SWORD: A Developer Toolkit for Web Service Composition

Shankar R. Ponnekanti and Armando Fox
Computer Science Dept
Stanford University
Stanford, CA 94305
{pshankar, fox}@cs.stanford.edu

Abstract:

The formidable problem of automatic or semi-automatic composition of existing Web services is the subject of much current attention. We address a particular subset of this problem with SWORD, a set of tools for the composition of a class of web services including ``information-providing'' services. In SWORD, a service is represented by a rule that expresses that given certain inputs, the service is capable of producing particular outputs. A rule-based expert system is then used to automatically determine whether a desired composite service can be realized using existing services. If so, this derivation is used to construct a plan that when executed instantiates the composite service. As our working prototype and examples demonstrate, SWORD does not require (but could benefit from) wider deployment of emerging service-description standards such as WSDL, SOAP, RDF and DAML. We also distinguish SWORD from some other plausible existing approaches, especially information integration. We show that although SWORD's expressive capabilities are weaker, the abstractions it exposes capture more appropriately the limited kinds of queries supported by typical Web services and thus result in simplicity and efficiency. [Word Count: 7950 words]

Keywords: Information integration, service composition, service integration, rule-based system.

1. Introduction

Service composition: the problem of composing autonomous services to achieve new functionality, is generating considerable interest in recent years in several computer science communities. Service composition has the potential to reduce development time and effort for new applications. The web is a particularly interesting domain for service composition for several reasons. Firstly, increasing numbers of interesting services are moving online and the web is fast transforming from a collection of static pages to a provider of numerous useful services. Another reason is that web services conform to the standard HTTP protocol which makes it (relatively) easier to integrate them into a common framework. Third, because the web has several independent service providers providing related services, there is an inherent need for composing complementary services provided by independent providers to achieve the end-user's needs.

The service composition problem is particularly challenging because web services fall into many categories and composing such a diverse set of services may require several different tools, techniques, and technologies. In this paper, we propose one such toolset: SWORD. SWORD is a toolset that allows service developers to quickly compose *base* web services to realize new *composite* web services. SWORD can compose information providing services (such as the web services providing information about people, movies, theaters, restaurants, etc) and a class of other services (such as email and image conversion services). The key idea behind SWORD is as follows:

1. **Individual services are defined in terms of their inputs and outputs in an (entity relationship based) ``world model''. Given the inputs and outputs of the service, a *rule* (as in rule-based expert systems [24]) is then defined which indicates which outputs can be obtained by the service given which inputs.**
2. **When a developer wishes to create and deploy a new composite service, she specifies the inputs and outputs of the composite service in the world model and submits it to SWORD.**

3. **SWORD determines using a rule engine if the composite service can be realized using the existing services. If so, SWORD generates a *composition plan* for the composite service.**
4. **The developer can then view the generated plan and if appropriate, request that a persistent representation of the plan be generated. This representation contains the sequence of services that need be invoked to obtain the composite service outputs from its inputs. Roughly speaking, each step used during the derivation (i.e., each rule that fired) in the rule engine corresponds to a service invocation.**
5. **When an actual request for the composite service is received, the service(s) specified in the plan are executed, starting with the known inputs, in order to compute the desired outputs. In practice, some services can yield multiple responses to a query for which a single answer is logically desired; and some services can legitimately yield multiple matches. (As an example of the first, a name-to-address lookup service may return multiple matches if there are multiple individuals with the same name. As an example of the second, a service that returns restaurants within a certain radius will likely return multiple restaurants). If the user's goal was (eg) to obtain driving directions, the user would have to select a particular restaurant with which to continue execution. Our execution environment provides a way of resolving these cases, either by prompting the user or by supplying *filter* code. We focus on making the overall system, including this filtering mechanism, very easy to use so that the cost to the user of trying multiple options is low.**

The contributions of SWORD are three-fold:

- **A composition model well-suited to typical informational web services as well as certain other web services.**
- **An execution model for the composite services with a customizable filter mechanism that makes the composite services easy to use. The cost to the user of trying multiple options when services (expectedly or otherwise) produce multiple results is low.**
- **An implemented prototype, which demonstrates that SWORD can work with *today's web services* unlike some other upcoming research/industry projects which rely on the deployment of standards such as SOAP [28], WSDL [29], UDDI [8], RDF [27] and/or DAML [14]. Further, given the simplicity of our model, it can easily be extended to work with such standards if they are widely deployed in the future.**

Note that in general, SWORD *cannot* currently handle web services with various side-effects, such as services involving account credits/debits, or various other business-business services. This is an area of future work and we plan to explore how the SWORD model can be (incrementally) expanded to include other types of web services.

In this paper, we describe the composition and execution model of SWORD and demonstrate the prototype implementation we have built. The paper is organized as follows: In section 2, we explain the SWORD service model and composition plan generation. In section 3, we analyze the strengths and weaknesses of the SWORD composition model. While doing so, we contrast the SWORD composition model with existing work on *data integration* or *information integration* which has been the focus of much research in the databases and AI communities for several years [19, 12]. We do so for two reasons:

1. **To help better illustrate the strengths and limitations of SWORD**
2. **To highlight the differences between the two and establish that these approaches are meant to solve different problems**

In section 4, we demonstrate the SWORD prototype in action with an example. The SWORD execution model is detailed in section 5. We deal with several miscellaneous unaddressed issues in section 6. Finally, we review related research and industry efforts and conclude.

2. Service Model and Plan Generation

In this section, we describe in greater detail the service model and the composition model. Then, we describe *rule-based plan generation*: the mechanism currently used in SWORD to generate composite service plans.

