Tools for Composite Web Services: A Short Overview**

Richard Hull
Bell Labs Research
Lucent Technologies
hull@lucent.com

Jianwen Su*
Department of Computer Science
UC Santa Barbara
su@cs.ucsb.edu

ABSTRACT

Web services technologies enable flexible and dynamic interoperation of autonomous software and information systems. A central challenge is the development of modeling techniques and tools for enabling the (semi-)automatic composition and analysis of these services, taking into account their semantic and behavioral properties. This paper presents an overview of the fundamental assumptions and concepts underlying current work on service composition, and provides a sampling of key results in the area. It also provides a brief tour of several composition models including semantic web services, the "Roman" model, and the Mealy/conversation model.

1. INTRODUCTION

The web services paradigm promises to enable rich, flexible, and dynamic interoperation of highly distributed and heterogeneous web-hosted services. Substantial progress has already been made towards this goal (e.g., emerging standards such as SOAP, WSDL, BPEL) and industrial technology (e.g., IBM's WebSphere Toolkit, Sun's Open Net Environment and JiniTM Network technology, Microsoft's .Net and Novell's One Net initiatives, HP's e-speak, BEA's WebLogic Integration). Several research efforts are already underway that build on or take advantage of the paradigm, including the DAML-S/OWL-S program [14, 36, 24], and automata-based models for web services [8, 26, 7]. But there is still a long way to go, especially given the ostensible long-term goal of enabling the *automated discovery*, *composition*, *enactment*, and *monitoring* of collections of web services working to achieve a specified objective. A fundamental challenge concerning the design and analysis of composite web services is to develop necessary techniques and tools to handle the novel aspects of the web services paradigm.

The challenge raises a variety of questions, several of which are relevant for the database research community. Some of the questions are: What is the right way to model web services and their compositions? What is the right way to query them in order to support automated composition and analysis algorithms? And how can the data management aspects of composite web services be incorporated into current web services standards? This paper attempts to provide the groundwork needed to address these questions, by describing emerging frameworks for studying composite services, and identifying emerging tools and techniques for both automated design and analysis of composite web services.

The focus of this short survey is on the foundations of composition and related issues. From this perspective, it is worthwhile to understand the overall process of designing composite services. Fig. 1 shows the key elements in the

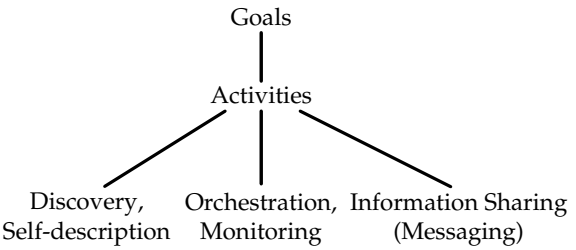

Figure 1: Anatomy of Web Service Composition

typical design process. It is not surprising that the design process is similar to planning studied in artificial intelligence (e.g., [42]). In this case, activities that may contribute to accomplish the overall goal need be discovered and orchestrated. The orchestration has to be constrained by the flow of available information, which is generally agreed to be in the form of *messages*. Note that monitoring is closely related to orchestration; specific models of the latter may determine how and to some degree what to monitor. Interestingly, the design process embodies elements from all four task groups listed above.

Fig. 2 illustrates three dimensions that "measure" Web service description/composition models. The *component service complexity* dimension indicates the information capacity of the languages and model in representing a service. In this dimension, OWL-S is low since it describes only the input and output. On the other hand, WSDL captures richer structural information with XML Schema and uses messages for input and outputs, and other models including BPEL, CSP [29] and π-Calculus [34], the Mealy model [8], the Roman model [7], and BPML also model service states and message or activity sequences. Clearly, the ability of assembling individual services together in this "cluster" of models make them high along the *glue language complexity* dimension. The third dimension is the ability to describe "semantics". OWL-S can describe the properties on the input and output of an operation, and also specify how the service interacts with an abstract model of the "real world", which distinguishes it from the other models.

*Supported in part by NSF grant IIS-0101134.
**Database Principles Column.
Column editor: Leonid Libkin, Department of Computer Science, University of Toronto, Toronto, Ontario M5S 3H5, Canada. E-mail: libkin@cs.toronto.edu.

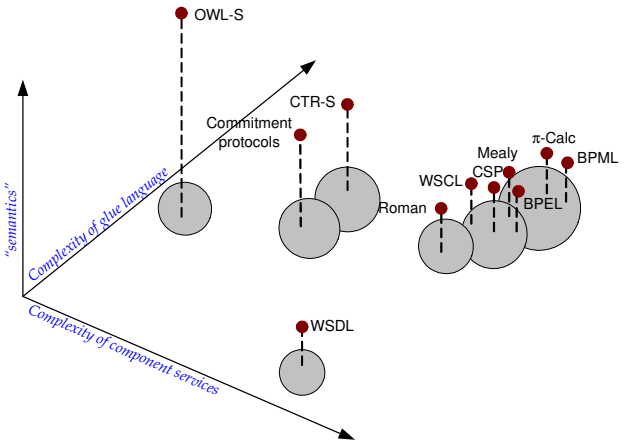

Figure 2: Anatomy of Web Service Composition

This paper is organized as follows. Section 2 gives a short overview of several standards related to Web services and composition. Section 3 is intended to provide some high level of key aspects web service composition and a discussion on models that have been studied in the context. Section 4 briefly summarizes techniques and approaches for analyzing Web services. Section 5 concludes the paper.

2. WEB SERVICES AND STANDARDS

A good starting point for understanding the web services paradigm is to consider the stated goals, as found in the literature and the standards communities. The basic motivation of standards such as SOAP and WSDL is to allow a high degree of flexibility in combining web services to create more complex ones, often in a dynamic fashion. The current dream behind UDDI is to enable both manual and automated discovery of web services, and to facilitate the construction of composite web services. Building on these, the BPEL standard provides the basis for manually specifying composite web services using a procedural language that coordinates the activities of other web services.

