A Planner for Composing Services Described in DAML-S

Mithun Sheshagiri*
University of Maryland, Baltimore County
1000 Hilltop Circle
Baltimore, Maryland 21250
mits1@csee.umbc.edu

Marie desJardins
University of Maryland, Baltimore County
1000 Hilltop Circle
Baltimore, Maryland 21250
mariedj@csee.umbc.edu

Timothy Finin
University of Maryland, Baltimore County
1000 Hilltop Circle
Baltimore, Maryland 21250
finin@csee.umbc.edu

ABSTRACT

A web service is a web-accessible piece of software or hardware. In recent years, industry has been showing increasing interest in web services as a technology for building distributed web applications. However, web services as a technology lacks in several departments. Representations for describing web services have been widely investigated by industry and academia. Service composition-that is, automated methods for constructing a sequence of web services to achieve a desired result- has been relatively neglected. We present in this paper, a planner that composes atomic/basic services described in DAML-S [4] into a composite service. We discuss issues involved with the design of planners for composition. We also propose a set of guidelines for describing services that facilitates composition.

Keywords

Web Services, Automatic Composition, DAML-S, Planning

1. INTRODUCTION

Web services are a relatively new paradigm for building distributed web applications. In spite of keen interest shown by industry in web services, several obstacles have prevented companies from effectively harnessing web service technology to build web applications. One obstacle is the lack of a set of tools that would allow developers (or intelligent agents) to describe, discover and compose web services. WSDL [6] is a popular XML-based language for describing web services, but it does not capture semantics. DAML-S is an application of DAML+OIL [2] that can be used for semantic description of web services. DAML-S consists of a set of ontologies that provide a vocabulary to describe services. The use of semantics enables inference about the requirements and effects of services, which in turn facilitates automatic discovery and reasoning. We have built a planner that uses STRIPS-style services to compose a plan, given the goal and a set of basic services. We have used the Java Expert Shell System (JESS) [7] to implement the planner and a set of JESS rules that translate DAML-S descriptions of atomic services into planning operators. Our experience with building this planner revealed certain desirable properties of service descriptions that will make composition for a planner easier. We also stress the importance of developing ontologies that can capture relationships between services, which are crucial for composition.

*Mithun Sheshagiri is a graduate student at UMBC

AAMAS '03 Melbourne, Australia

2. RELATED WORK

SWORD [10] is a model for web service composition. However, it uses its own simple description language and does not support any existing standards like WSDL or DAML-S. Services are modeled using inputs and outputs, which are specified using an Entity Relationship model. Inputs are classified into conditional inputs and data inputs. Outputs are classified similarly. Conditional inputs/outputs are assertions about entities on which the service operates and the relationships between entities. Data inputs/outputs constitute the actual data (attributes of entities) that the service uses.

The focus of [12] is primarily on information discovery, extraction and integration and does not deal specifically with web services or automatic composition. This system has been been designed to work with a specific set of services; our system is designed to be a generic service composer. This system describes a forward-chaining composer, specialized to information integration queries; our system uses a backward-chaining planner and we plan to build a more general partial-order planner. Like [10], they have their own model for describing web services; we specifically deal with services described in DAML-S.

A similar framework is being developed at IBM Research Laboratories as part of the Web Services Toolkit (WSTK) [1]. A composition engine [11] has been built for services described in WSDL. Although [11] describes the use of a planner for composition, constructing operators from the service description is not fully automated. This is primarily because of the absence of a mechanism to capture domain knowledge in WSDL. Our planner makes use of services described in DAML-S. DAML+OIL helps us to describe explicit ontologies for capturing domain language. This added knowledge gives our planner greater versatility and helps compose complex services.

Golog has been used for service composition [9] by constructing general templates that are then modified based on user preferences, yielding a composite plan. The templates are not automatically built and constitute part of the plan. Our approach is able to build plans dynamically from scratch and does not rely on templates for composition.

3. SERVICE DESCRIPTION USING DAML-S

DAML-S uses DAML+OIL to describe a set of ontologies for characterizing web services. DAML-S [3] describes the following ontologies:

1. A *Service* ontology that forms the topmost part of a hierarchy of services.

2. A *Service* "presents" a *ServiceProfile*. The ServiceProfile is typically used for advertising and discovering the service.