2.1 Service Model

We model each service by its inputs and outputs. These inputs and outputs are specified in a world model. The world model currently needs be built at the ``composition site'' by the *base service modelers* (developers integrating underlying web services

that will be used to create composite services). Using the terminology of the ER model [26], the world model consists of entities (such as people, images, emails, restaurants, stocks, and movies), and relationships among entities (a theater shows a movie, etc). Entities have attributes (such as the name of a person). Unlike the traditional use of ER model for modeling data, we use it to describe the inputs and outputs of web services. The following must be noted:

- **All the attributes of an entity need not be known in advance but can be incrementally added as needed.**
- **Attributes in our context are better thought of as accessor methods that return the value of a particular property, since we are not referring to data stored in a table or a file, but to the value returned when a web service is invoked.**
- **Relationship conditions among entities may include conditions such as ``images X and Y have the same content'', ``mail X was sent to person Y'', etc.**
- **SWORD itself does not provide a standard set of entities and relationship conditions and the associated semantics. Rather, the choice of entities and relationships to use is left to the base service modelers. SWORD only provides a mechanism to compose web services whose inputs and outputs have been expressed using a set of entities and relationships.**

Every service has two types of inputs: conditional inputs (that are assertions specifying the entities the service operates upon and the relationship conditions between the entities) and data inputs (the actual data required by the services expressed in terms of the attributes of the involved entities). The same holds true for the outputs: there are both conditional and data outputs. As a concrete example, consider the Yahoo people lookup service. This service is modeled using the following inputs and outputs:

```
Entities involved: X
Condition Inputs:
  Person(X) - indicates that X is a person entity
Data Inputs:
  firstname(X), lastname(X), city(X), state(X)
  - indicates that the service needs these four attributes of X

Condition Outputs:
  none - no condition outputs for this service
Data Outputs:
  streetaddress(X), phone(X)
  - indicates that the street address and phone attributes of the
    person are returned by the lookup service.
```

Two other examples are shown below: an email service and a driving directions service.

```
EMail service:
Entities involved: X, Y
Condition Inputs:
  EMailMessage(X) - indicates that X is an email message
Data Inputs:
  subject(X), body(X), emailaddress(Y)
Condition Outputs:
  MailSent(X, Y)
Data Outputs: none

Driving Directions service:
Entities involved: X, Y
Condition Inputs:
  none
Data Inputs:
  streetaddress(X), city(X), state(X),
  streetaddress(Y), city(Y), state(Y)
Condition Outputs:
  none
Data Outputs:
  drivingdirections(X, Y)
```

In reality, this information (i.e., the condition and data inputs/outputs) alone is not sufficient to model a service. We also need run time information about how the service can be invoked and how it returns the results (if any). While we do not show this here, the complete service model descriptions also include this information. We will come back to this issue in greater detail when we discuss the SWORD execution model.

2.2 Composition

Suppose we wish to create a service that looks up the driving directions between two persons' homes given their names and cities. The inputs and outputs for this composite service are:

```
Names to Driving Directions service:
Entities involved: X, Y
Condition Inputs: Person(X), Person(Y)
Data Inputs:
    firstname(X), lastname(X), city(X), state(X),
    firstname(Y), lastname(Y), city(Y), state(Y)
Condition Outputs: none
Data Outputs: drivingdirections(X, Y)
```

This composite service can be realized by composing the people lookup service and the driving directions service. SWORD attempts to determine this automatically from the service specifications. In general, this can be modeled as a planning problem as follows:

- **Each service can be modeled as an action.**
- **The precondition for the action is a conjunction of all the condition inputs and the *known facts* corresponding to the data inputs of the service. The known fact corresponding to an attribute attr($X_1, X_2, .., X_m$) is defined as Known(attr, $X_1, X_2, .., X_m$). Thus, the known fact corresponding to the data input: input($X_1, X_2, .., X_n$), is Known(input, $X_1, X_2, .., X_n$).**
- **The postcondition for the action is a conjunction of all condition outputs and the known facts corresponding to the data outputs of the service.**
- **The initial state is a conjunction of all the condition inputs of the composite service and all the known facts corresponding to the data inputs of the composite service.**
- **The desired final state is a conjunction of all the condition outputs of the composite service and all the known facts corresponding to the data outputs of the composite service.**

Example:

```
People lookup service:
precondition:
  Person(X) & Known(firstname, X) & Known(lastname, X)
  & Known(city, X) & Known(state, X)
postcondition:
  Known(streetaddress, X), Known(phone, X)

Driving directions service:
precondition:
  Known(streetaddress, X), Known(city, X), Known(state, X)
  Known(streetaddress, Y), Known(city, Y), Known(state, Y)
postcondition:
  Known(drivingdirections, X, Y)

Names-to-driving-directions composite service:
Initial state:
```

```
  Person(X), Person(Y), Known(firstname, X),
  Known(lastname, X), Known(city, X), Known(state, X)
  Known(firstname, Y), Known(lastname, Y),
  Known(city, Y), Known(state, Y)
Desired final state:
  Known(drivingdirections, X, Y)
```

2.3 Rule-based plan generation

For all the services we currently model, the preconditions and postconditions can be expressed as conjunction of negation free facts. Further, we also avoid the use of function symbols or arithmetic predicates. (Some of these limitations can be overcome through a *run time filter mechanism* that will be explained later.) For this class of services, plan generation can be achieved efficiently using a rule-based expert system [24]. (Given a set of initial facts and a set of if-then rules, an expert system provides a mechanism of obtaining the facts that can be derived from the initial facts and rules. We use a rule-based engine implemented in Java, called Jess [15].)

When modeling a service, a base service modeler simply defines a (Horn) rule of the form: precondition $\Rightarrow$ postcondition.

Example: The rules corresponding to the people lookup service and the driving directions service are as follows:

```
Person(X) & Known(firstName, X) & Known(lastName, X) &
Known(city, X) & Known(state, X)
=> Known(streetaddress, X) & Known(phone, X)

Known(streetaddress, X) & Known(city, X) & Knwon(state, X) &
Known(streetaddress, Y) & Known(city, Y) & Knwon(state, Y)
=> Known(drivingdirections, X, Y)
```

Note that these rules are part of the service models and need be defined manually *only once* per base service, and *not* every time a *composite service developer* wishes to create a composite service! To create a composite service, the developer only needs specify the initial and final state facts for the composite service. The composite service developer need *not* even be familiar with which underlying services are available. (For the sake of clarity, we distinguish the composite service developer from the base service modeler. Of course, they could be the same person. Also, an unqualified ``developer'' refers to a composite service developer.)

Given the initial and final state facts supplied by the developer for the desired composite service, and given the rules for the base services available in their respective service models, SWORD uses the following algorithm to determine if a plan exists for the composite service:

Step 1: SWORD asserts the rules corresponding to all the available services in Jess. Thus, SWORD asserts the two rules defined above (people lookup service rule and driving directions service rule) and similarly defined rules corresponding to all the other available services.