Much more ambitious goals are espoused by the OWL-S coalition [14] and more broadly the semantic web services community (e.g., [15]). These goals are to provide machine-readable descriptions of web services, which will enable automated discovery, negotiation with, composition, enactment, and monitoring of web services. OWL-S is an ontology language for describing web services, in terms of their inputs, outputs, preconditions and (possibly conditional) effects, and of their process model. Importantly, OWL-S provides a formal mechanism for modeling the notion of the state of the "real world", and describing how atomic web services impact that state over time. (This is described in more detail in Subsection 3.1 below.) OWL-S also provides a *grounding*, which provides mechanisms for mapping an OWL-S specification into a WSDL specification (e.g., see [40]).

A kind of middle ground is also emerging, which provides abstract "signatures" of web services that are richer than WSDL but retain a declarative flavor. Most popular here is the use of automata-based descriptions of permitted sequencing patterns of the web services, with a focus on either activities performed [7] or messages passed [26].

The underlying structure for the web services paradigm will most likely be guided by already established standards and practices. Some of the current standards are illustrated by the layered structure shown in Figure 3. Briefly, web services interact by passing XML data, with types specified using XML Schema. SOAP can be used as the communication protocol, and the i/o signatures for web services are given by WSDL. All of these can be defined before binding web services to each other. Behavioral descriptions of web services can be defined using higher level standards such as BPEL, WSCDL, BPML, DAML-S, etc.

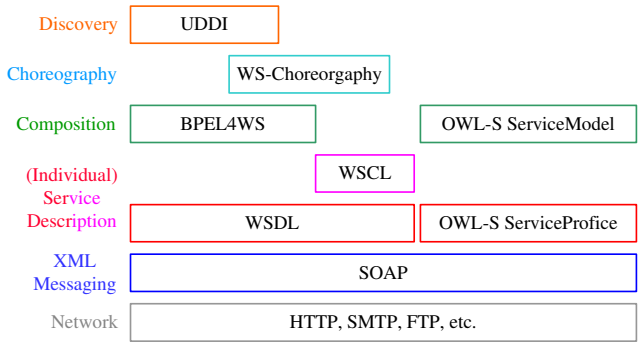

Figure 3: Web Service Standards Stack

In Figure 3, XML messaging and Network layers provide the foundation for interoperations or interactions between services. The starting point for service descriptions is the W3Cs Web Service Description Language (WSDL) [50]. WSDL describes a Web service via its set of visible operations. They can be thought of as message endpoints, or the set of messages that it can send and receive. WSDL specifies on the one hand "reactive" operations in which a message is received by the service. If the reactive operation is declared as "one-way", then it does not return a response, otherwise it is a "request-response" operation, and the return type is also declared. It also describes proactive operations that send out messages from the service. "Notification" operations send out messages without waiting for a response, while "solicit-response" operations block waiting for a response, with the response type being specified with the operation. The receive and response types of the operations are mapped onto concrete XML Schema types to be used in messages. WSDL can thus be seen as an extension of traditional input/output signatures in programming languages and distributed computing to a peer-to-peer setting. A service is viewed both as a server (via its reactive operations) a client (via its proactive operations).

WSDL models services that are essentially stateless. (WSDL 2.0 incorporates a limited notion of state, since it enables specification of certain message patterns that a service should satisfy.) While it is clear that the complete *internal* state of a running service is not expected to be known, it is also commonly accepted that *some* execution states need be visible either due to the fact that they are *observable* or the necessity (e.g., to interact correctly with other services). The Web Service Conversation Language (WSCL) [49] proposal is an interesting approach in defining the overall input and output message sequences for one service using essentially a finite state automaton (FSA) over the alphabet of message types. The FSA is called a *conversation*.

Note that WSCL complements extremely well with WSDL. A WSDL description along with a WSCL conver-

sation of a service provide a rather rich semantics about the service while keeping the internal implementation/execution encapsulated. For this reason, they are combined into a layer named "individual service description" in the stack in Fig. 3.

While WSDL and WSCL can define Web services, clearly there is a need for languages in the corresponding level of abstraction to compose services. BPEL [12] was developed as a language both for programming Web services and for specification of Web services (e.g., in legacy systems). For example, [26] identified two topologies for service composition: "peer-to-peer", and "hub-and-spoke" with a mediator. Mediators play the role of coordinating the activities of other web services. A primary goal of BPEL is provide a language for specifying the behavior of mediator services, rather than for general-purpose programming.

In contrast to specifying individual services that BPEL provides, WS-Choreography (WSCDL) [48] attempts to specify globally how different component services should behave. WSCDL emphasizes on the "choreography" aspects: the roles of the participating services, information that being passed between services, and channels that enable the information flow.

At the top of the standards stock is the UDDI (Version 3 was recently adopted as an OASIS standard). UDDI allows services to be registered with a *registry*. Services in a registry can be searched with querying abilities provided by the standard.

3. FOUNDATIONS FOR SERVICE COMPOSITION: A SAMPLING

Web service *composition* addresses the situation when a client request cannot be satisfied by a single pre-existing service, but can be satisfied by suitably combining some available, pre-existing services. Reference [25, 1] identifies two aspects of composition: *composition synthesis* is concerned with synthesizing a specification of how to coordinate the component services to fulfill the client request; and *orchestration*, is concerned with how to actually achieve the coordination among services, by executing the specification produced by the composition synthesis and by suitably supervising and monitoring that execution. The focus here is on composition synthesis and related issues.

The theoretical study of (automatic) composition synthesis for web services is still in its infancy. The models underlying this work on automatic composition have roots in AI, logic, situation calculi, transition systems, and automata theory, and thus rely on a variety of philosophical bases. The discussion here is not intended to be comprehensive, but rather highlights some of the most important foundations and results that might be most interesting to the database community.

The initial results on automatic composition considered here are grouped around three different models for web services, which we identify here as (a) *OWL-S* (e.g., [14]), which includes a rich model of atomic services and how they interact with an abstraction of the "real world", (b) *Roman* (e.g., [7]), which uses a very abstract notion of atomic service in a finite-state automata framework for describing process flows, and (c) (*message-based*) *Mealy machine* (e.g., [8]), which focuses on (message-based) "behavioral signatures" of services, and again using a finite-state automata framework for process flow.

The work on the different models has centered around three different aspects of composition. To describe these, we first establish some vocabulary. When a family of web services interact, the overall topology may be, speaking informally, *mediator-based* or *peer-to-peer*. By mediator-based we mean that there is one web service, called the *mediator*, which has the specialized role of controlling the operation and interaction of the other services; the other services are called *component* services. In a peer-to-peer framework, each participating service is called a *component* service.

