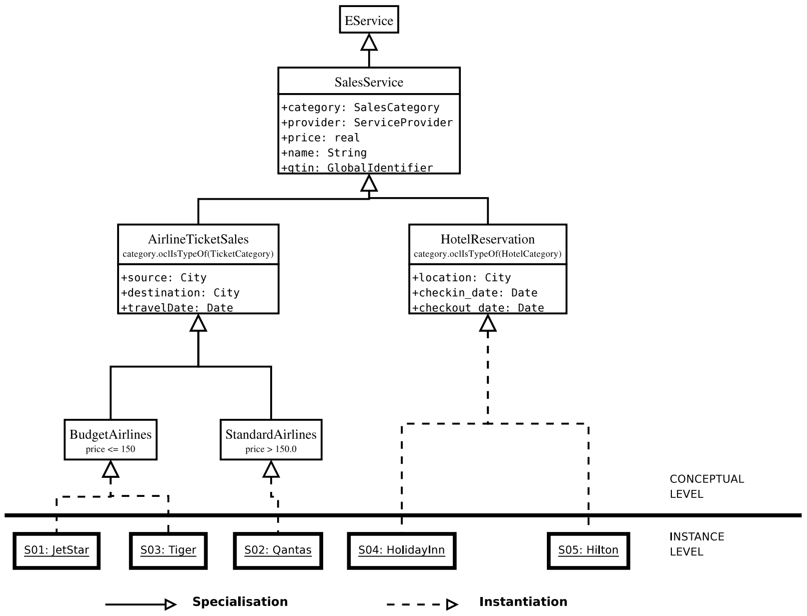


Figure 3. Repository of Conceptual and Instance level Profiles

of web applications aimed at human users. While these operations are potential sources of *instance* level information, acquiring the information directly requires to access the web application. Guided exploration of the application is essential to acquire the required *instance* level details as well as to avoid operations that either have permanent effects or do not produce relevant information. We propose to synthesise the web applications into service processes in order to be able to access them using standard web service calls. Following the synthesis, the service processes can potentially be used to query *instance* level information.

3.1 Qantas Booking Application

We use the Qantas[2] on-line ticket booking application to demonstrate the synthesis process. The application consists of a number of operations to perform the following tasks: searching for flights, picking flights, reviewing flights, and paying for the tickets. For instance, the four different operations that may be used to search for flights are as follows.

***oneway_fixed_search*:** search for flights between a source and a destination on a specific date;

***oneway_flexible_search*:** search for flights between a source and a destination on and around the given date;

***return_fixed_search*:** search for onward and return flights between a source and a destination on specific dates;

***return_flexible_search*:** search for onward and return flights between a source and a destination on and around given dates.

3.2 Synthesis of Web Applications

We consider the synthesis of web applications into service processes. The synthesised process of an application can then be used to acquire *instance* level information of the service by partially executing the process. We consider synthesis of an application at different levels of granularity, namely synthesis of operations, data models required to perform the operations, and effects of the operations including annotations of potential cost of execution[3] and additional information whether the effects of the operation can be undone. The additional information can potentially be used to avoid those operations that incur cost or have permanent effects.

Operations

In our approach, web applications are synthesised as service processes by wrapping every operation as a web service call qualified with a set of inputs, outputs, pre- and post-conditions. The behaviour of a process model is specified using a state chart where every transition is a web service call. The behaviour of a synthesised

[2] http://www.qantas.com.au

[3] For example, costs are involved in the *payment* operation

service process modelling the Qantas on-line airline ticket booking application is presented in Figure 4. Every transition in the state chart represents a web service call that corresponds to a specific operation in the web application. For example, the transition between the states *START* and *SELECT1* represents the service that corresponds to the operation *return_flexible_search* from the Qantas on-line ticket booking process in Section 3.1.

A transition from one state to another is carried out by making the corresponding web service call. Executing a web service call is analogous to performing the corresponding operation in the web application. Every operation in the web application is synthesised into a web service with inputs, outputs, pre- and post-conditions specifying the semantics of the service.

Data Models

Every input/output of an operation is synthesised as a web service input/output message. Figure 5 shows the input and output specifications of the various search operations supported in the process. The accompanying OCL constraints give additional information about the parameters. For example, *OneWaySearch* and *FlightSearchResult* in Figure 5 represent input and output messages of the service *oneway_flexible_search* (as shown in Figure 6).

