Towards a Formal Verification of OWL-S Process Models [*]

Anupriya Ankolekar, Massimo Paolucci, and Katia Sycara

Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
{anupriya,paolucci,katia}@cs.cmu.edu

Abstract. In this paper, we apply automatic tools to the verification of interaction protocols of Web services described in OWL-S. Specifically, we propose a modeling procedure that preserves the control flow and the data flow of OWL-S Process Models. The result of our work provides complete modeling and verification of OWL-S Process Models.

1 Introduction

Verification of the interaction protocol of Web services is crucial to both the implementation of Web services and to their use and composition. The verification process can prove important and desirable properties of the control flow of a Web service. At implementation time, a Web service provider will want to verify that the protocol to be advertised is indeed correct, e.g. does not contain deadlocks. A Web service provider may also want to guarantee additional properties, e.g. purchased goods are not delivered if a payment is not received.

Even if the Web service provider verifies the correctness of the programming logic behind its Web services, it will still need to verify the advertised interaction protocol. The mapping from the programming logic of the Web service to the interaction protocol of the Web service is typically lossy. Thus, the Web service provider will need to verify that claims that were true of the Web service program also hold true of the interaction protocol. Furthermore, the interaction protocol may make use of several Web services provided by the same Web service provider or possibly by other third-party providers. In either case, verifying the programming logic of multiple Web services is impracticable. In these cases, verifying the interaction protocol itself is both possible and useful.

During composition and use of Web services, a Web service client may want to verify the Web service provider's interaction protocol to obtain a guarantee that the protocol is correct, e.g. it does not contain an infinite loop, and that it conforms to the client's requirements. For example, the client may want to ensure that whenever a payment is received by the service provider, the goods

[*] This research was funded by the Defense Advanced Research Projects Agency as part of the DARPA Agent Markup Language (DAML) program under Air Force Research Laboratory contract F30601-00-2-0592 to Carnegie Mellon University.

are delivered to the client, or that there is the possibility of reimbursement, if the goods are returned.

In this paper, we explore the verification of OWL-S[1] interaction protocols using automatic verification tools, such as the Spin model-checker [8]. OWL-S is one of the leading standards for the description of Web services on the Semantic Web. The OWL-S Process Model describes the interaction protocol between a Web service and its clients. Such protocols are inherently non-deterministic and can be arbitrarily complex, containing multiple concurrent threads that may interact in unexpected ways. By performing an efficient exploration of the complete set of states that can be generated during an interaction between a Web service and its clients, Spin is able to verify numerous properties of the OWL-S Process Model.

The work presented in this paper builds on work presented in [2][2]. In particular, we relaxed many of the abstractions in the previous version, added the modeling of loops and enriched the literature review. The rest of this paper is organized as follows. After reviewing related work in section 2, we provide a quick overview of OWL-S 1.1 in section 3, using a running example based on the Amazon Web service. In section 4, we provide an introduction to verification with Spin. In section 5, we then define a mapping of the Amazon example from OWL-S 1.1 to Spin's Promela language, which is used to construct models that the Spin system can analyze. We then describe the verification of claims on the Amazon Process Model using Spin in section 6. Finally, in section 7 we will discuss our results and future work.

2 Related Work

Previous work on OWL-S verification is scant. Narayanan et al. [9] proposed a Petri Net-based operational semantics, which models the control flow of a Process Model exclusively[3]. On the basis of this mapping of OWL-S Process Models to Petri Nets, a number of theorems are proven on the computational complexity of typical verification problems, such as reachability of states and discovery of deadlocks. The results show that the complexity of the reachability problem for OWL-S Process Models is PSPACE-complete. This result is not surprising given the complexity of the OWL-S Process Modeling language.

Our approach improves on Narayanan's seminal work in three directions. First, we provide a model of Web service data flow in addition to control flow. As a result, the verification procedure can detect harmful interactions (see section 5.1) between data and control flow that would be undetected otherwise. Second, as part of our modeling methodology, we translate an OWL-S Process Model

[1] Our work is based on the OWL-S 1.1 release available at http://www.daml.org/services/owl-s/1.1/.

[2] The authors are in debt to the participants of the workshop on "Semantic Web Services" at ISWC2004 for their useful comments.