It is also useful to distinguish between compositions that are intended for *single* or *multiple* use. In the former case, a composition algorithm is invoked each time that a client identifies a desired goal service, similar to typical scenarios on AI planning. In the latter case, the goal established for the composition algorithm is to produce a (composite) service that can be run multiple times, in support of the same or different clients.

The three families of initial results are now described; more details are given below.

Synthesis of mediator from atomic component services. The first category of results, which has been used successfully [38] with OWL-S, is focused on building a workflow schema (e.g., flowchart, Petri net, composite OWL-S service, ...) that invokes atomic services in order to achieve a specified goal within specified constraints. This workflow schema would typically be enforced by a mediator, and in practice might be specified using BPEL. This work tends to focus on the single-use case.

Selection of multi-step components and synthesis of mediator. In this work, initiated with the Roman model, the component services are generally non-atomic, and have process flow specified using a transition system or finite-state automata. Unlike the OWL-S work, however, the atomic services inside the component services are very abstract. As described below, seminal results here focus on the construction (if one exists) of a specialized kind of multi-use mediator that achieves a desired goal behavior.

Synthesis of component services in peer-to-peer setting. The results here focus on message-passing Mealy machines. In this context, a *composition schema* is defined as a template that component web services can be plugged into. The goal behavior is specified as a family of permitted message exchange sequences, or *conversations*, that should be realized by the system. Key results here characterize when a conversation language can be realized, and synthesize component services in that case. As detailed below, these results can be used to help with practical composition, and they provide an approach to map "global" choreography constraints onto "local" constraints concerning the message sequencing behaviors of the component services.

In Subsection 3.1 below we describe the OWL-S model and results in more detail. Special emphasis is placed on the model, and a natural bridge from the AI foundations that OWL-S relies on to a formal basis that essentially builds on a relational database framework. In Subsection 3.2 we describe the Roman model and results, and in Subsection 3.3 we describe the model and results on message passing and Mealy machines. Finally, Subsection 3.4 presents a brief overview of recent work on web service models and results

that combine the fundamental aspects of the above models.

3.1 OWL-S: Model and composition

We now consider salient aspects of the OWL-S model and some of the key composition results obtained. In our presentation we do not follow exactly the original formulation [14], but instead adopt the approach recently introduced by work on the First-order Logic Ontology for Web Services (FLOWS) [9]. FLOWS provides the basis for the first-order logic component of the recently released Semantic Web Services Ontology (SWSO) [15]. We use the FLOWS framework here, as it is somewhat closer to a relational database formulation than the underpinnings originally used by OWL-S.

A key contribution of OWL-S is the explicit modeling of how web services in OWL-S interact with the "real world". The basic building block of the OWL-S process model is the notion of "process"; this includes both atomic and composite processes. OWL-S processes are specified to have inputs, outputs, pre-conditions, and conditional effects (IOPEs). The IOPEs of two example OWL-S atomic processes taken from a stock broker domain are informally sketched now.

```
select_stock
 input: stock_name, quantity
 output: price, reservation_id
 pre-condition: the quantity of stock is
                available for sale
 conditional effects:
   if true, then modify world state to
   reflect the fact that this stock
   is being held for sale

purchase_stock
 input: reservation_id, billable_account
 output: confirmation_id
 pre-condition: reservation_id is valid
                and has not expired
 conditional effects:
   if enough money in billable_account then
      transfer of $$ to stock owner and
      transfer of stock to buyer
   if not enough money, then
   make reservation_id void
```

A client would use `select_stock` in order to check the availability of a given quantity of some stock, and if it is available, then reserve that for purchase. Speaking intuitively, successful execution of the service would result a commitment that this stock will be held for purchase (e.g., for 10 minutes). After a successful reservation of that sort, the client might invoke `purchase_stock` to actually make the purchase, using some bank account to pay for it. `purchase_stock` will execute as long as the reservation is valid, but the outcome of this execution depends on whether there are sufficient funds in the account. A composite OWL-S service might be created by combining these two atomic processes using the sequence construct; in this case the IOPE of this composite process can be inferred from the IOPEs of the atomic processes.

To provide a formal basis for this interaction with the real world, OWL-S takes the approach of the Situation Calculus [41], a logic-based formalism which explicitly models the fact that over time different "situations" or world states will arise. To briefly illustrate this here we follow FLOWS, and use the Process Specification Language (PSL) [43, 23], a recent ISO standard which among other things incorporates core aspects of the Situation Calculi. PSL is a first-order logic ontology for describing the core elements of processes. PSL is layered, with PSL-Core at the base and many extensions. PSL-Outercore is a natural family of extensions above PSL-Core, useful for modeling processes. FLOWS is a family of extensions on top of PSL-OuterCore.

PSL-OuterCore provides first-order predicates that can be used to describe the basic building blocks of processes and there execution. To give the flavor, we mention a few highlights. Fundamental is the class of *activity*; an activity can be viewed as a process template, and there may be zero or more *occurrences* of an activity (corresponding to instances of a process or enactments of a workflow). The binary predicate `occurrence_of`(occ, a) is used to specify that occ is an occurrence of activity a. There are *complex* activities, and the binary `subactivity` predicate is used to specify sub-activities of an activity. There are also *atomic* activities, such that, intuitively, the execution of an occurrence of an atomic activity is "atomic" in the typical database sense. A notion of time points in a discrete linear ordering is included; occurrences have a begin time and an end time.

In a typical usage of PSL (FLOWS), an application domain is created by combining the PSL-OuterCore (FLOWS-Core) axioms with domain-specific predicates and sentences to form a (first order logic) theory. The models of this theory can be thought of as a tree or forest, whose nodes correspond to occurrences of (atomic) activities, and with edges from one occurrence to another if they follow each other with no intervening occurrence. A path in this tree (forest) can be viewed as one possible sequence of steps in the execution of the overall system.

Speaking loosely, each sentence in an application domain theory can be viewed as a *constraint* or restriction on the models (in the sense of mathematical logic) that satisfy the theory. In particular, and following the spirit of Golog [41] and similar languages, even process model constructs such as `while` or `if-then-else` correspond formally to constraints rather than procedural elements. A mapping of OWL-S to PSL is provided in [24].