Step 2: SWORD asserts the initial state facts for the composite service. For the names-to-driving-directions service, SWORD asserts the following initial facts:

```
Person(A)
Person(B)
Known(firstname, A)
Known(lastname, A)
Known(city, A)
Known(state, A)
```

```
Known(firstname, B)
Known(lastname, B)
Known(city, B)
Known(state, B)
```

Step 3: SWORD runs the engine which causes matching rules to fire. When the rules stop firing, SWORD queries the engine for the facts corresponding to the final state of the composite service. For the names-to-driving-directions composite service, SWORD queries the engine for the following fact:

```
Known(drivingdirections, A, B)
```

Th conditions on the rules (both LHS and RHS are conjunctions of facts that do not use negations or arithmetic/function symbols) ensure that:

1. **Forward chaining terminates**
2. **The same facts are derived regardless of the order in which rules may fire. (Although the order does determine the actual plan selected by SWORD in the next step.)**

Step 4: If the final state facts can be derived, then SWORD constructs a plan from the derivation structure (which is inferred from the call-backs that happen when rules fire) used to derive these facts. There can be multiple ways of deriving the same facts, and SWORD currently picks an arbitrary plan (since there is no cost model yet for evaluating alternative plans).

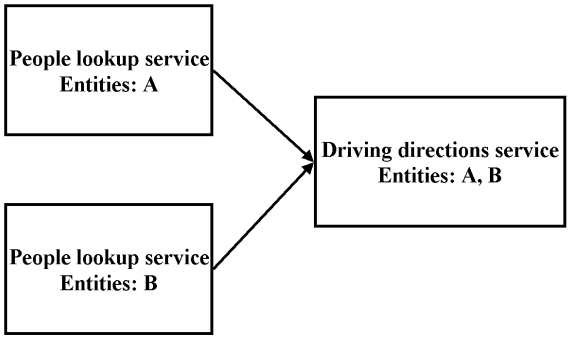

Figure 1: Simplified view of the names-to-driving-directions plan

Step 5: The constructed plan can be viewed by the developer using the GUI plan-viewer tool. Figure 1 shows a simplified view of the names-to-driving-directions plan.

3. Implications of the SWORD composition model

In this section, we discuss the strengths and weaknesses of the SWORD composition model. We sometimes contrast the SWORD model with existing data integration work to better explain the issues. However, note that:

1. **SWORD is not a model for data integration, and cannot be applied to solve the problem of data integration.**
2. **On the other hand, SWORD can be used to compose web services that data integration cannot (such as the EMail and image conversion services). Further, as we shall argue below, SWORD offers a better model for composing typical informational web services.**

3.1 Data integration

The goal of data integration is to provide a uniform interface to multiple (possibly distributed) data repositories [19, 12]. Here, we only focus on the LAV-approach [12], since it is most similar to SWORD. In the LAV approach, data sources are expressed as queries (or views) on a mediated schema consisting of *virtual relations*. User queries are also posed on the virtual relations and they need to be answered using the source relation views. In most cases, this problem of answering queries using views has proven intractable (NP-complete) [19]. Currently, the only algorithm for answering queries using views known to scale gracefully is the MiniCon algorithm [22].

While the data integration work applies (for example) to integrate a number of actual databases to create a web service, using this model for composing across web services creates several problems. First, even though they can sometimes be modeled as relations, typical informational web services allow only a limited number of pre-specified queries for various reasons (either because other queries cannot be logically answered or for administrative, engineering or privacy reasons). For example, the Yahoo people lookup service allows only looking up the address of a person by name, but not vice-versa. Similarly, the MapQuest driving directions service only allows looking up the driving directions between two addresses. If these were actual relations, they would allow other queries also such as find the names of persons living at a given address, find the destination address given the source address and driving directions, etc. (The second query cannot be logically answered since the driving directions are computed on-the-fly and there is no table listing all pairs of addresses in the world and the driving directions between them.)

Thus, most existing data integration work does not apply to web services. A section of the data integration research has addressed integration of data sources with query restrictions, which are referred to as *binding pattern limitations* [23, 9]. However, we still have the following problems *in the context of web services*:

- **Guaranteed correct: Data integration research focuses on only finding the guaranteed correct answers for a query (because much of this work was intended for enterprise data integration and query optimization using materialized views, where incorrect answers are not acceptable). However, in many cases (esp. in the context of web services), it may not be possible to find any guaranteed correct answers for a query given the available underlying services. As shall be illustrated in section 3.2, a better alternative is to return ``uncertain'' results, rather than return nothing.**
- **Abstraction mismatch: Existing data integration research attempts to create the abstraction of a relational model over the underlying sources. When the sources are web services, the problem is the following: while the relational model is very powerful in its ability to express complex queries, the underlying web services can answer only a limited number of fixed pre-specified queries as explained above. It is often very hard to create the illusion of a stronger abstraction over an underlying layer that supports weaker abstractions. In the ``relational model atop web services'' context, this mismatch manifests itself in at least two different ways:**
  1. **Many queries on the virtual relations will be rendered unanswerable since the underlying web services do not provide enough information to answer them (explained further in section 3.3). Thus, the illusion of a stronger abstraction may not really be useful, and can even be frustrating to a developer posing queries on the virtual relations.**
  2. **Even when a certain query can be answered, it may be hard to do so. As explained earlier, the problem of answering queries using views has turned intractable in most cases. Even the Minicon algorithm does not address sources with query restrictions (binding pattern limitations), and thus cannot be applied to web services.**

  Thus, in the context of web service composition (as opposed to data repository integration), it seems a weaker abstraction that more closely matches the underlying services may be better. We will examine the SWORD abstraction in section 3.3 and explain why it is better.
- **Evolvability: Specifying a virtual relation requires specifying all the attributes of the relation (since a relation is just a collection of attributes). While this is acceptable when data integration is applied for a specific application or for a specific web site, it is difficult to anticipate *all* the needed attributes for such general relations as a Person or a Movie. (Note that new web services may become available online in the future that provide information about other aspects of people and movies). In SWORD, on the other hand, all the entity attributes need not be pre-specified.**

Once again, we remind the readers that we do *not* claim that SWORD is a better model for data integration. The purpose of the above discussion is to show that while the data integration research solves the problem of integrating actual data sources, it does *not* apply well to the web service composition problem, because we have a different set of assumptions and requirements here. Also, data integration obviously does not handle composing non-query services such as conversion and email.

3.2 Liberalness

Rule-based chaining in SWORD can sometimes generate ``uncertain'' results. To understand why this happens, consider an entity X with three attributes a, b, c. Suppose the following two services are available: a service S1 that provides the b(X) given a(X) and a service S2 that provides c(X) given b(X). If it were desired to have a composite service S3 that provides c(X) given a(X), SWORD would simply chain S1 and S2 together to achieve S3. However, this can cause uncertain results.