3. A *Service* "isDescribedBy" a *ServiceModel*. The ServiceModel contains information required for composition: inputs, outputs, preconditions and effects (IOPEs). A *ProcessModel* is a type of *ServiceModel*.

4. A *Service* "supports" a *ServiceGrounding*. *ServiceGrounding* provides low-level details like communication protocols (e.g., RPC, SOAP), ports and descriptions of data structures exchanged. This information is used by a web service execution engine to actually invoke services.

The manner in which DAML-S is used is slightly unusual. Typically when using DAML+OIL, a schema or ontology is designed and concepts (classes and properties) defined in this schema are instantiated as a next step. DAML-S uses an intermediate step where services are described as new schemas and not instantiated. The actual instantiation of services is done only during run time. This is analogous to writing procedures which form the service descriptions and a call to these procedures creates an instance of this procedure [8]. An atomic service is a directly invocable service that executes in a single step. An atomic service consists of inputs, preconditions, effects and outputs. These are defined as properties in DAML-S specifications.

4. PLANNER

The Service model describes the service and consists of inputs, preconditions, outputs and effects of services. The first step in planning a service composition consists of converting DAML-S *ServiceModel* descriptions of services into Verb-Subject-Object(VSO) triples. All VSO triples corresponding to the services are asserted into the JESS KB as facts.

The next step involves building planning operators that correspond to each of the atomic services. We have written a set of rules and queries (*defrules* and *defqueries* in JESS) that transform services encoded as VSO triples into a set of facts that form the planning operator. The planning operators are similar to STRIPS planning operators and consist of a service name, inputs, preconditions, outputs and effects.

We characterize inputs and preconditions (IPs) into two types: external IPs and internal IPs. External IPs are IPs that are provided directly by the agent or satisfied externally and are not generated as effects or outputs of other operators. Internal IPs, on the other hand, can be provided or satisfied only by other operators/services. For example, the user provides his/her contact information through some interface. Therefore, this information is an external IP. Suppose that there is an operator that produces the effect *ProfilePresent* when user fills in his/her profile information. In this case, *ProfilePresent* is an internally generated effect; therefore it is an internal IP if used as a precondition by another operator.

Outputs and effects as defined by DAML-S specifications can be conditional: that is, depending on the current state of the world, the effect or output of a service can be different. It is not necessary for a composer to represent the control structure for conditional effects and outputs; merely enumerating them as multiple effects or outputs is sufficient for composition. The DAML-S website includes a sample book-buying domain called *congo.com*, which includes service descriptions and ontologies associated with the domain. Figure 1 shows an example of a service from this domain that creates an account:

```
<daml:Class rdf:ID="CreateAcct">
 <rdfs:subClassOf rdf:resource="&process;#AtomicProcess"/>
 <rdfs:subClassOf>
  <daml:Restriction daml:cardinality="1">
   <daml:onProperty rdf:resource="#createAcctInfo"/>
  </daml:Restriction>
 </rdfs:subClassOf>
 <rdfs:subClassOf>
  <daml:Restriction daml:cardinality="1">
   <daml:onProperty rdf:resource="#createAcctOutput"/>
  </daml:Restriction>
 </rdfs:subClassOf>
</daml:Class>

<rdf:Property rdf:ID="createAcctInfo">
 <rdfs:subPropertyOf rdf:resource="&process;#input"/>
 <rdfs:domain rdf:resource="#CreateAcct"/>
 <rdfs:range rdf:resource="#AcctInfo"/>
</rdf:Property>

<daml:Property rdf:ID="createAcctOutput">
 <rdfs:subPropertyOf rdf:resource="&process;#output"/>
 <rdfs:domain rdf:resource="#CreateAcct"/>
 <rdfs:range>
  <daml:Class>
   <rdfs:subClassOf rdf:resource="&process;#UnConditionalOutput"/>
   <rdfs:subClassOf>
    <daml:Restriction>
     <daml:onProperty rdf:resource="&process;#coOutput"/>
     <daml:toClass rdf:resource="#CreateAcctOutputType"/>
    </daml:Restriction>
   </rdfs:subClassOf>
  </daml:Class>
 </rdfs:range>
</daml:Property>
```