We now get back to our stock purchase example, and provide a database-oriented cast on PSL and OWL-S. The PSL- and domain-specific predicates associated with our example can be viewed essentially as relations in some (potentially vast) relational database. Some of the domain-specific relations might be "accessible" by just one service, others accessible by several services, and still others might be accessible by all services. In the example, we might have the following domain-specific relations

```
available_stock(stock_name, qty, price)
reserved_stock(stock_name, qty, price, res_id)
purchase_reservations(res_id, start_time)
billable_accounts(acct_id, current_balance)
```

Using the above relations it is now relatively straightforward to specify the behaviors of the atomic processes (which can be viewed as atomic activities in the parlance of PSL). For example, if `select_stock` is called with inputs s, q, then the pre-condition can be specified as a test against the current contents of relation `available_stock`, to see if there is a tuple (s, q', p) with $q' \geq q$. The effect (which in this case will always occur if the pre-condition holds) might be to replace that tuple with $(s, q' - q, p)$, create a "new" reservation_id r, and insert (s, q, p, r, t) into `reserved_stock`, where t is a

time-stamp. All of these operations should be considered as an atomic database transaction. (A refinement would be to enable the case where the shares of stock fulfilling a single request would have multiple prices.)

Now, there is one step remaining before we have the full picture on how the behavior of the OWL-S atomic processes can be formally specified in PSL. Two general approaches have arisen in the context of the situation calculi (see [39] for a detailed discussion). Under one approach, relations such as `available_stock` are transformed by adding a new field, which holds a time value. Under a second approach, taken by PSL, this relation would be transformed into a function, say, `f_available_stock` that takes three arguments. Two PSL predicates, `holds` and `prior` are used to associate (intuitively) truth values to terms created using this function. For example, if the predicate `holds(f_available_stock(s,q,p), occ)` is true in a model, this corresponds to the intuition that the term `holds(f_available_stock(s, q, p)` is true "about the world" immediately after the occurrence `occ` was executed, or in terms of the relational perspective, that tuple (`s`, `q`, `p`) was in the relation `available_stock` at that time. The predicate `prior` works analogously, referring to the time immediately prior to the start of an occurrence.

OWL-S provides a variety of features beyond its rich notion of IOPEs. We mention one here, which is a family of constructs for building composite processes from atomic ones. These constructs are inspired by GOLOG and are reminiscent of flowchart-style constructs, but in OWL-S they are viewed as declarative constraints in the specification of a web service, rather than procedural imperatives.

We now briefly describe two of the key results concerning automatic composition for OWL-S and OWL-S-like services. The first is developed in [38], where salient aspects of the OWL-S model are mapped first into a Situation Calculus, and from there into the Petri net domain. In this composition result, it is assumed that all relevant aspects of the application domain are represented using a finite number of propositional fluents, and the family of permitted input and output argument values is finite. (In a relational database setting, this can be achieved by restricting attention to a finite domain.) The goal of the (single-use) composition is then specified in terms of an overall effect to be achieved (on all of the relevant propositional fluents) starting from some initial state (which in essence includes relevant input argument values). Another element of the result is the translation of OWL-S atomic processes into Petri net fragments; these are used to assemble a Petri net that corresponds to all possible sequencing of the processes. (For simplicity, it is assumed that no atomic service is invoked twice in these sequences.) The overall complexity of determining the existence of a composition is EXPSPACE. Of course, various heuristics can be applied to make this tractable. Also, the technique can be generalized to support composition of composite processes, as well as atomic ones.

Another approach to composition, also developed in the context of OWL-S, is developed in [35]. This work proposes a two-tier framework, based on *generic programs* and *customization via constraints*. Beginning with a family of atomic OWL-S services, a generic program can be specified using the GOLOG language. Note that this generic program may not be completely specified. As a second phase, additional constraints can be written to capture customizations needed by a given client. (It is typical to express these using

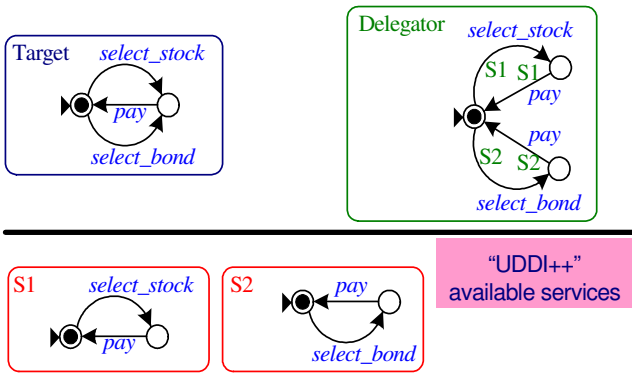

Figure 4: Illustration of composition with Roman model

Horn clauses). If the overall family of constraints is satisfiable, then there is a composition, which can be equated with a branch of the execution tree of the generic program that satisfies certain properties. Reference [35] describes how a ConGolog interpreter can be used to find such branches.

3.2 Roman Model perspective: Actions with Automata-based Process Model

We now consider a family of results involving automated composition of multi-step services. This work was launched by the seminal work [7, 5], which is of interest for at least two reasons: (a) the paper develops a novel, general framework for studying composition of human-machine style web services, and (b) the theoretical techniques developed in the paper can be generalized to a much broader context.

The paper starts with a very abstract model of web services, based on an abstract notion of *activities*. Basically, there is a (finite) alphabet of activity names, but no internal structure is modeled (no input, output, or interaction with the world). To specify the internal process flow of a web service, [7, 5] starts with transition systems. In the most general case, these are potentially infinite trees, where each branch corresponds to a permitted sequencing of executions of the activities. For the theoretical results they restrict attention to finitely specifiable transition systems; in particular on systems that can be specified as deterministic finite-state automata, where the activities act as the input alphabet (i.e., where the edges are labeled by activities). It is convenient to refer to this as the "Roman" model.

References [7, 5] focus on human-machine style web services. As a simple illustration of how services work, consider the service labeled `Target` in Figure 4. When this service is being used by a client (could be human or automated) it performs the following sequence of steps:

(1) Give the client a set S of activities to choose from (possibly including the special "activity" `terminate`).

(2) Wait for the client to choose exactly one element of S

(3) Execute the chosen activity and return to step (1), or terminate if that was chosen.