Pre- and post-conditions represent the relationship between input and output messages. For example, the synthesised pre- and post-conditions of search services are presented in Figure 6. The post-conditions of the *oneway_fixed_search* service in Figure 6 specify that all the flight listings that are returned by the service agree with the inputs with respect to source, destination and travel date. The post-conditions of the *oneway_flexible_search* specify that the flight listings returned by the service includes both the flights on the requested date and flights travelling between the requested cities on other available dates. That the results of *oneway_fixed_search* is a subset of the results of *oneway_flexible_search* is captured through the inheritance specification between the corresponding output parameters *FlightSearchResult* and *FixedSearchResult* as shown in Figure 5.

The transitions of the synthesised process can further be annotated with the cost of executing a transition. As observed in (Carman et al. 2003), such annotations can either be derived from the domain knowledge or can be derived by analysing the input parameters and the service definition. Similarly, the states can be qualified as being either persistent or undoable. That is, whether a particular state can be rolled back to a previous state. For example in Figure 4, the transition *pay* incurs cost and leads to the persistent state *END*.

3.3 Specification Language

We use UML/OCL as the formalism to represent the semantics of the synthesised services. UML/OCL has been advocated as a comprehensive formalism for modelling declarative service semantics (Thiagarajan and Stumptner 2006; Albert et al. 2005; Zisman and Spanoudakis 2006). Existing approaches for modelling processes like OWL-S and BPEL leave the choice of the declarative language to the user. In our prior work (Thiagarajan and Stumptner 2006, 2007), we have also shown that UML/OCL offers better expressiveness of procedural knowledge compared with traditional semantic web formalisms.

The input, output, pre- and post-conditions of an operation serve as a declarative specification of its effects. In the next section, we present a strategy that utilizes these declarative specifications to determine the sequence of transitions to execute in order to acquire *instance* level information to answer specific queries.

4. Partial Execution of Service Process To Acquire Instance level Information

It is possible to acquire *instance* level details by executing the whole service process (as in (Carman and Serafini 2003)). A service is executed with synthesised inputs, and the similarity between the result and the request is computed to assess whether the service matches the request. While this strategy is useful in assessing atomic services, it is not suitable in the case of service processes because some operations may incur costs or may lead to some persistent state in the process. Executing a service process may have permanent effects and necessary compensation actions are required to rollback. For example to evaluate a ticket search query, it is not wise to execute the whole process in Figure 4 as it may incur costs as a result of the *pay* operation. Moreover, we may have to *cancel* the tickets as a rollback measure. As observed in (Pistore et al. 2005a), applications such as booking generally