Example: Consider the following services: personname-to-address and address-to-phone. Further, assume (as is the case) that multiple people can live at the same address and that different people living at the same address may use different phones. For the composite service name-to-phone, SWORD would just chain the name-to-address and address-to-phone services. Now, if John and Jack live at the same address but have different phones numbers, the composite service would return both their phone numbers when asked for John's phone. The reason this happens is because address neither uniquely determines the phone number nor the name.

In general, for services S1 and S2 described as above, chaining can cause uncertain results unless b $\rightarrow$ c or b $\rightarrow$ a (where $\rightarrow$ denotes a ``functional dependency'' [26] using database terminology). This is because chaining is essentially like a join and a join can be *lossy* [26] when the above property is not satisfied.

While the fact that SWORD can yield uncertain results may seem alarming at first, note that:

1. **Given information about which attributes uniquely determine which others, SWORD can detect that uncertain results may be produced when the name-address and address-phone services are chained together. While we could avoid such chainings when detected, we believe that ``some information is better than no information''. Another alternative (which SWORD does not currently implement) is to produce all results but tag the ones that are known to be uncertain.**
2. **Suppose a new name-phone service becomes available. Then, running the SWORD plan generation algorithm again for the same composite service would detect that the composite service can be realized either by chaining the name-address and address-phone services or by the new name-phone service alone. Currently, SWORD just picks one of the plans arbitrarily. In the future, we wish to generalize SWORD such that it either selects the plan that is likely to yield more certain results or selects both the plans but tags the results with certainty measures. Note that in practice, even the name-phone service can actually produce uncertain results either because of outdated information or due to other errors.**
3. **Given just the name-address and address-phone services, there is no way of exactly determining the phone number for a given name.**

The last bullet above implies that where SWORD yields uncertain results, an equivalent query on a virtual relation may yield no results at all (since there are no guaranteed correct answers here).

Example: An obvious choice for a virtual relation in the mediated schema for the name-address-phone example is the ``person'' relation: person(name,address,phone). (In reality, any reasonable person virtual relation will have several other attributes.) The natural equivalent for the composite name-phone service would be the following query (using the terminology of [23, 9]): v(n^b, p^f) :- person(*n*,*a*,*p*). However, this query would yield no results given only the name-address and address-phone services. Thus, in this example, the tradeoff is essentially getting a few phone numbers one of which could be the actual number vs. getting no results at all.

3.3 SWORD abstraction

The SWORD ``query model'' is weaker than the query languages used in traditional data integration approaches. For example, it does not allow the specification of arbitrary joins. Rather, joins can only be expressed as relationship conditions. For example, suppose there exist two services: a service that returns the director of a given movie, and a service that returns the theaters showing a given movie. Consider the following (simplified) choices for the world model (with the SWORD approach) and mediated schema (with traditional data integration approach):

```
SWORD world model:
Entities: Movie - Attributes: name, director
          Theater - Attributes: name, address
Relationships: Shows - Involved entities: Theater, Movie

Mediated schema:
Movie(name, director)
Theater(name, address)
Shows(moviename, theatername)
```

Given the above mediated schema, all relational query languages would allow posing queries involving arbitrary joins such as the following (we use the notation commonly used in data integration literature here but also provide explanations):

- **Answers(X, Z) :- Movie(X, Y), Movie(Z, Y) - find all pairs of movies with the same director**
- **Answers(X, Z) :- Theater(X, Y), Theater(Z, Y) - find all pairs of theaters located at the same address**

These queries cannot be answered given only the two available web services. The main problem is that the exported schema gives little information about which queries can actually be answered. On the other hand, these unanswerable queries cannot be posed in SWORD. (Recall that all condition input/outputs must be entity or relationship assertions only, and all data input/outputs must be entity or relationship attributes). However, note that SWORD allows the query ``find all theaters showing a given movie'', which may be posed as follows:

```
Condition Inputs: Movie(X), Theater(Y), Shows(X, Y)
Data inputs: name(X)
Data outputs: name(Y)
Condition outputs: --none--
```

Suppose a new web service becomes available that when given the name of a movie X, provides the names of all movies that were also directed by the director of X. Then, we can add a new relationship to the SWORD world model called ``Same-Director''. This can then be used to pose queries about movies directed by the same director.

In general, we believe that the weaker query model of SWORD is better suited for web services, since it results in a simple and efficient composition process, while still allowing almost all the *answerable* queries that may arise in practice.

Note that the SWORD query model can be augmented to allow arbitrary queries such as the ones above. However, we do not wish to forgo the simplicity and efficiency of the weaker query model. An interesting alternative that emerges is to support both a simple query model and a complicated one, where the simpler one is known to be more efficient, and the complicated one is only used when necessary, with the expectation that most queries using it may be unanswerable.

3.4 Rule-based plan generation

Service model vs. rule-based plan generation: While our service model presented in section 2.1 (describing a service using its condition and data inputs/outputs) is fairly general, the rule-based plan generation algorithm of section 2.3 is not as flexible. For example, by using negation, disjunction, arithmetic and function symbols, we could describe a large class of web services by their condition and data input/outputs. However, the simple rule-based plan generation algorithm would not work any more (at least not in its current form) for these services. However, there is a sufficiently interesting class of services that can be modeled

subject to the current SWORD constraints. In the future, as we add services that can't be modeled with these constraints, we will need to incrementally generalize the plan generation algorithm.

Plan generation can fail: Note that one of the following can happen in the plan generation process:

1. **A composite service cannot be expressed using the SWORD model. (Since SWORD can currently only compose a class of web services.)**
2. **A composite service can be expressed using the SWORD model, but SWORD fails to generate a plan for the target composite service because there is no sequence of services that can be used to achieve the composite service.**
3. **SWORD succeeds in generating a plan, but after viewing the plan, the developer finds it unsuitable for some reason.**

In the event one of the above happens (or is anticipated in advance), the developer has to resort to other techniques to create and deploy the composite service. (For example, create the composite service manually, or use other composition tools and techniques).

Semantic issues in rule-base: When a new base service is added (i.e., modeled) by a base service modeler, the rule for the new service gets added to the ``rule-base'', and becomes available for all future compositions. However, an important issue is consistency across rules in the rule-base built over a period of time. The rules must consistently use the same facts for representing the same concepts. In SWORD, this must be ensured by the base service modelers.