In each iteration, the set S is chosen according to the current state of the service – in a state p the set S includes each activity that labels an out-going edge from p, and includes "terminate" if p is a final state.

To illustrate, when the `Target` is launched it gives the client a choice of {`select_stock`, `select_bond`, `terminate`}.

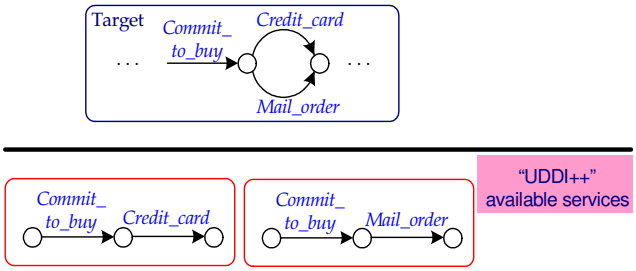

Figure 5: A composition problem with a 1 look-ahead solution

Supposing the client chooses `select_stock`, in the next iteration the service gives the set {`pay`}. Execution continues until the client chooses `terminate`.

We can now describe the basic composition problem studied with the Roman model. Consider now the two services `S1` and `S2` shown in Figure 4. Suppose that these are the only pre-existing services available. The question is whether it is possible to use some or all of those to "build" a service that acts the way `Target` does. One way to accomplish this is with the "service" shown as `Delegator` in Figure 4. The operation of this is similar to the operation of the other Roman services, except that with each transition of `Delegator` it is also assumed that one or more of the pre-existing services performs a transition. In particular, the pre-existing service must perform the same activity as the `Delegator`. Intuitively, one can think of `Delegator` as not executing the services, but rather, as delegating the execution to the pre-existing services. Furthermore, in a valid execution of `Delegator`, each of the pre-existing services must perform a valid (and terminating) execution.

The central result of [7] is that the existence of a delegator can be determined in EXPTIME, and that there is a constructive approach for building such delegators. Importantly, the form or skeleton of the delegator may be different than the form or skeleton of the target service (e.g., in the example the delegator has more states than the target service).

This result can be proved by using a transformation of the problem into Propositional Dynamic Logic (PDL), a well-known logic for reasoning about programs [31]. As detailed in [5], there is a natural, polynomial-size translation of a Roman composition problem into deterministic PDL. The result then follows because satisfiability for deterministic PDL is EXPTIME, and a constructive technique is available. Further, [6] describes how a description logic reasoner has been adapted to perform this test and construction.

We briefly mention two extensions of the results in [7]. Reference [10] extends the basic model by considering (pre-existing) services that can themselves delegate to other services. In the model there, if an edge in a service $S1$ has label $^{>>}a$ this indicates that $S1$ can perform activity a, and if an edge in $S2$ has label $a^{>>}$ this indicates that $S2$ is able to delegate activity a to some other service. The delegation feature adds a level of non-determinism, but checking the existence of, and constructing, a (generalized) delegator is still possible in EXPTIME.

Consider now the composition problem illustrated in Figure 5. There is no delegator that can simulate the target service in this case. However, as discussed in [22], there is a "1 look-ahead" delegator, that is, a delegator which can make the correct delegations, as long as the delegator is given information not only about the immediate choice of the client, but also the choice that the client will make in the next move. Reference [22] shows that there is a non-collapsing hierarchy of k look-ahead delegation problems, and there are examples requiring "infinite" or arbitrarily long look-ahead. Also, the problem of checking existence of a k look-ahead delegator can be transformed into a problem of checking existence of a conventional (0 look-ahead) delegator, and has EXPTIME complexity.

3.3 Conversations and Mealy services

The OWL-S and Roman models focus on *what* a service or composition does, either in terms of the input/output of services (activities) and their impact on the world, or the sequencing of the activities in a service. At the operational level, neither addresses the issue of *how* the component services in composition should interact with each other. The notion of "conversations" was naturally formulated in addressing this need [27, 28, 49]. Preliminary theoretical work on conversations was reported in [8, 18]. In this subsection, we highlight key technical notions and results. Much of this work assumes a peer-to-peer framework.

To begin with, we assume the existence of an infinite set of *service names*. These will act essentially as place-holders for *service spec(ification)s*, which can be viewed as completely or partially specified implementations for the service names. We also assume an infinite set of *message classes*, where each class mc has a service name as *source* and a service name as *target*. A *composition schema* is a a pair (P, M) where P is a finite set of service names and M is a finite set of message classes. whose sources and targets are in P. Figure 6 shows a composition schema for a "stock analysis service" (SAS) [20]. The SAS composition schema uses three service names: *Investor*, *Stock Broker*, and *Research Department*, and uses the 9 message classes indicated in the figure. This composition schema models abstractly a domain in which the *Investor* service can request reports on different stocks from the *Research Department*. The requests are controlled by the *Stock Broker* service, which handles issues such as permissions and billing.

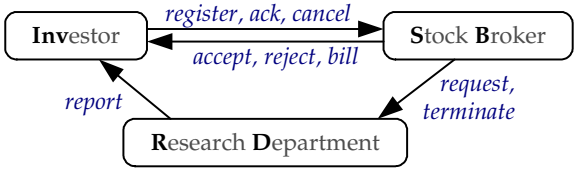

Figure 6: A Composition Schema

A *composite service* for composition schema (P, M) is an association of actual web services to each service name in P, where the actual web services are capable of sending/receiving messages coming from the relevant message classes in M. Given a composite service, a basic approach for the underlying operational model is that each service has a FIFO queue for all incoming messages (possibly from different services). This corresponds abstractly to the message oriented model in SOA [25, 1]. Queuing of messages provides a model asynchronous communication, which is realistic in the web setting.

Given a composite service S over composition schema (P, M), a *conversation* of S is the (global) sequence of messages sent during one successful execution of S. Several standards and research works are concerned with properties of

conversation languages. WS-Choreography [48], and references [27, 28, 48] focus on "global" constraints on conversations. In contrast, the standards WSCL [49] and WSDL2.0 [51] focus on "local" constraints on conversations, in the sense that they focus on constraints about message patterns of the component services of a composite service.