Figure 1: CreateAcct Service described in DAML-S

The equivalent planning operator of the service described in Figure 1 is the following set of facts:

```
(sname http://www.csee.umbc.edu/mits1/daml/modCongoProcess.daml#CreateAcct)
(precon http://www.csee.umbc.edu/mits1/daml/modCongoProcess.daml#CreateAcct
   nil)
(input http://www.csee.umbc.edu/mits1/daml/modCongoProcess.daml#CreateAcct
   http://www.csee.umbc.edu/mits1/daml/modCongoProcess.daml#createAcctInfo)
(effect http://www.csee.umbc.edu/mits1/daml/modCongoProcess.daml#CreateAcct
   nil)
(output http://www.csee.umbc.edu/mits1/daml/modCongoProcess.daml#CreateAcct
   http://www.csee.umbc.edu/mits1/daml/modCongoProcess.daml#createAcctOutput)
```

Since the output is unconditional in the above case, we

just have a single output. Now consider an example of a service which produces an effect *BookPresent* if *BookInStock* is true or *BookAbsent* if *BookInStock* is false. The equivalent planning operator will have the following effects as follows.

```
(sname foo)
(input foo BookName)
---
(effect foo BookPresent)
(effect foo BookAbsent)
```

Although the service description exposes the *BookInStock* condition, its "value" is decided by the input *BookName* and therefore not known in advance. Such services are therefore non-deterministic. The planning operators for these services show both effects. The actual effect cannot be ascertained till execution time. We assume the that the effect produced during execution is the one desired by us. We plan a thorough treatment of this issue in our next version. In cases where the value of the condition is known in advance, the planning operators are built using a technique similar to the "when" clause described in [13]. The approach for conditional ouputs is similar.

```
(sname S1)
--
(condeffect S1 CE1)
  (CE1 FLAGA E1)
  (CE1 FLAGB E2)  //FLAGA = (¬FLAGB)
```

The planner applies the following two steps repeatedly until none of the services satisfy any of the goals.

1. Find services that satisfy existing goal and store them (add them to the plan).

2. Convert the inputs and preconditions of all the operators stored in step 1 into new set of goals.

Composition is successful if all outstanding goals are external IPs. We have developed a set of atomic services that are similar to the set of services described for *congo.com*. The equivalent planning operators are shown below. For the sake of readability complete URI's have not been used.

```
(sname Login)
(input Login UserName)
(output Login UserType)

(sname GetInfo)
(input GetInfo UserType)
(effect GetInfo ProfileExists)
(effect GetInfo ProfileDoesNotExist)

(sname QueryUser)
(precon QueryUser ProfileDoesNotExist)
(output QueryUser AskUser)

(sname BookLookUp)
(input BookLookUp BookName)
(output BookLookUp ISBN)
(output BookLookUp BookInStock)
(output BookLookUp BookOutOfStock)

(sname PutInCart)
(input PutInCart ISBN)
(precon PutInCart ProfileExists)
(precon PutInCart BookInStock)
(effect PutInCart InCart)

(sname CreditCard)
(input CreditCard CardType)
(input CreditCard CardNum)
(input CreditCard CardExpiryDate)
(effect CreditCard Approved)
(effect CreditCard NotApproved)

(sname ShipItem)
(precon ShipItem InCart)
(precon ShipItem Approved)
(effect ShipItem BookShipped)
(output ShipItem InformUser)
```

The goal we specify to start composition is *BookShipped*. Figure 2 shows the plan generated by the planner. The light gray rectangles represent external IPs-that is, information that must be provided by the user or some external source such as database. The dark rectangles request internal IPs-that is, preconditions that are satisfied by effects of services. The ellipses represent atomic services to be executed. The structure of the plan imposes a partial ordering on the execution of the services. For example, *Login* must be executed before *GetInfo*, but either of these steps can be executed in parallel with *BookLookUp*.

5. DESCRIBING SERVICES FOR COMPOSITION

Current specifications permit the description of services without effects or outputs. We claim that it is difficult to compose services without effects or outputs. A web service either provides information (has outputs) or alters the world (has effects) or both. In either case, the effect or output represents the change in the state of the world that the execution brings about. A planner uses this information to form links with other services. If one describes a service without any effects or outputs, then the change of state is not made explicit, and therefore a composer would have to maintain additional data to represent the change of state. The effects or outputs describe what the service does and are therefore essential, for composition. Our recommendation is that DAML-S specifications should be modified to make specification of outputs or effects mandatory.