[3] Narayanan's semantics was defined for an earlier version of OWL-S (namely DAML-S 0.5), which did not model data-flow.

into a simpler model that nevertheless preserves all the essential behavior to be verified. Third, we provide initial results on the actual verification of OWL-S Process Models using existing verification tools such as Spin. The result of our work is a complete procedure for the modeling and verification of OWL-S Process Models.

While we are aware of only one other work on the verification of OWL-S Process Models, there has been a considerable amount of work on the verification of BPEL [1] Models. For example, WSAT (Web Service Analysis Tool) [13, 6] provides a formal verification of composite Web services expressed in BPEL and WSDL using guarded automata (GA) to construct the model, and then mapping the GA into Promela using Spinas verifier. A different approach to the verification of BPEL is followed by [7], which is based on message sequence charts, while [11] provide a Petri Net semantics and verification model.

Unfortunately, there is no clear mapping between OWL-S and BPEL. BPEL aims to represent the composition of Web services, showing how different services can interact to solve a problem. Consequently, any verification of BPEL compositions aims to check that the different composed services can indeed work together. OWL-S, on the other hand, provides a representation of the Process Model of one single Web service, leaving the composition problem to some other entity, typically a synthetic planner. The focus here is on verifying whether the particular Web service has the properties that the client expects in order to make use of it. It is therefore quite difficult to export the results of work on verifying BPEL to the verification of OWL-S Web services.

3 OWL-S Process Model

The OWL-S Process Model is organized as a workflow of processes. Each process is described by three components: inputs, preconditions and results. Results specify what outputs and effects are produced by the process under a given condition. For example, a process may have different results depending on whether the client is a premium user, or an ordinary user. OWL-S processes describe the information transformation produced by the Web service; while preconditions and effects describe the knowledge state transition produced by the execution of a Web service.

Processes in the workflow are related to each other by data flow and control flow. Control flow allows the specification of the temporal relation between processes. OWL-S supports a wide range of control flow mechanisms including sequentially executed processes, spawning of concurrent processes, synchronization points between concurrent processes, conditional statements and non-deterministic selections of processes. OWL-S distinguishes between atomic and composite processes. Atomic processes are indivisible processes that result in a message exchange between the client and the server. Composite processes are used to describe the control flow relation between processes. Fig. 1 shows a simple fragment of the Process Model adopted by Amazon.com's Web service. The nodes of the tree correspond to composite processes that represent different

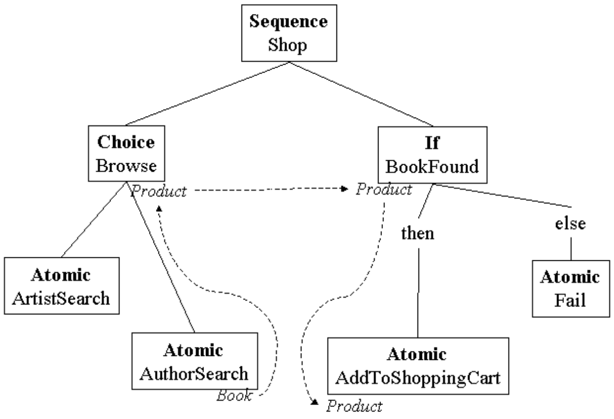

Fig. 1. The Process Model of Amazon.com's Web service

control constructs such as **Choice** for non-deterministic choices, **Sequence** for deterministic sequences of processes, and **If** conditionals. Atomic processes are represented as the leaves of the tree. For example, *Author Search* requires the client to provide information such as the name of an author. It then reports books found written by that author.

Data flow allows the specification of the relation between inputs and outputs of processes. An example of data flow is shown using dashed lines in Fig. 1. An output of the process *AuthorSearch* is a book which is then passed to the parent process, *Browse* and further up until it reaches the input of the process *AddToShoppingCart*. The scope of the data flow is limited to within a composite process. Therefore processes in a composite process can exchange data among themselves or with the parent process, but with no other processes. As the figure shows, data exchanges between two arbitrary processes, as for example *AuthorSearch* and *AddToShoppingCart* result from the composition of data flow links in the whole Process Model.