Reference [8] initiated the formal study of the relationship between such global and local constraints. A *conversation protocol* for a composition schema (P, M) is a finite state machine over the set of message classes in M. Figure 7 shows a conversation protocol for the SAS service. In this example, the *register* message from INV can include a list of stock symbols. Upon acceptance of a message of this type, each symbol will result in a separate *request* from the *Stock Broker* service to the *Research Department* service (corresponding to the *request-report-ack* loop in the protocol).

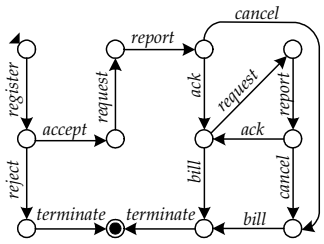

Figure 7: A Conversation Protocol for SAS

Three natural questions are (Q1) Given a conversation protocol over composition schema (P, M), can it be realized by some composite service S over (P, M)? (Q2) If so, what can be inferred about the individual services in S? (Q3) If the individual services in a composition have a message-passing behavior corresponding to a regular language, then is the resulting conversation also characterized by a regular language (i.e., by a conversation protocol)?

In [8] these questions are studied using the notion of *Mealy services*. These are service specs based on finite state automata whose transitions correspond to sending a message m (denoted as $!m$) or receiving a message m (denoted as $?m$). Fig. 8 shows a Mealy service for INV. A Mealy composite service is a composite service over Mealy services.

We note that a Mealy service can be viewed in three different ways: (a) as an implementation of a service name; (b) as a "behavioral signature" that abstractly describes some properties of a service behavior; and (c) as a constraint on the local messaging behavior that a service should satisfy.

With regards to question (Q3), in [8], it was shown that the conversation languages of some Mealy composite services are not regular (see Fig. 9), nor even context-free in the language hierarchy, but they are always context-sensitive. The key factor here is the use of *unbounded* queues. This is not surprising, since finite state automata with queues

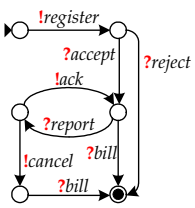

Figure 8: A Mealy Service for Investor

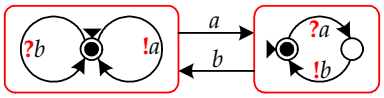

Figure 9: A Non-regular Conversation Language

can simulate Turing machines [13]. When all queues are bounded, the conversation languages are always regular.

Turning to question (Q1), a finer analysis reveals two important aspects of conversations languages: (a) What individual Mealy service sees vs. the (global) conversation, and (b) When two Mealy services are both ready to send, it isn't possible to force a particular ordering. These two factors are formulated into two operators in [8]:

- *Projecting* a conversation to the message alphabet of a single service name and *joining* message sequences of Mealy services into a conversation, and
- *Preponing* (i.e., swapping) messages from different senders (in a local perspective).

It turns out that Mealy conversation languages are closed under (1) projection-join and (2) preponing.

The closure results indicate that if a conversation protocol is not closed under either operator, then it cannot be realized by any composite service. Intuitively, this suggests that during design it may not be a good candidate as a global constraint on the composition during design. On the other hand, it is interesting to know if the projection-join and prepone closure of a regular language is *realizable* as the conversation language of some Mealy composite service. The answer turns out to be positive [8], and, relevant to question (Q2), a construction is provided.

It is not clear whether one can always determine if a regular language is closed under projection-join and preponing, though a sufficient condition was given in [18].

We conclude this discussion by describing two broad ways in which the above results can be applied in practical settings. One is to help in the process of practical compositions. Suppose that one seeks to create a (peer-to-peer) composition, based on a pre-determined composition schema, and satisfying a given conversation protocol. Suppose further that a family of component Mealy services is synthesized to satisfy this protocol. This family of Mealy services can be used to select a family of candidate (real-world) web services that satisfy the "local" constraints specified by the Mealy services. Then perform further analysis of the candidate services, on aspects that are not captured by the Mealy services alone.

The second practical application is to view a conversation language as a global constraint that should be satisfied by a family of web services, essentially in the spirit of the WS-Choreography standard. The results above can be used to determine whether this global constraint can be realized at all. It can also be used to infer local constraints, that specify properties of the behavioral signatures of each component service.

3.4 Selected Additional Developments

We close our sampling of composition models and results with a brief overview of two very recent efforts, which hold the promise of providing a unifying framework for this area.

The first, already touched upon in Subsection 3.1, is the work on FLOWS [9, 15], which provides a first-order logic

ontology for web services. We provide here a little more detail on FLOWS.

FLOWS includes objects of type *service*. These can be related to several types of object, including non-functional properties (e.g., name, author, URL), and also to its *activity*. This is a PSL activity, and may have specialized kinds of subactivities. This includes (*FLOWS*) *atomic processes*, which are based on OWL-S atomic processes with inputs, outputs, pre-conditions and conditional effects. Some of these atomic processes, as in OWL-S, are *domain-specific*, in that they interact with the "real world" as represented in the application domain, and others are *service-specific*, in that they focus on the standardized mechanics of services, e.g., message-passing and channel management (see below).

The flow of information *between services* can occur in essentially two ways: (a) via message passing and (b) via shared access to the same "real world" fluent (e.g., an inventory database, a reservations database). With regards to message passing, FLOWS models *messages* as (conceptual) objects that are created, read, and (possibly) destroyed by web services. A message life-span has a non-zero duration. Messages have types, which indicate the kind of information that they can transmit. FLOWS also includes a flexible *channel* construct.

To represent the acquisition and dissemination of information *inside a Web service*, FLOWS follows the spirit of the ontology for "knowledge-producing actions" developed in [45]; this also formed the basis for the situation calculus semantics of OWL-S inputs and effects [38].

FLOWS is very open-ended concerning the process or data flow between the atomic processes inside a services. This is intentional, as there are several models for the internal processing in the standards and literature (e.g., BPEL, OWL-S Process Model, Roman model, Mealy model) and many other alternatives besides (e.g., rooted in Petri nets, in process algebras, in workflow models, in telecommunications). The FLOWS work is still quite preliminary, but holds the promise of providing a unifying foundation for the study and comparison of these many variations on the notion of web service.

A second very recent work is reported in [4, 3]. That work develops Colombo, a formal model for web services that combines

(a) A world state, representing the "real world", viewed as a database instance over a relational database schema

(b) Atomic processes in the spirit of OWL-S,

(c) Message passing, including a simple notion of ports and links, as found in web services standards (e.g., WSDL, BPEL)

(d) An automata-based model of the internal behavior of web services, where the individual transitions correspond to atomic processes, message writes, and message reads.