Note that an ``improper'' choice of an initial set of entities and relationships can hinder the graceful evolution of the rule-base. A related important issue is the standardization of the rules across service providers or across composition sites. If such standardization becomes possible in the future (perhaps due to deployment of standards such as RDF [27], DAML [14], etc), SWORD could prove more effective. On the other hand, SWORD is useful even without such standardization. (Base service modelers at each ``composition site'' write their own wrappers and rules for each web service using the world model of that composition site.)

Efficiency: The rule-based system we use is based on the Rete [11] algorithm, and takes only $O(rfp)$ time per iteration where r is the number of rules, f is the number of facts and p is the number of patterns on the LHS. Thus, the overall time for forward chaining is $O(rF^2p)$ where F is the total number of facts (initial and derived). (In fact, for informational services, the only facts that can be derived are the known facts corresponding to the attributes of the entities involved in the composite service inputs.) Also, rule-based systems are well-understood and several practical implementations are available.

4. Prototype in Action

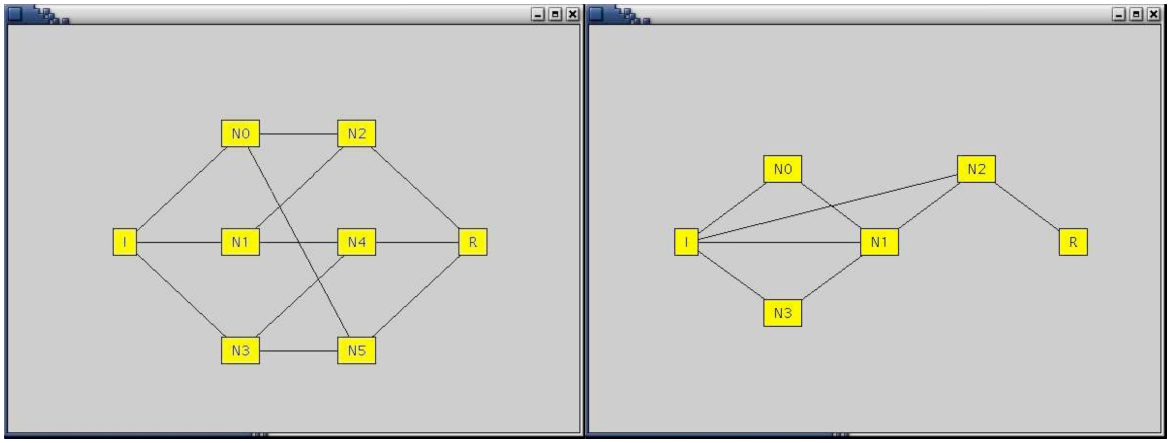

Figure 2: Two composite service plans as viewed by the SWORD plan viewer tool. Clicking on any node gives information about the service for that node (such as the information shown in figure 1). Plan 1 corresponds to the movie, restaurant trip planner composite service. Given the name of a person, the city where (s)he lives in, a movie name and a cuisine, this service finds the driving directions from the person's house to a restaurant serving the given cuisine, from the restaurant to a theater showing the given movie, and finally from the theater back to the person's house. Plan 2 shows the ``stock notification composite service''. This composite service looks up the current quotes for two symbols, constructs a stock notification message with the looked up quotes and finally emails the constructed notification message.

We have implemented a proof-of-concept prototype of the plan generation algorithm and the plan viewer tool described in section 2, and an execution system that can instantiate and execute the plans. We have used SWORD to create seven composite services from a set of ten base web services.

Figure 2 shows two different plans *as viewed by the plan viewer*. (Recall that figure 1 was a simplified illustration.)

We now demonstrate the movie, restaurant trip planner composite service (plan 1) in action. Figure 3(a) shows the interface to the trip planner as seen by an end-user on the web. As seen in the figure, the trip planner service prompts the user to enter the person name and the city, the movie name and a cuisine. The user enters this information and submits the form. As shown in the figure, say ``John White'' is the entered person name, ``Hardball'' is the entered movie name and ``Chinese'' is the entered cuisine. Suppose multiple matches are found for the name ``John White'' and the given city. Then, the trip planner service prompts the user with the matches found the requests the user to select the appropriate ``John White'' (figure 3(b)). The trip planner then locates the restaurants serving Chinese cuisine near the entered zip code and prompts the user to select one of the found restaurants (figure 3(c)). Similarly, the service finds the theaters showing the movie ``Hardball'' and prompts the user to select one of them (not shown here). Finally, the service finds the three pairs of driving directions for the trip and displays them to the user.

A composite service need not necessarily be interactive. For example, the ``stock notification composite service'' runs without any interaction with the user. Even the trip planner service can be run such that it does not prompt the user for selecting the restaurant etc., but (for example) just selects the first match and proceeds.

In the next section, we discuss plan instantiation and execution (the mechanism that translates the plans shown in figure 2 to the actions shown in figure 3). The run time mechanisms also compensate for some of the limitations of the current service representation and composition model.