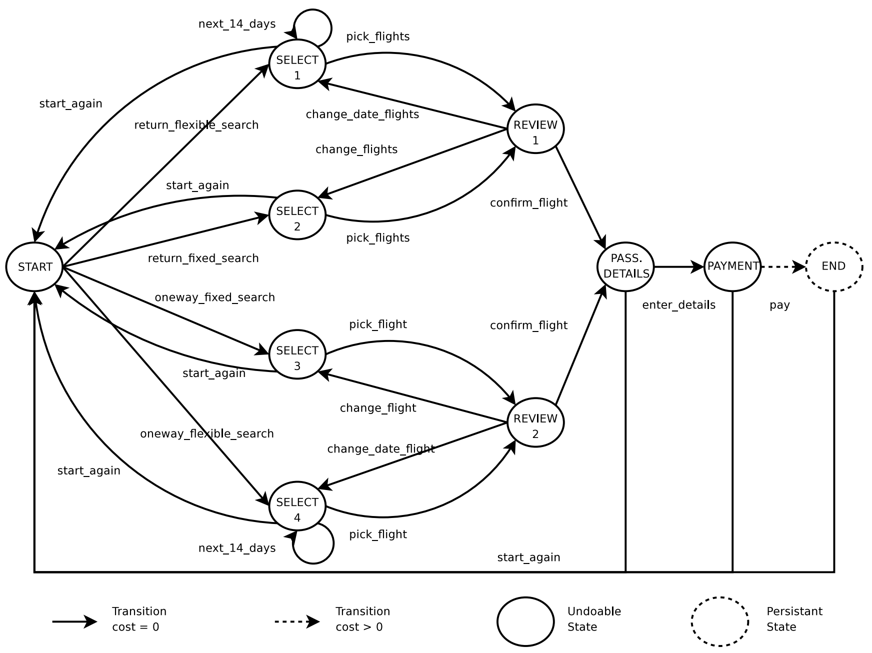


Figure 4. Synthesised Airline Booking Process

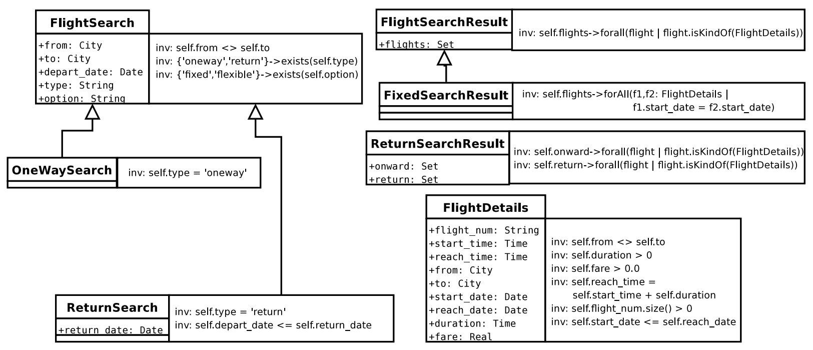

FlightSearch
+from: City
+to: City
+depart_date: Date
+type: String
+option: String
inv: self.from <> self.to
inv: {'oneway','return'}->exists(self.type)
inv: {'fixed','flexible'}->exists(self.option)
OneWaySearch
inv: self.type = 'oneway'
ReturnSearch
+return_date: Date
inv: self.type = 'return'
inv: self.depart_date <= self.return_date
FlightSearchResult
+flights: Set
inv: self.flights->forall(flight | flight.isKindOf(FlightDetails))
FixedSearchResult
inv: self.flights->forAll(f1,f2: FlightDetails | f1.start_date = f2.start_date)
ReturnSearchResult
+onward: Set
+return: Set
inv: self.onward->forall(flight | flight.isKindOf(FlightDetails))
inv: self.return->forall(flight | flight.isKindOf(FlightDetails))
FlightDetails
+flight_num: String
+start_time: Time
+reach_time: Time
+from: City
+to: City
+start_date: Date
+reach_date: Date
+duration: Time
+fare: Real
inv: self.from <> self.to
inv: self.duration > 0
inv: self.fare > 0.0
inv: self.reach_time = self.start_time + self.duration
inv: self.flight_num.size() > 0
inv: self.start_date <= self.reach_date

Figure 5. Synthesised Data Model

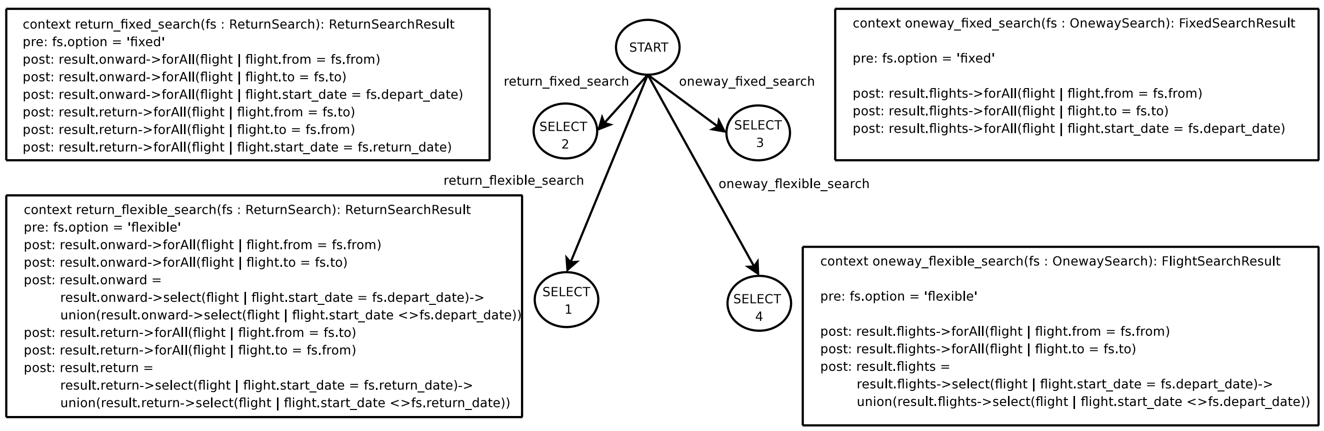
consists a number of steps and are not accessible as *one-off* atomic services.