The first three elements parallel in several aspects the core elements of FLOWS. Colombo also includes

(e) a "local store" for each web service, used manage the data read/written with messages and input/output by atomic processes; and

(f) a simple form of integrity constraints on the world state

Using the Colombo model, [4] develops a framework for posing composition problems, that closely parallels the way composition will have to be done using standards-based web services. Specifically, the basic composition problem studied is how to build a mediator that simulates the behavior of a target web service, where the mediator can only use message passing to get the pre-existing web services to perform atomic activities (which in turn impact the "real world"). Under certain restrictions, [4] demonstrates the decidability of the existence of a mediator and develops a method for constructing them. This result is based on (a) a technique for reducing potentially infinite domains of data values into a finite set of symbolic data values, and (b) in a generalization of [5], a mapping of the composition problem into PDL. The results reported in [4] rely on a number of restrictions; a broad open problem concerns how these restrictions can be relaxed while still retaining decidability.

4. ANALYSIS COMPOSITE SERVICES

The need for analysis is particularly acute for composite services, especially if they are to be created from pre-existing services using automatic algorithms. The ultimate goal is to ensure that the eventual execution of a composite service produces the desired behavior. Ideally, one would be able to statically verify properties (e.g., in temporal logic) for composite services. There have been various attempts at developing such static analysis methods for web services and workflow systems.

Based on workflows represented in concurrent transaction logic, [16] studied the problem of whether a workflow specification satisfies some given constraints similar to the Event Algebra of [44]. Algorithms were given for such analysis. An alternative approach is developed in [47], which maps conventional workflows to Petri nets, and then applies standard results to analyze properties such as termination and reachability for workflows. Similar results have been obtained for OWL-S compositions [38].

There are two recent projects that use model checking techniques to verify BPEL composite web services. In [21], mediated composite services specified in BPEL were verified against the design specified using Message Sequence Chart and Finite State Process notations, with a focus on the control flow logic. In [19], both conversation protocols and Mealy services were extended to *guarded automata* which incorporate (i) XML messages and (ii) XPath expressions as the basis for verifying temporal properties of the conversations of composite web services. The following shows an example of a transition in a protocol for the SAS service:

```
t14 {
  s8 -> s12 : bill,
  Guard {
    $request//stockID =
    $reginfo//stockID [position() = last()]
    =>
    $bill[//orderID := $reginfo//orderID]
  }
},
```

Roughly, the transition is defined from state "`s8`" to "`s12`" on message "`bill`". The part of guard prior to "`=>`" defines an additional equality condition with both side XPath expressions. The part after "`=>`" specified an assignment.

The extended model makes it possible to verify designs at a more detailed level and to check properties about message content. A framework is presented where BPEL specifications of web services are translated to an intermediate representation, followed by the translation of the intermediate representation to the verification language Promela, input language of the model checker SPIN [30]. Since the SPIN model checker is a finite-state verification tool, the tool can only achieve partial verification by fixing the sizes of the input queues in the translation. Sufficient conditions for complete verification are also obtained.

An important step in statically analyzing Web services defined in BPEL (or other languages) is to translate them into formalisms that are suitable for analysis. Such formalisms include finite state machines [37, 21], extended Mealy machines [19], process algebra [32]. Effectively, these translations provide formal semantics to BPEL. Most of the translations mentioned above focus only on control flow structures, with an exception of [19] (where XML message types, local data, and XPath expressions are also taken into consideration. More recently, an extensive translation of BPEL into a version of Petri nets was developed [33], which deals with scoping rules, variable assignments, correlation sets, among other things.

The presence of databases makes the static analysis extremely hard but not entirely impossible. In [17], a rule based language was developed for specifying interactive Web services. The rules may access associated (relational) databases. Web service properties are defined using a linear temporal first-order logic. Based on the Abstract State Machine techniques [46], it was shown that some properties are verifiable for a subclass of Web services.

Service oriented architecture may "fundamentally change the way software is made and used" [11]. The current development platforms may no longer be suitable. While this is a rare chance to revisit important software development environment, there is no doubt that analysis tools (static or dynamic), testing and debugging tools, monitoring tools are becoming critically importance.

5. CONCLUSIONS

The web services paradigm raises a vast array of questions, including many that will be of interest to, and can benefit from, the database research community. First is the question of finding appropriate models and abstractions for representing Web services and their "behaviors", which are suitable to the web services paradigm, and can support efficient querying and manipulation as needed by web service composition and analysis algorithms. More broadly, to what extent can the problem of automated composition be re-cast to be a problem in writing and answering one or several queries against behavioral descriptions of services? Another aspect concerns the application of XML constraint-checking techniques to perform compile-time or run-time checking of Web service specifications (e.g., in WSDL and BPEL, or in emerging behavioral specification languages).

A second category of questions is how to bring data manipulation more clearly into the web services paradigm and their associated standards. The standards and most research at present are focused primarily on process model and I/O signatures, but not on the data flow and the manipulation of the data as it passes through this flow. Is there value in associating integrity constraints with web service I/O signatures? What is an appropriate way to model the data transformations occurring in a web service, which will enable reasoning about the behavior of data being passed or written to databases by a composite web service? Are there specialized models of Web service composition that will be more suitable for applications that are targeted primarily at data processing? A starting point here might be [2, 17].

Acknowledgements: The authors wish to thank their collaborators for many fruitful discussions and inspirations, including Michael Benedikt, Daniela Berardi, Tevfik Bultan, Diego Calvanese, Vassilis Christophides, Guiseppe De Giacomo, Xiang Fu, Cagdas Gerede, Michael Gruninger, Oscar Ibarra, Grigorios Karvounarakis, Maurizio Lenzerini, Massimo Marcella, and Sheila McIlraith.

6. REFERENCES

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services. Concepts, Architectures and Applications.* Springer-Verlag, 2004.

[2] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *Proc. ACM Symp. on Principles of Database Systems*, 1998.

[3] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella. Automatic composition of web services in Colombo. In *Proc. of 13th Itallian Symp. on Advanced Database Systems*, June 2005.

[4] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella. Automatic Composition of Transition-based Semantic Web Services with Messaging. Technical report, University of Rome, "La Sapienza", Roma, Italy, April, 2005.