6. SERVICE DESCRIPTION AND ADDITIONAL CONSTRAINTS

The ServiceModel only lets one describe core properties of services. Non-trivial composition of services from atomic services involves factors like business logic, changing market scenarios, temporal dependencies between services. These, though required, are not (and typically should not be [5]) part of the service.

Consider the example of a book look-up service that takes a book's name as input and gives the ISBN number as an output. Companies A and B sell books and use the book look-up service to obtain the ISBN. Company A has a policy that lets any user pose an ISBN request. Company B only

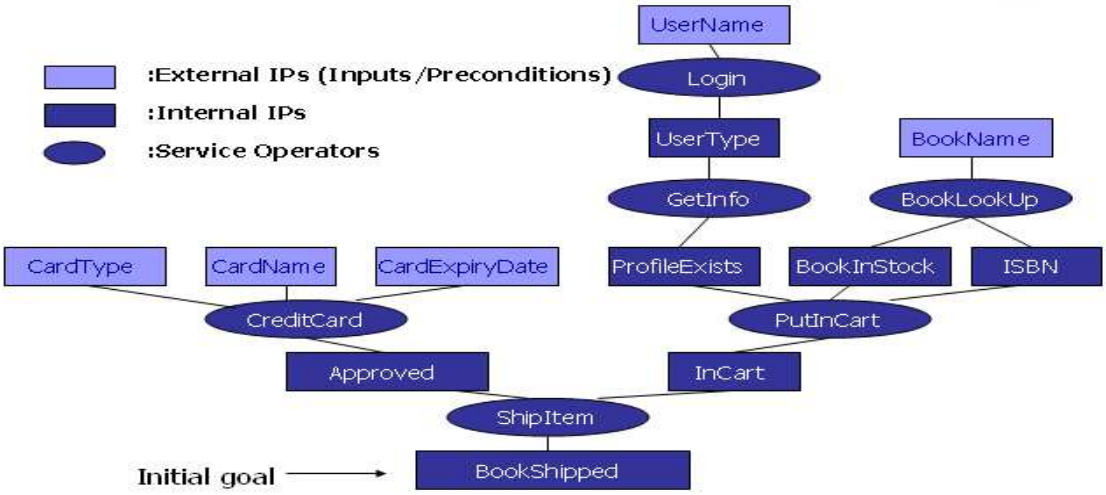

Figure 2: Plan generated by composer

lets users registered with them to use the ISBN look-up. Company B therefore needs a similar ISBN look-up service with an additional precondition. Instead of designing a new service for B, we could provide a generic description of the service and let B use it as it wants to. Company B could define their policy using DAML+OIL which specifies additional constraints on the operator. It is also in the interest of the service provider to advertise a generic version of the service to enable varied entities to use it.

We therefore claim that for non-trivial composition, a combination of a general description of the service and some additional logic description is required. DAML lets you express these two parts in a single language and therefore is a better choice for describing services. WSDL currently provides a vocabulary that lets you describe just the services; therefore it is difficult for a system using WSDL alone to compose a service from atomic services.

The design of our planner is in accordance with the above principle: the actual planning engine is domain-independent. The planner can be configured to handle specific domains by plugging in domain specific ontologies that capture the additional constraints. The service itself provides a minimal core set of IOPEs that form the planning operator; additional knowledge can then be added that uses domain-centric ontologies to modify the existing IOPEs.

7. FUTURE WORK

In the near term, we plan to design an ontology that would let one specify relations among services as a policy, as described in the previous section. We also plan to demonstrate our framework in a pervasive computing environment. In this domain, the composer might come up with different plans for the same goal. The agent's location, speed and physical limitations determine the best plan; these parameters will be captured as additional knowledge.

The planner we have currently implemented is a proof of concept, demonstrating that service composition can be performed automatically from DAML-S descriptions of atomic services. In future work, we plan to develop a more sophisticated planner that can apply planning methods to do service composition in real-world environments. We have identified three key challenges in developing such a planner: (1) Selecting among alternative atomic and composed services; (2) reasoning about interactions between atomic services within the plan; and (3) interleaving planning and execution during service composition. In the following subsections, we briefly discuss the key issues in each of these areas and our proposed approach to addressing them.