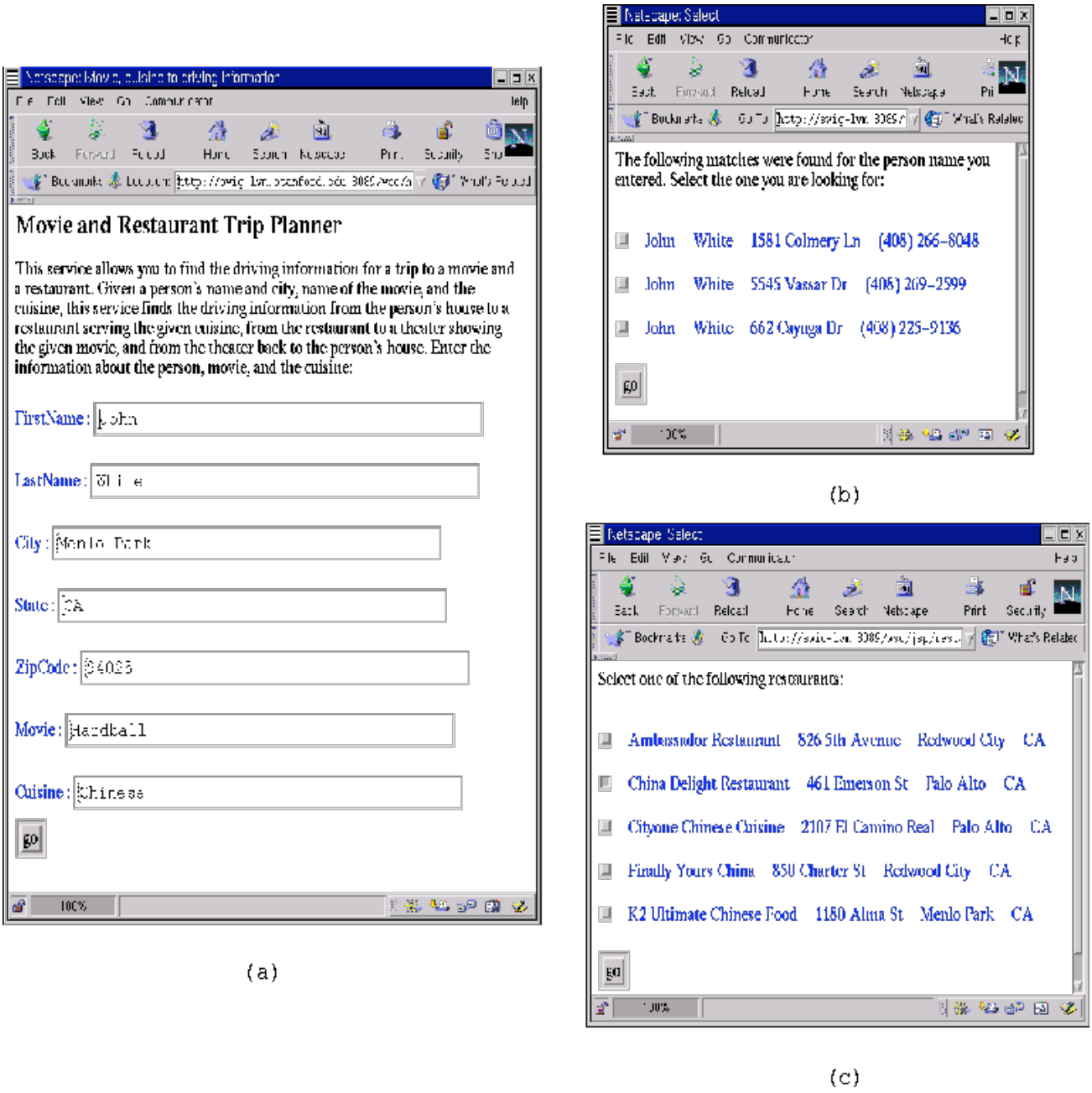

Figure 3: Screen-shots of the trip planner composite service in action

5. Execution

If a plan viewed by the plan viewer is found to be appropriate, the developer can request that a persistent representation of the plan be generated. This representation is used for execution whenever the composite service is invoked by an end-user on the web.

5.1 Persistent XML plan representations

The persistent XML representation for a plan lists the services that need to be invoked, the order in which they need be invoked and the *flow of entities* between them. Further, for each service, the plan representation includes the run time information - the *script* and the *filter*. This information is obtained from the respective service model descriptions. (Recall from section 2 that in addition to the condition and data input/outputs, each service model also includes the run time information. The script and the filter constitute the most important pieces of run time information for any service.)

The script points to a WebL [17] (a language custom designed for web scripting) script. We have *not* focussed on the problem of interacting with the web services and extracting the results from them. Apart from WebL, several other research and industry efforts are addressing this problem [28, 29], and SWORD can easily use these alternatives if widely deployed.

Once the XML representation is generated, the developer creates a suitable custom web interface for the composite web service, which points to the appropriate plan (such as the form shown in figure 3(a)).

5.2 Entity-flow

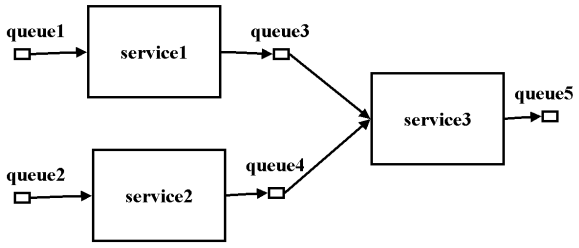

Figure 4: SWORD execution model consists of a graph of service operators connected by queues

In the following description, we will use the names-to-driving-directions composite service (shown in figure 1) to explain how the execution works. SWORD run time is built upon a data-flow engine called Paths [16]. The run time instantiates an (appropriately interconnected) *operator* for every service listed in the XML representation of the plan. Thus, at run time, we have a graph of operators interconnected by queues. The execution graph for the plan shown in figure 1 is shown in figure 4. The execution model can be thought of as a flow of entities through this graph. For example, suppose the composite service is invoked (from the custom created web interface) with the following data:

```
Person A:
firstname: John
lastname: Doe
city: Los Angeles
state: CA

Person B:
firstname: Joe
lastname: Shmoe
city: New York
state: NY
```

The run time first creates a Person entity each for John Doe and Joe Shmoe. Each entity is identified by a unique identifier.

Suppose the entity for John Doe has the identifier ``A101'' and the entity for Joe Shmoe has the identifier ``B200''. Then, the system sets the data inputs (if any) for these entities as follows:

```
A101.firstname = ``John''
A101.lastname = ``Doe'
A101.city = ``Los Angeles''
A101.state = ``CA''

B200.firstname = ``Joe''
B200.lastname = ``Shmoe'
B200.city = ``New York''
B200.state = ``CA''
```

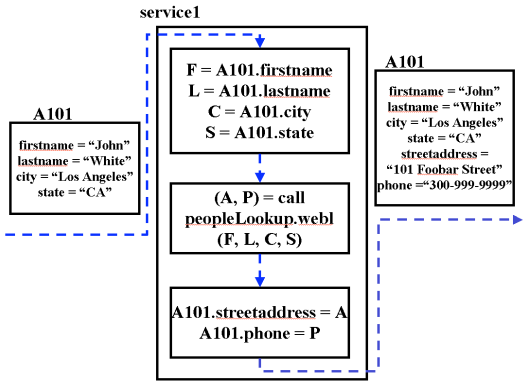

Figure 5: Execution of an individual service at an operator illustrated

Then, the system places the entity A101 in 'queue1' and the entity B200 in 'queue2'. As shown in figure 5, the operator corresponding to 'service1' picks up the entity A101, extracts the firstname, lastname, city and state attributes and launches the Webl script ``peopleLookup.webl'' with these four arguments. When this script returns, the operator extracts its results and assigns them to A101 as shown below (and in the figure):

```
A101.streetaddress = ``101 FooBar street''
A101.lastname = ``300-999-9999''
```

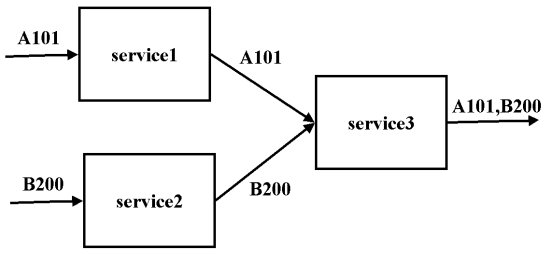

Figure 6: Flow of entities through the execution graph

After this assignment, the operator places A101 in 'queue3' where it is picked by the operator corresponding to 'service3'. The execution at the other operators proceeds similarly. Figure 6 shows the flow of entities through the graph. When the execution completes, the system removes the entity pair (A101,B200) from 'queue5' and extracts the 'drivingdirections' attribute of this pair. Finally, the system invokes a result Java server page (which JSP to invoke is also contained in the plan) with the extracted driving directions as arguments. This Java server page formats the results appropriately and displays them to the user.