Apart from avoiding persistent states and costs, it is also desired to have a guided action selection procedure that acquires the new and relevant *instance* details to answer specific queries. We propose a strategy to search the state chart of a synthesised process to determine the set of transitions that give us the required *instance* level details to evaluate a particular query. We can potentially acquire information at a low cost while avoiding to commit to any persistent operation. For example, to evaluate a ticket search query, we have to determine the sequence of service calls that should be made in order to gather enough details to answer a particular query while avoiding persistent states.

4.1 Transition Selection Strategy

Our transition selection strategy works by evaluating all the outgoing transitions in a current state and executing the one which results in a state that is closer to the goal state. Here, the goal state is the state in the process in which all the required information that is necessary to evaluate a request is available. The strategy evaluates the transitions by predicting the *new* and *relevant instance* level information that may be obtained by executing the transition, based on the current state and the specification of the transition. Any information that was previously unknown and is part of the request is considered *new* and *relevant*.

The evaluation is carried out in two steps. In the first step, we evaluate costs of executing the transitions



```
context return_fixed_search(fs : ReturnSearch): ReturnSearchResult
pre: fs.option = 'fixed'
post: result.onward->forAll(flight | flight.from = fs.from)
post: result.onward->forAll(flight | flight.to = fs.to)
post: result.onward->forAll(flight | flight.start_date = fs.depart_date)
post: result.return->forAll(flight | flight.from = fs.to)
post: result.return->forAll(flight | flight.to = fs.from)
post: result.return->forAll(flight | flight.start_date = fs.return_date)
```

```
context oneway_fixed_search(fs : OnewaySearch): FixedSearchResult

pre: fs.option = 'fixed'

post: result.flights->forAll(flight | flight.from = fs.from)
post: result.flights->forAll(flight | flight.to = fs.to)
post: result.flights->forAll(flight | flight.start_date = fs.depart_date)
```

```
context return_flexible_search(fs : ReturnSearch): ReturnSearchResult
pre: fs.option = 'flexible'
post: result.onward->forAll(flight | flight.from = fs.from)
post: result.onward->forAll(flight | flight.to = fs.to)
post: result.onward =
        result.onward->select(flight | flight.start_date = fs.depart_date)->
        union(result.onward->select(flight | flight.start_date <>fs.depart_date))
post: result.return->forAll(flight | flight.from = fs.to)
post: result.return->forAll(flight | flight.to = fs.from)
post: result.return =
        result.return->select(flight | flight.start_date = fs.return_date)->
        union(result.return->select(flight | flight.start_date <>fs.return_date))
```

```
context oneway_flexible_search(fs : OnewaySearch): FlightSearchResult

pre: fs.option = 'flexible'

post: result.flights->forAll(flight | flight.from = fs.from)
post: result.flights->forAll(flight | flight.to = fs.to)
post: result.flights =
        result.flights->select(flight | flight.start_date = fs.depart_date)->
        union(result.flights->select(flight | flight.start_date <>fs.depart_date))
```

Figure 6. Pre- and Post-condition of Search Transitions

and evaluate the type of the resultant state. We only consider those that do not have any costs and those that do not lead to a persistent state. In the second step, the output of every free transition is predicted by using the already known information, pre- and post-conditions of the transition. The predicted output states are analysed to determine their proximity/distance to the goal. The transition that leads closest to the goal is considered.

We demonstrate our approach in the context of evaluating the suitability of the service in Figure 3 for the query 'airline service that sells return tickets between Adelaide and Sydney for less than $400 starting on 23/11/2008 and returning on the 24/11/2008'. By applying the consistency-based matchmaking procedure in (Thiagarajan and Stumptner 2007; Thiagarajan et al. 2007), we determine that the only matching conceptual profile in the first step is *StandardAirlines*. Therefore, the instance profiles that specialise the *StandardAirlines* are potential matches that have to be further analysed. The *Qantas* service that instantiates the *StandardAirlines* profile is a potential match. We describe our approach to access service processes in the following and demonstrate our strategy by matching the query mentioned above with the Qantas process in Figure 4.