4 Model Checking with Spin

OWL-S Process Models are typically verified using human inspection, simulation and testing. However, due to their complex and concurrent nature, OWL-S Process Models are not very amenable to such verification techniques. Instead, we use model checking [5], a method that has been successful in the verification of distributed systems, such as Web services [6, 12]. Model checking exhaustively checks all possible executions of a system to verify that certain properties hold. It can thus formally *prove* the correctness of a system.

To construct such proofs, model checking requires two decisions to be made [8]. The first decision is about *what claims to prove*: a claim states invariant properties of the code, e.g. that a variable will always be instantiated or that it will always reach a given value. Typically, two kinds of properties are proven about a given protocol: safety properties, which guarantee that specified undesired states, such as deadlocking states, are never reached; and liveness properties, which specify that desired states are eventually reached.

The second decision relates to *what and how to model*, in other words which aspects of the protocol are relevant to the claims to be verified, and how to ensure that the model of the protocol preserves the behaviors to be checked. Certain aspects of the protocol may be verified better in other ways, for instance type safety can be ensured using a type checker. Moreover, a simplified model of the implementation, one that captures the essentials of the design, but avoids the full complexity of the implementation, can often be verified easily, even when the full implementation cannot. Thus, generating a *verification model* for an interaction protocol entails the translation of the protocol into a formal specification, which encapsulates the modeling decisions and specifies the claims to be verified.

This specification is input to a model checking tool, such as Spin, to automatically verify that the protocol satisfies the claims. If Spin verifies that a claim is true in the Promela specification, given that the specification captures the relevant behavior of the OWL-S Process Model, we know that the claim is also true in the corresponding Process Model. On the other hand, if the protocol contains an error, a model checker can provide a counter-example, identifying the conditions under which the error occurs. The claim may still hold in the OWL-S Process Model, because the Promela specification does not capture the full behavior of the Process Model. In this case, the counterexample that `Spin` provides can be analyzed and simulated in the actual Process Model. If this does produce faulty behavior, then a bona fide bug has been discovered, else a spurious bug [5] has been identified.

In this work, we use the Spin model checker, a generic verification system that supports the design and verification of a system of asynchronous processes. Spin accepts design specifications in Promela (a *Pro*cess *Me*ta *La*nguage) and correctness claims in LTL (Linear Temporal Logic). Promela is akin to a highly concurrent programming language, while LTL enables the representation of formulae about possible execution paths of a process. Due to space limitations, the readers are referred to Chapters 3 and 6 of [8] for a comprehensive discussion of Promela and LTL. In the rest of the paper, we will describe aspects of Promela and LTL only as relevant to our work.

5 Modeling OWL-S Process Models in Promela

The mapping of OWL-S to Promela hinges on the decision of which aspects of the OWL-S Process Model are to (and can) be expressed in Promela, and on how to perform such a mapping. In the rest of this section, we describe the mapping of OWL-S Process Models to Promela models. Throughout this section, the Process Model of the Amazon Web service (Fig. 1) [10] will be used as a running example to illustrate the mapping rules.

```
(1)  proctype Shop () {
(2)   chan syncChan = [1] of { int,mtype };
(3)   chan dataChan = [1] of { int };
(4)   pid x1, x2;
(5)   x1 = run Browse(syncChan, dataChan);
(6)   if
(7)   :: syncChan??eval(x1),done ->
(8)        x2 = run ProductFound(syncChan, dataChan);
(9)        if :: syncChan??eval(x2),done -> skip; fi
(10)  fi;
(11) }
```

Fig. 2. The `Shop` Process

Modeling Composite Processes OWL-S Processes map naturally onto processes in Promela. Processes in Promela are introduced by `proctype` and are instantiated with the `run` operator. For example, Fig. 2 shows the result of the translation of the top-level `Shop` process to Promela. `Shop`, being a top-level process, does not take any arguments. Instantiated processes are inherently concurrent. Thus, `Browse` and `ProductFound` run concurrently with `Shop`.