5.3 Filters

As observed in the demonstration of the trip planner composite service in section 4, base services may produce multiple outputs either expectedly (such as multiple Chinese restaurants in the same area) or unexpectedly (such as multiple matches for ``John White''). Both these cases cases are handled using a general *filter* mechanism in SWORD. (Thus, the web pages for user interaction shown in figures 3(b) and figures 3(c) were actually generated by filters). A filter may optionally be specified for each service in the service model. If a filter is specified for a service, the run time invokes the filter with the output of the service. When the filter returns, the run time extracts the output of the filter and places it in the output queue for the service. As viewed by the SWORD run time, the filter is a ``black-box'', and it can do anything: seek user intervention and modify the output, or perform some computation (eg. sorting) and modify the output. Since Java server pages can perform both arbitrary computation and provide a user interface, we have chosen them as the medium to implement the filter mechanism.

Note that the web pages for user interaction are difficult to generate automatically, especially since they may need be customized to the individual service, composite service, and/or the web site hosting the composite service. In addition to enabling user interactions, filters can sometimes compensate for the absence of arithmetic/function symbols in the composition logic, since they can perform arbitrary computation (such as sorting, searching, etc).

5.4 Performance

We have taken preliminary performance measurements for both plan generation and execution. Generating the movie, restaurant trip plan of figure 2(a) took 0.98 seconds, while generating the plan in figure 2(b) took 0.68 seconds. In both cases, the forward chaining itself took only about 50 milliseconds. (Some of the overhead is due to HTTP latency - the plan generator is also web-based). Executing the movie, restaurant trip plan in a non-interactive mode require 15.7 seconds. The run time overhead added by SWORD was 1.7 seconds, while the rest of the time was spent making the HTTP requests to the web services and extracting the results. The SWORD run time is not particularly optimized and we believe the run time overhead added by SWORD is acceptable.

5.5 Other practical issues

Multiple entities selected: Suppose the people lookup service returns two matches for ``John Doe''. Also, assume that the end-user cannot decide between the two when asked to do so by the filter. The filter then returns a pair of street addresses and phone numbers (say [``101 FooBar street'', 300-999-9999], [``490 BarBaz street'', 300-888-8888]). When this happens, the operator ``clones'' the entity A101 to create another Person entity (say with identifier A102) and sets the following:

```
A101.streetaddress = ``101 FooBar street''
A101.phone = ``300-999-9999''
A102.streetaddress = ``490 BarBaz street''
A102.phone = ``300-888-8888''
```

Note that the other attributes of A101 and A102 will have identical values since A102 was created by cloning A101. The

following 'service3' operator then finds the driving directions for both the combinations [A101, B200] and [A102, B200].

Optional inputs: Many web services take multiple inputs, not all of which are mandatory. For example, the people lookup service may need only the lastname, city, and state as the mandatory inputs while the firstname may be an optional input. In these cases, we currently only use the mandatory inputs while specifying the service model. However, we allow for using optional inputs at run time wherever possible. While invoking any service at run time, we check if the values for optional inputs are known and include them in the request sent to the web service. This strategy has helped us make it possible to use optional inputs where available while keeping the composition process simpler. However, sometimes it is desirable to derive optional inputs using other services and include them while making the request to a particular service. This does not currently happen in SWORD.

Due to lack of space, we do not discuss other issues such as missing outputs and service invocation errors.

6. Discussion

In this section, we discuss the benefits of building composite web services in an automated fashion. In addition, we also discuss an alternative mechanism for web service composition.

6.1 Benefits of automating building composite services

Building composite services with an automated/semi-automated tool like SWORD obviously requires less effort than creating the service manually. This is especially important because composite service creation is not necessarily a one-time effort. Rather, the composition may need to change as the underlying services evolve, new services become available and old ones cease to exist. We also have the following ``non-obvious'' advantages due to automation:

1. ***Desirable plans:*** **Given an algorithm to evaluate the ``desirability'' of a composition (based on metrics such as efficiency, accuracy of results produced, etc), a tool such as SWORD could automatically consider several compositions and select the most desirable ones from them.**
2. ***Scalability:*** **As the number of web services grows, it may not be feasible for a developer to consider all the possible compositions to achieve a particular composite service. When combined with automatic discovery [8], an automated composition tool becomes even more powerful.**

Also, while SWORD is currently intended as a toolkit for creating composite web services, many of the ideas could also be used for creating applications and agents intended for end-users.

6.2 Online vs offline planning

SWORD currently does planning only at composition time and *not* at run time. Both online and offline planning have their own advantages. The primary advantage of offline planning in SWORD is predictability at run time. Once the developer verifies the generated plan, she can be (relatively) confident of the run time behavior. Another advantage is efficiency, since no planning overhead is incurred at run time for every request or for every entity flowing through the execution graph. Also, note that multiple alternative plans could possibly be generated at composition time itself to deal with different run time possibilities such as missing outputs, service invocation errors, etc.

7. Related Work

As explained earlier, SWORD does *not* rely on, but could benefit from, the deployment of emerging standards such as

SOAP [28], WSDL [29], UDDI [8] and DAML [14]. The semantic web initiative [3, 14, 10, 13, 20] can be broadly subdivided into two categories.

- **Researchers are developing standards and markup languages (with varying expressive power such as RDF [27, 5], DAML [14], DAML/OIL [10]) which may be used to create a web of services whose effects and outputs are encoded unambiguously.**
- **Agent technologies [13, 20] will be developed that will use the markup exported by services to achieve the end-user's needs.**

In principle, SWORD belongs to the latter category. However, SWORD does *not* rely on the actual deployment of any specific semantic markup language by service providers. Further, since it is unclear as to how rich a semantic markup language would get adopted in practice (if at all), we believe that the simple world model of SWORD is advantageous from a deployment perspective.