7.1 Choosing Among Alternatives

Our first version of the planner is a simple backward-chaining algorithm that assumes that there is always a single service that satisfies a given goal. There can easily be scenarios where a goal can be satisfied by more than one service. In these cases, choosing an appropriate operator is an obvious requirement. Composition might succeed or fail based on the choice of service, since different operators might have different sets of inputs and preconditions, which form the new set of goals that need to be satisfied in the next planning cycle. One possible approach would be to generate all possible plans, then choose the best alternative based on user specified criteria such as cost. This technique, however, is not scalable. An alternative would be to use a best-first backtracking approach by applying heuristics such as selecting the operator (service) with the least number of preconditions or inputs, or selecting operators that have pre-conditions that are known to be easily satisfiable.

7.2 Reasoning About Interactions

Our current planner does not reason about interactions between actions within the plan. However, it may be the case that one action interferes with another (e.g., clearing out the user's cookies in between two service invocations on the same website may delete the necessary authorizations, requiring more inputs from the user), or that the effects of a service can be used to satisfy preconditions of more than one action in the plan (e.g., looking up the title of a book

gives not only the ISBN number to use for ordering the book, but also the full publication information to enter in a bibliography being produced elsewhere in the plan). The partial-order planning and hierarchical task network planning paradigms both provide models for reasoning about interactions within the plan, and resolving conflicts (or taking advantage of opportunities).

7.3 Interleaving Planning and Execution

There are two ways to compose and execute a composite service. In the first approach, planning and execution are separate. The second, more advanced, approach interleaves planning and execution. The initial planner we have implemented is an instance of the former approach: the planner produces a plan assuming that all involved services will be available and execute normally. The design of the planner will be different for a framework that supports parallel composition and execution. In this case, services are checked for availability and successful execution before being included in the final plan. If a service for some reason cannot be executed, then the execution engine requests an alternative service from the planner. More generally, the failure to execute a service could lead to a cascading effect, requiring the planner to supply an alternative sub-plan. It is not possible at any stage of the planning process to predict the success of the plan. Therefore, not all services can be checked by executing them; checking can be performed only for idempotent services that do not alter world state. The advantage of this framework is its capability to produce plans that are more likely to be executed successfully compared to the ones produced by a framework with disconnected composition and execution. Ideally, the planner would not only be able to dynamically replan in case a service fails, but generate contingency plans to anticipate such failures.

8. CONCLUSION

The overwhelming interest shown by industry indicates that web service as a technology is here to stay. However, a rich set of tools are required to demonstrate its use in building distributed web applications. We present a planner for composing services that handles services described in DAML-S. We also argue that additional logic is required for composing non-trivial tasks. This can be done by the use of explicit domain-specific ontologies.

9. REFERENCES

[1] Web Services Toolkit, http://www.alphaworks.ibm.com/tech/webservicestoolkit, 2000.

[2] DARPA Agent Markup Language, http://www.daml.org, 2001.

[3] DAML-S Ontologies, http://www.daml.org/services/0.7, 2002.

[4] A. Ankolenkar, M. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, and K. Sycara. DAML-S: Semantic Markup for Web Services. In *The First International Semantic Web Conference (ISWC), Sardinia (Italy)*, 2002.

[5] C. Bussler, A. Maedche, and D. Fensel. A Conceptual Architecture for Semantic Web Enabled Web Services. 2002.

[6] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Service Description Language, http://www.w3.org/TR/wsdl, 2001.

[7] E. J. Friedman-Hill. Jess, The Expert System Shell for the Java Platform, 2002.

[8] D. Martin. Web Services Mailing List, http://lists.w3.org/Archives/Public/www-ws/2001Nov/0009.html, 2001.

[9] S. McIlraith and T. Son. Adapting Golog for Composition of Semantic Web Services. In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR2002), Toulouse, France*, 2002.

[10] S. R. Ponnekanti and A. Fox. SWORD: A Developer Toolkit for Web Service Composition. In *Proc. of the Eleventh International World Wide Web Conference, Honolulu, HI*, 2002.

[11] B. Srivastava. Automatic Web Services Composition Using Planning. In *Proc. of KBCS 2002, Mumbai, India*, December 2002.

[12] S. Thakkar, C. A. Knoblock, J. L. Ambite, and C. Shahabi. Dynamically Composing Web Services from On-line Sources. In *Workshop on Intelligent Service Integration, The Eigth National Conference on Artificial Intelligence (AAAI), Edmonton, Alberta, Cananda*, 2002.

[13] D. S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.