In Promela, if processes are to be executed in a particular order, e.g. in a sequence, they must be explicitly synchronized. Each parent composite process, therefore, creates a `syncChan`, a typed channel for control flow[4], and optionally an additional typed channel for data flow, `dataChan`, to be used by its child processes. Channels are used to model data flow between processes and can be either globally scoped or locally scoped within a single process. Channels can have a predetermined storage capacity. When the channel capacity has been reached, additional messages sent to the channel will be dropped. Receive statements that retrieve messages from channels block until a message is present in the channel. Fig. 2 shows the definition of these channels, within the `Shop` process, in lines 2-3. The channels have a storage capacity of at most one message. They are passed to the processes `Browse` and `ProductFound` (lines 5 and 8 resp.).

The `syncChan` channel holds tuples consisting of an integer, corresponding to the process id of the sending process, and `done`. Messages sent to `dataChan` are integers, representing the data values sent via data flow links (see below). Promela supports all the traditional programming language types such as `int`, `char`, `boolean`, arrays and records. In addition, Promela supports a form of

[4] An alternative to channels is the use of variables, as follows: a process would set a particular synchronization variable just before it terminates and other processes would wait for the variable to become true before executing. Although this mechanism is attractively simple, it fails when multiple concurrent processes are used. Since Promela admits only two kinds of scope, global or local to a single process, any synchronization variable must necessarily be globally defined. However, global variables may be overwritten when multiple instances of processes can be spawned dynamically.

```
(1)  proctype SplitJoin(chan syncChan, dataChan) {
(2)   chan childSync = [2] of { int,mtype };
(3)   pid childA = run A(childSync);
(4)   pid childB = run B(childSync);
(5)   if
(6)   :: childSync??eval(childA),done ->
(7)       if
(8)       :: childSync??eval(childB),done;
(9)       fi
(10)  fi
(11)  syncChan!_pid,done;
(12)}
```

Fig. 3. Implementation of a prototypical `SplitJoin` statement

enumerated type called `mtype`, which is typically used to describe message types. However, Promela does not support an OOP-style (Object-Oriented Programming type hierarchy.

Modeling `Split` and `SplitJoin` Since processes in Promela are intrinsically concurrent, `Split` and `SplitAndJoin` can be naturally implemented as follows: the counterpart of each construct is a process in Promela, which simply spawns all its child processes. At this point, a `Split` process would immediately terminate, whereas a `SplitAndJoin` process would wait for the termination of the processes it spawned.

Since there are no `Split` and `SplitAndJoin` statements in the Amazon example, Fig. 3 shows a prototypical implementation of a `SplitAndJoin` in lines 3-4. The process spawns off two processes `A()` and `B()` with no data flow link in between. The guards in lines 6 and 8 check whether `childSync` contains a `done` message sent by `childA` or `childB`, respectively. The entire `SplitAndJoin` process blocks until the guard becomes true, thus synchronizing the process with the termination of its child processes. Finally, in line 11, the process signals its own termination. The implementation of a `Split` statement would be identical, but skip lines 5-10, which implement the `Join` synchronization.

Modeling Sequences While concurrent processes can be implemented in a relatively straightforward way, the modeling of OWL-S sequences requires explicit synchronization, which is similar to the synchronization proposed for `SplitJoin`. We implement sequences by first spawning off the first process in the list, blocking until the process terminates, then spawning off the second process. The implementation of the `Shop` process, a `sequence` of `Browse` and `ProductFound` processes is shown in Fig. 2. The Promela specification of `Shop` first spawns the `Browse` process in line 5. In the `if` statement, the execution of `Shop` is blocked (line 7) until it receives a `done` message from `Browse`, signaling that the `Browse` process is complete. `Shop` then spawns `ProductFound` (line 8) and waits for it to complete before terminating itself.

```
(1)  proctype Browse (chan syncChan, dataChan) {
(2)   chan childSync = [1] of { int,mtype };
(3)   chan childData = [1] of { int };
(4)   pid child; int product;
(5)   if
(6)   :: true -> child =
          run AuthorSearch(childSync, childData);
(7)      if
(8)       :: childData?product -> dataChan!product;
(9)       :: childSync??eval(child),done;
(10)     fi
(11)  :: true -> child =
          run ArtistSearch(childSync, childData);
(12)     if
(13)      :: childData?product -> dataChan!product;
(14)      :: childSync??eval(child),done;
(15)     fi
(16)  fi;
(17)  syncChan!_pid,done;
(18)}
```