Research/industry efforts are also beginning to address web service discovery [8, 25]. Discovery and composition are complementary technologies and each could benefit from the other.

Much work has been done in the databases and AI communities on the problems of integrating relational databases [19, 12], as well as semi-structured and XML data [21, 2]. As already explained in section 3, SWORD has a different set of goals from data integration. In that section, we also distinguished SWORD from the relevant aspects of the data integration research. Research efforts have also addressed searching/crawling based technologies for extracting information from static web pages as well as the web's link structure. These technologies include question answering [18, 1] and topic distillation [7, 6, 4].

8. Conclusions

In this paper, we described SWORD, a toolset that allows developers to quickly compose existing web services to realize new, composite web services. SWORD provides a specific data-point in the obvious expressiveness-complexity-efficiency tradeoff that exists in web service composition. The key aspects of SWORD are summarized below:

- **A simple and efficient composition model for a class of web services (including informational web services and others).**
- **An execution model that can instantiate and execute generated plans. The execution model includes a filter mechanism for (customizable) user interactions and any other necessary intermediate computation.**
- **An implemented prototype that demonstrates the feasibility of SWORD even for web services as they exist now.**

Acknowledgments: We thank Mike Genesereth, Prasenjit Mitra, Mayank Bawa and Honglei Zeng for reviewing earlier drafts of this paper and providing valuable feedback that helped improve this draft. We are also grateful to Michael Michael, Jairam Ranganathan, Emre Kiciman and Laurence Melloul for help with various parts of the implementation.

References

1

Eugene Agichtein, Steve Lawrence, and Luis Gravano. Learning search engine specific query transformations for question answering. In *Tenth International World Wide Web Conference (WWW-10)*, Hong Kong, May 2001.

2

Chaitanya K. Baru, Amarnath Gupta, Bertram Ludäscher, Richard Marciano, Yannis Papakonstantinou, Pavel Velikhov, and Vincent Chu. XML-Based Information Mediation with MIX. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 597-599. ACM Press, 1999.

3

Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, May 2001.

4

Krishna Bharat and Monika R. Henzinger. Improved Algorithms for Topic Distillation in a Hyperlinked Environment. In *SIGIR '98: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 24-28 1998, Melbourne, Austrailia*. ACM, 1998.

5

Jeen Broekstra, Michel C. A. Klein, Stefan Decker, Dieter Fensel, Frank van Harmelen, and Ian Horrocks. Enabling knowledge representation on the Web by extending RDF schema. In *Tenth International World Wide Web Conference (WWW-10)*, Hong Kong, May 2001.

6

Soumen Chakrabarti. Integrating the Document Object Model with hyperlinks for enhanced topic distillation and information extraction. In *Tenth International World Wide Web Conference (WWW-10)*, Hong Kong, May 2001.

7

Soumen Chakrabarti, Byron E. Dom, David Gibson, Jon Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Mining the Link Structure of the World Wide Web. *IEEE Computer*, 32(8):60-67, 1999.

8

UDDI community. Universal Description, Discovery, and Integration. http://www.uddi.org.

9

Oliver M. Duschka, Michael R. Genesereth, and Alon Y. Levy. Recursive query plans for data integration. *Journal of Logic Programming, special issue on Logic Based HeterogeneousInformation Systems*, 43(1):49-73, 2000.

10

Dieter Fensel, Ian Horrocks, Frank van Harmelen, Deborah L. McGuinness, and Peter F. Pate;-Schneider. OIL: An Ontology Infrastructure for the Semantic Web. *IEEE Intelligent Systems (Special Issue on Semantic Web)*, 16(2), 2001.

11

C. L. Forgy. RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem. *aij*, 19:17-37, 1982.

12

Alon Halevy. Logic-based techniques in data integration. *Logic Based Artificial Intelligence*, 2000.

http://www.cs.washington.edu/homes/alon/site/files/levy-di00.ps.

13

James Hendler. Agents and the Semantic Web. *IEEE Intelligent Systems (Special Issue on Semantic Web)*, 16(2):30-37, 2001.

14

James Hendler and Deborah L. McGuiness. DARPA Agent Markup Language. *IEEE Intelligent Systems*, 15(6):67-73, 2001.

15

Java Expert System Shell (Jess). http://herzberg.ca.sandia.gov/jess/.

16

Emre Kiciman and Armando Fox. Using dynamic mediation to integrate COTS entities in a ubiquitous computing environment. In *Handheld and Ubiquitous Computing (HUC 2000), First International Symposium*, September 2000.

17

Thomas Kistler and Hannes Marais. WebL: A Programming Language for the Web. In *Seventh International World Wide Web Conference (WWW-7)*, Brisbane, Australia, April 1998.

18

Cody C. T. Kwok, Oren Etzioni, and Daniel S. Weld. Scaling question answering to the Web. In *Tenth International World Wide Web Conference (WWW-10)*, Hong Kong, May 2001.

19

Alon Y. Levy. Answering Queries using Views: A Survey. http://www.cs.washington.edu/homes/alon/views.ps, 1999.

20

Sheila A. Mcllraith, Tran Cao Son, and Honglei Zeng. Semantic Web Services. *IEEE Intelligent Systems (Special Issue on Semantic Web)*, 16(2):46-53, 2001.

21

Yannis Papakonstantinou and Vasilis Vassalos. Query Rewriting for Semistructured Data. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 455-466. ACM Press, 1999.

22

Rachel Pottinger and Alon Y. Levy. A Scalable Algorithm for Answering Queries Using Views. In *Proceedings of the International Conference on Very Large Data Bases (VLDB) 2000*, Cairo, Egypt, September 2000.

23

Anand Rajaraman, Shuky Sagiv, and Jeff Ullman. Answering Queries Using Templates With Binding Patterns. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California*. ACM Press, 1995.

24

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

25

Katia P. Sycara, Matthias Klusch, Seth Widoff, and Jianguo Lu. Dynamic Service Matchmaking Among Agents in Open Information Environments. *SIGMOD Record*, 28(1):47-53, 1999.

26

Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems*. Prentice-Hall, 1997.

27

W3C. RDF specifications. http://www.w3.org/RDF/.

28

W3C. SOAP specification. http://www.w3.org/TR/SOAP/.

29

W3C. WSDL specification. http://www.w3.org/TR/WSDL/.