[5] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic Services Composition based on Behavioral Descriptions. *International Journal of Cooperative Information Systems (IJCIS)*, 2004. To appear.

[6] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. $\mathcal{ESC}$: A Tool for Automatic Composition of e-Services based on Logics of Programs. In *Proc. Workshop on Technologies for E-Services*, 2004.

[7] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *Proc. 1st Int. Conf. on Service Oriented Computing (ICSOC)*, volume 2910 of *LNCS*, pages 43–58, 2003.

[8] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: A new approach to design and analysis of e-service composition. In *Proc. Int. World Wide Web Conf. (WWW)*, May 2003.

[9] D. Berardi, M. Gruninger, R. Hull, and S. McIlraith. Towards a first-order ontology for web services. In *W3C Workshop on Constraints and Capabilities for Web Services*, October 2004.

[10] D. Berardi, G. De Giacomo, M. Lenzerini, M. Mecella, and D. Calvanese. Synthesis of underspecified composite *e*-services based on automated reasoning. In *Proc. Second International Conference on Service-Oriented Computing*, pages 105–114, 2004.

[11] J. Bloomberg. The seven principles of service-oriented development. *XML & Web Services*, 3(5):32–33, 2002.

[12] Business Process Execution Language for Web Services (BPEL), Version 1.1. `http://www.ibm.com/developerworks/library/ws-bpel`, May 2003.

[13] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.

[14] OWL Services Coalition. OWL-S: Semantic markup for web

services, November 2003.

[15] SWSL Committee. Semantic Web Service Ontology (SWSO), 2005. Available in `http://www.daml.org/services/swsl/report/`.

[16] H. Davulcu, M. Kifer, C. R. Ramakrishnan, and I. V. Ramakrishnan. Logic based modeling and analysis of workflows. In *Proc. ACM Symp. on Principles of Database Systems*, pages 25–33, 1998.

[17] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web services. In *Proc. ACM Symp. on Principles of Database Systems*, 2004.

[18] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and verification of reactive electronic services. In *Proc. Int. Conf. on Implementation and Application of Automata (CIAA)*, 2003.

[19] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proc. Int. World Wide Web Conf. (WWW)*, May 2004.

[20] X. Fu, T. Bultan, and J. Su. Model checking XML manipulating software. In *Proc. Int. Symposium on Software Testing and Analysis (ISSTA)*, July 2004.

[21] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proc. the 18th IEEE Int. Conf. on Automated Software Engineering Conference (ASE 2003)*, 2003.

[22] C. E. Gerede, R. Hull, Oscar H. Ibarra, and J. Su. Automated composition of e-services: lookahead. In *Proc. Intl. Conf. on Services-Oriented Computing*, 2004.

[23] M. Grüninger and C. Menzel. Process specification language: Principles and applications. *AI Magazine*, 24:63–74, 2003.

[24] M. Grüninger. Applications of PSL to semantic web services. In *Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases*, 2003.

[25] H. Haas. Architecture and future of web services: from SOAP to semantic web services. `http://www.w3.org/2004/Talks/0520-hh-ws/`, May 2004.

[26] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-services: A look behind the curtain. In *Proc. ACM Symp. on Principles of Database Systems*, 2003.

[27] J. E. Hanson, P. Nandi, and S. Kumaran. Conversation support for business process integration. In *Proc. 6th Int. Enterprise Distributed Object Computing (EDOC)*, Ecole Polytechnic, Switzerland, 2002.

[28] J. E. Hanson, P. Nandi, and D. W. Levine. Conversation-enabled web services for agents and e-business. In *Proc. International Conference on Internet Computing (IC)*, pages 791–796. CSREA Press, 2002.

[29] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[30] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, 2003.

[31] D. Kozen and J. Tiuryn. Logics of programs. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science — Formal Models and Semantics*, pages 789–840. Elsiever, 1990.

[32] M. Koshkina and F. van Breugel. Verication of business processes for web services. Technical Report CS-2003-11, Department of Computer Science, York University, 2003.

[33] A. Martens. Analysis and re-engineering of web services. In *Proc. 6th Int. Conf. on Enterprise Information Systems*, 2004.

[34] R. Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[35] S. McIlraith and T. Son. Adapting Golog for composition of semantic web services. In *Proc. of the Eighth International Conference on Knowledge Representation and Reasoning (KR2002)*, pages 482–493, April 2002.

[36] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. In *IEEE Intelligent Systems*, March/April 2001.

[37] S. Nakajiama. Verification of web services with model checking techniques. In *Proc. First Int. Symposium on Cyber Worlds*. IEEE Computer Society Press, 2002.

[38] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Proc. Int. World Wide Web Conf. (WWW)*, 2002.

[39] J. A. Pinto. *Temporal Reasoning in the Situation Calculus*. PhD thesis, University of Toronto, 1994.

[40] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Importing the semantic web in UDDI. In *Proc. Workshop on Web Services, E-business and Semantic Web (at CAISE conference)*, 2002.

[41] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, Cambridge, MA, 2001.

[42] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[43] PSL standards group. PSL home page. `http://ats.nist.gov/psl/`.

[44] M. Singh. Semantical considerations on workflows: An algebra for intertask dependencies. In *Proc. Workshop on Database Programming Languages (DBPL)*, 1995.

[45] R. B. Scherl and H. J. Levesque. Knowledge, action, and the frame problem. *Artificial Intelligence*, 144:1–39, 2003.

[46] M. Spielmann. Verification of relational transducers for electronic commerce. *Journal of Computer and System Sciences*, 66(1):40–65, 2003.

[47] W. M. P. van der Aalst. On the automatic generation of workflow processes based on product structures. *Computer in Industry*, 39(2):97–111, 1999.

[48] Web Services Choreography Description Language Version 1.0 (W3C Working Draft). `http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/`, December 2004.

[49] Web Services Conversation Language (WSCL) 1.0. `http://www.w3.org/TR/2002/NOTE-wscl10-20020314/`, March 2002.

[50] Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, March 2001.

[51] Web Services Description Language (WSDL) Version 2.0 Part 2: Predefined Extensions (W3C Working Draft). `http://www.w3.org/TR/2004/WD-wsdl20-extensions-20040803/`, August 2004.