Fig. 4. Choice and Conditionals: the Browse Process

Modeling Choices and Conditionals OWL-S Choices and Conditionals are both implemented using PROMELA's guarded non-deterministic choice statements `if :: fi`. A non-deterministic choice in PROMELA is defined by an `if` statement, where all guard conditions are true. The implementation of `Browse`, shown in Fig. 4, provides an example of a `choice` between two atomic processes, `AuthorSearch` and `ArtistSearch`. The conditions of the `if` statement at lines 6 and 11 are both true, so PROMELA non-deterministically chooses one of the branches for execution. After spawning the chosen process, the execution blocks, waiting for the process to complete, and then sets the output `product`.

In OWL-S conditions occur in `Result` statements and `if` statements. A `Result` condition specifies when a given output or effect is generated, an `if` is defined as part of the control construct. OWL-S `Result` conditions reflect the state of the server. For example, while interacting with a Web service like Amazon's, the client may discover that the book being sought is not available. Similarly, `if` conditions in OWL-S depend on the knowledge of the agent at execution time, in particular on the effects of previous steps and their interaction with the agent's knowledge. From the point of view of software verification, such a condition could be considered a random variable, whose value cannot be known at verification time and may equally be true or false. We therefore model `Results` and `if` statements as non-deterministic choice. This forces the verifier to evaluate the correctness of both branches of the Model. An OWL-S conditional is implemented in a similar way to OWL-S Choice, but with the `if` condition as a guard to the `then` statement and an `else` guard to the `else`

```
(1) proctype AuthorSearch (chan syncChan, dataChan) {
(2)   if /* implement conditional outputs */
(3)   :: true -> atomic {
(4)      int bookResult= 1;
(5)      dataChan!bookResult;}
(6)   :: true -> skip
(7)   fi;
(8)   syncChan!_pid,done;
(9) }
```

Fig. 5. Atomic process: the `AuthorSearch` Process

statement. According to PROMELA semantics, the `else` guard is only true, if all other guards are false.

Modeling Atomic Processes Finally, we present the mapping of an *atomic* process, which produces different results, to PROMELA. We model the selection of results with a non-deterministic choice. The implementation of the atomic process `AuthorSearch` is shown in figure 5. The conditional outputs are specified in lines 3 and 6 with a non-deterministic choice. If line 3 is selected, then the variable `bookResult` is assigned to 1 (line 4) and its value is sent out on the data channel (line 5). The other atomic processes, `ArtistSearch` and `AddToShoppingCart` can be specified analogously.

Modeling Data Flow For a given data flow link that maps outputs to inputs, one would ideally like a guarantee that the class of the input always subsumes the class of the output. Verifying this using SPIN would require the subsumption relations in the ontology of the client to be represented within the PROMELA model. In addition, SPIN would need to be able to compute a subsumption hierarchy of classes. Since this would immediately overwhelm the verifier, we abstract from the actual values of inputs and outputs. Instead, the types of inputs and outputs are modeled as integers and data flow links as channels. Inputs that are not bound by a data flow link are expected to be initialized with some suitable value, usually 0. The evaluation of type subsumption claims are deferred to a pre-processor, such as a type-checker or a reasoner, that can methodically verify the integrity of all data flow links.

The data flow is represented by a variable that represents the output and the `dataChan` channel that transfers data between processes. Different parts of the

```
proctype AddToShoppingCart (chan syncChan, dataChan) {
  int product; dataChan?product;
  assert(product);
  syncChan!_pid,done;
}
```

Fig. 6. Data flow: the `AddToShoppingCart` Process

```
(1)  proctype Repeat-While(chan syncChan, dataChan) {
(2)   int v_1 = v_1_init;
(3)   int v_2 = v_2_init;
(4)   do
(5)   :: c -> p
(6)   :: else -> break
(7)   od
(8)   syncChan!_pid,done;
(9)  }
```

Fig. 7. Implementation of a prototypical `Repeat-While` statement

data flow have been represented in the samples code shown above. For instance, lines 3 to 5 of Fig. 5 represent the output `bookResult` and the transmission of its value on the `dataChan` channel. Lines 8 and 13 of Fig. 4 show how channels are chained in composite processes, where the results of child processes are transmitted as the results of the parent process. This chaining implements the data flow chain, shown in Fig. 1. Finally, the data transmitted across all the links of the chain should reach the input of another atomic process and be consumed there. Line 3 of Fig. 6 shows the implementation of the input `product` and its instantiation with the value coming from `dataChan`. The line `assert(product)` (line 3) specifies a claim on the state reached, namely that the value of product should not be zero, i.e. the input is instantiated to some value.