Physical Costs and Persistent States

The evaluation of the physical cost or resultant type is carried out by exploiting the effect annotations in the state chart of a service process. Transitions that incur costs or transitions that lead to a persistent state are ignored from consideration. For example in Figure 4, the *pay* transition incurs costs and is never executed. Since the transitions originating from the *START* state in Figure 4 are all free and do not lead to a persistent state, all transitions are potential candidates for the next step of the evaluation.

Undoable States

The second step of the evaluation involves predicting the output of a transition and measuring how close the predicted result is to the goal. The prediction of the output state of a transition is performed by analysing the inputs that can be provided from the current state to the transition and the relationship between the input and output values specified in the post-conditions of the transition. The inputs to a transition can either be derived from the requirements specification or from the results obtained from any of the transitions initiated prior to the current state.

Both the transitions *return_flexible_search* and *return_fixed_search* require *ReturnSearch* as input and both the transitions *oneway_flexible_search* and *oneway_fixed_search* require *OneWaySearch* as input. Given the request, we are able to derive the following information: $\{$*from = Adelaide, to = Sydney, depart_date = 23/11/2008, return_date = 24/11/2008, fare* $< 400\}$. In this case, we do not have any information from the prior transitions as the current state is also the first state. Thus, only the information derived from the requirements are mapped to the inputs required by the individual transitions.

Using the input information available, the output state of the transitions are predicted by taking into consideration the pre- and post-conditions that are in place. Predicting the outputs of the four transitions originating from the *START* state leads to four output states described as follows.

- *SELECT1:* A set of onward flights starting from *Adelaide* to *Sydney* including the ones on *23/11/2008* and a set of return flights from *Sydney* to *Adelaide* including the ones on *24/11/2008*.
- *SELECT2:* A set of onward flights starting from *Adelaide* to *Sydney* on *23/11/2008* and a set of return flights from *Sydney* to *Adelaide* on *24/11/2008*.
- *SELECT3:* A set of flights starting from *Adelaide* to *Sydney* on *23/11/2008*.
- *SELECT4:* A set of onward flights starting from *Adelaide* to *Sydney* including the ones on *23/11/2008*.

The predicted state is then compared to the goal state to determine how close this predicted state is to the goal. The estimate on the closeness is carried out by determining the amount of unknown information there is in the predicted state. Unknown information is any data that is requested in the query but is not available in the current state. The lower the amount of unknown information the closer it is to the goal. Following this prediction, the transition that predicts a state closest to the goal is chosen for execution.

The predicted states are now analysed with the requirements to determine the amount of missing information in each of the transitions. For example, the state *SELECT4* does not have the required return flight information. The location information such as *from* and *to* can be derived from the input but since the *return_date* parameter is not part of the input for the *oneway_fixed_search* the information about *return_date* will be unknown to evaluate the query. On the other hand, the state *SELECT1* has all the information required to answer the query. Under this strategy, both *return_flexible_search* and *return_fixed_search* seem viable transitions to take in order to acquire the required *instance* level details.

Once a transition is executed and if there is no unknown information in the resultant state, the information available is used to construct the instance profile of this service process. Once the instance profile is constructed, the instance profile and the request are checked for consistency. If the profile is consistent then the service is deemed a match for the request. The consistency check at the conceptual and *instance* levels are implemented as Constraint Satisfaction Problems (CSP) in our service composition framework in (Thiagarajan and Stumptner 2007; Thiagarajan et al. 2007). In this case, we can execute either *return_flexible_search* or *return_fixed_search* to get the list of onward and return flights to confirm or refute the query stated above.

5. Related Work

A web service planning approach that interleaves composition and execution of services is presented in (Carman and Serafini 2003; Carman et al. 2003). This approach is, to the best of our knowledge, the only work that proposes to execute services to determine their capability. The requirements are specified as documents expected out of service executions. Services are executed using synthesised inputs to determine whether they indeed result the required document. If a particular service does not produce the required document after a number of tries then another alternative service is considered. Our approach is different to this work in that we do not execute the whole service process but we first choose the best set of transitions to execute by reasoning on the specifications. We therefore avoid executing those transitions that incur cost or have permanent effect. Therefore, it is not necessary in our approach to explicitly state the rollback operations. Moreover, we have investigated a guided exploration strategy to select the 'best' transitions from a process to execute in order to gather details from the service. Moreover, we synthesise service processes in cases where directly accessible service models are unavailable.