Modeling Loops There are two kinds of loops in OWL-S: the `Repeat-While` process and the `Repeat-Until` process. OWL-S loops have a loop condition `c`, a process `p` that is executed during every iteration of the loop and a number of variables, `v_i` that are local to the loop. Some of these variables may be referenced in the loop condition `c`.

Promela supports loops through the guarded `do :: od` statements. As an example of the implementation of loops, the definition of the `Repeat-While` process using `do` is shown in Fig. 7. The process first declares and initialises the loop variables, `v_1` and `v_2`, in lines 2 and 3 respectively. Then, the process enters the `do` loop, checking during each iteration that the condition `c` is true (line 5). If

```
(1)  proctype Repeat-Until(chan syncChan, dataChan) {
(2)   int v_1 = v_1_init;
(3)   int v_2 = v_2_init;
(4)   p;
(5)   do
(6)   :: !c -> p
(7)   :: else -> break
(8)   od
```

Fig. 8. Implementation of a prototypical `Repeat-Until` statement

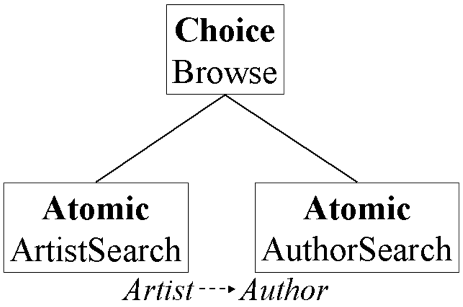

Fig. 9. An example of interaction between data and control flow in OWL-S

so, the process `p` is executed; otherwise, the loop is broken (line 6) and the process signals its termination (line 8). The `Repeat-Until` statement is implemented analogously (Fig. 8) with two key differences. The loop process `p` is executed once before checking the loop condition `c`. Secondly, in the `Repeat-Until` construct, the condition `c` is a termination condition, such that the loop terminates when *c* is true. Therefore, in the `do` loop, the loop process is executed if the condition is not satisfied.

5.1 Verifying Interaction between Data and Control Flow

Data and control flow can often interact in unexpected ways. The simple process model depicted in Fig. 9 shows one such interaction that may prove harmful. The figure depicts a choice process, named `Browse`, that can be realized by either an atomic process named `ArtistSearch` or by an atomic process named `AuthorSearch`. A data flow link exists between the output of `ArtistSearch` to the input of `AuthorSearch`. Although this Process Model is legal in OWL-S, it is flawed. This is because either `AuthorSearch` or `ArtistSearch` is executed, but not both. Thus, whenever `AuthorSearch` is executed, `ArtistSearch` is not and therefore the input to `AuthorSearch` is never instantiated.

The Promela model generated by the mapping described thus far, would detect the harmful interaction between control flow and data flow. The model of the choice statement specifies that one of the two atomic processes will execute, while the `assert` constraint on the input of `AuthorSearch` requires that `ArtistSearch` is always instantiated. Since there does not exist a model where both claims are simultaneously true, Spin reports an error. The ability to detect such interactions between data flow and control flow in OWL-S Process Models is one of the main contributions of this work, which goes beyond other verification models constructed for OWL-S. Indeed we claim that the model provided by Narayanan et al. [9], would not detect the flaw in the process model described above.

5.2 Summary of the OWL-S Model construction

This section presented a detailed description of the modeling of OWL-S Process Models in the Promela modeling language. A summary of our modeling is presented in Table 1, highlighting the OWL-S Process Model features retained, partially modeled and the features out of scope. We already discussed how the checking of data values could be deferred to a type-checker or semantic reasoner. Thus, while we represent inputs and outputs, we do not represent their values or their (ontological) data-type, limiting ourselves to modeling assignment.