A *process level* composition framework that works by combining service processes is presented in (Pistore et al. 2005a,b). The combined process is computer from the parallel product of the state machines of the individual processes. This work deals with the composition of previously discovered service processes. Our work complements the *process level* composition procedure in providing a discovery mechanism that analyses service specifications, partially executes services to obtain *instance* level details, and utilizes this information to match the request.

6. Conclusion

We considered the synthesis of web applications to service processes and presented a transition selection strategy to search the process for 'best' set of transitions to execute in order to obtain *instance* level information. Implementing the synthesis process and formalising the transition selection strategy within our semantic web service composition framework are among the future work.

References

Patrick Albert, Laurent Henocque, and Mathias Kleiner. Configuration Based Workflow Composition. In *Proc. of ICWS 2005*, July 2005.

Mark Carman and Luciano Serafini. Planning For Web Services the Hard Way. In *SAINT Workshops*, Orlando, FL, USA, January 2003.

Mark Carman, Luciano Serafini, and Paolo Traverso. Web Service Composition as Planning. In *ICAPS Workshop on Planning for Web Services*, Trento, Italy, June 2003.

Jos de Bruijn, Rubén Lara, Axel Polleres, and Dieter Fensel. OWL DL vs. OWL flight: conceptual modeling and reasoning for the semantic Web. In *Proc. of WWW 2005 Conf.*, Chiba, Japan, May 2005.

Roy Grønmo and Michael C. Jaeger. Model-Driven Semantic Web Service Composition. In *APSEC05*, pages 79–86, December 2005.

Takahiro Kawamura, Jacques-Albert De Blasio, Tetsuo Hasegawa, Massimo Paolucci, and Katia P. Sycara. Public Deployment of Semantic Service Matchmaker with UDDI Business Registry. In *Proc. ISWC*, pages 752–766, Hiroshima, Japan, November 2004.

Lei Li and Ian Horrocks. A software framework for matchmaking based on Semantic Web technology. In *Proc. of WWW 2003 Conf.*, pages 331–339, Budapest, May 2003.

Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia P. Sycara. Semantic matching of web services capabilities. In *Proc. ISWC*, pages 333–347, Sardinia, Italy, June 2002.

Peter F. Patel-Schneider and Ian Horrocks. A comparison of two modelling paradigms in the semantic web. *J. Web Sem.*, 5(4):240–250, 2007.

Marco Pistore, Pierluigi Roberti, and Paolo Traverso. Process-level composition of executable web services: ”on-the-fly” versus ”once-for-all” composition. In *Proc. of ESWC 2005*, pages 62–77, Heraklion, Crete, Greece, 2005a.

Marco Pistore, Paolo Traverso, Piergiorgio Bertoli, and Annapaola Marconi. Automated synthesis of executable web service compositions from BPEL4WS processes. In *WWW*, pages 1186–1187, 2005b.

Rajesh Thiagarajan and Markus Stumptner. A Native Ontology Approach for Semantic Service Descriptions. In *Proc. of AOW 2006*, pages 85–90, Hobart, Australia, 2006.

Rajesh Thiagarajan and Markus Stumptner. Service Composition With Consistency-based Matchmaking: A CSP-based Approach. In *Proc. ECOWS*, Halle, Germany, November 2007.

Rajesh Thiagarajan, Markus Stumptner, and Wolfgang Mayer. Semantic Web Service Composition by Consistency-based Model Refinement. In *Proc. of IEEE APSCC*, Tsukuba Science City, Japan, December 2007.

Hai Wang and Zengzhi Li. A Semantic Matchmaking Method of Web Services Based On $\mathcal{SHOIN}^{+}(D)*$. In *Proc. of IEEE APSCC*, pages 26–33, Guangzhou, China, December 2006.

Andrea Zisman and George Spanoudakis. UML-Based Service Discovery Framework. In *Proc. ICSOC*, pages 402–414, Chicago, IL, USA, December 2006.