Similarly, we do not model OWL-S preconditions and effects. Preconditions to OWL-S processes are essentially warnings, that if the preconditions are not heeded, the execution of the process may fail. If the client is a hard-coded process, then the preconditions and effects serve as warnings or information for the programmer; on the other hand, if the client is an agent, the OWL-S preconditions and effects are for the benefit of the agent's planner. However, there is nothing to prevent a client from ignoring the preconditions, trying to execute the process, and possibly failing. Thus, OWL-S preconditions and effects do not affect execution of the Process Model. Consequently, they do not affect the verification of the Process Model either. Nevertheless, if a client wishes to ensure that the preconditions can be fulfilled in addition to verifying the Process Model, a promising approach might be to use a planner based on model-checking [4].

6 Verification of the Amazon example

Given a Promela specification of an OWL-S Process Model, Spin constructs a verifier, that can check several claims on the execution of the Process Model. These properties include the values of certain variables at certain points in the code and true statements that can be made about execution states (state properties) or the paths of execution (path properties). In addition, since Spin searches the entire state space of a verification model, it can also identify unreachable or dead code in a Process Model.

In this section, we present various kinds of verification that can be performed on a Promela model generated by the mapping described in the sections above. Using Spin and the Promela specification presented in the previous section, several properties of the execution of the Amazon OWL-S Process Model were verified. These properties were verified as part of five tests described below. For

Full Modeling	Partial Modeling	Out of scope
Processes Control Flow Concurrency Data Flow Loops	Conditions (non-deterministic choice) Inputs/Outputs (model assignment)	Preconditions and Effects Data Values

Table 1. Summary of the Modeling of OWL-S Process Models in Promela .

each test, the size of the model constructed by SPIN, the time taken in seconds to construct the model and the time for verification were measured[5].

1. *Simple Amazon*: In the first case, the PROMELA specification of the Amazon.com Web service was checked for basic safety conditions, such as the absence of deadlocks and the correctness of the data-flow within the model which derive directly from the mapping reported in the previous section.
2. *Data flow*: To the simple Amazon model, we added an `assert` statement to verify the data flow between the `Browse` and `ProductFound` processes. The statement specifies that `Browse` must return a product before the product is added to the shopping cart, i.e. before `ProductFound` executes the process `AddToShoppingCart`.
3. *Liveness*: Several interesting liveness claims can be made about the Amazon example. For example, a client may wish to verify that the Amazon Web service will always complete and not execute in an infinite loop, before deciding to use it. In other words, the user would like to express the requirement that "`ShopBook` process will eventually complete." In LTL this statement is expressed as: $\Diamond\mathtt{Done\_ShopBook}$. Another liveness claim a client may wish to verify is that if a desired product is found with Amazon, then the client can always add it to the shopping cart. This can be expressed as "in every execution sequence in which a product was found, the next process to be executed is `AddToShoppingCart`." In LTL this statement is expressed as : $\Box(\mathtt{productAvailable} \rightarrow X(\Diamond\mathtt{Done\_AddToShoppingCart}))$. In other words, whenever `productAvailable` is true, in the next state, the `AddToShoppingCart` process will eventually complete.
4. *Loop-2 and Loop-3*: In order to test how loops could affect the performance of SPIN, we added a loop to the Promela model, which created multiple concurrent instances of `ShopBook`. In the cases of Loop-2 and Loop-3, two and three concurrent instances of `ShopBook` were created respectively.

The experiment shows that the verification of OWL-S Process Models that do not contain any loops can be done very effectively. This is an important result since we expect that the great majority of Process Models will be loop-free[6]. Narayanan et al. [9] shows that the complexity the verification of the

	#States	Model Construction Time	Verification Time
Amazon	132	0.20	0.01
Data flow	139	0.35	0.02
Liveness	345	0.15	0.04
Loop-2	654382	0.03	8.77
Loop-3	3902280	0.04	>7200

Table 2. Performance of OWL-S verification using SPIN (time in seconds).

[5] The tests were carried out on a 750MHz Pentium 4 machine with 256MB of memory.

[6] The great majority of e-business sites available on the Web are loop-free. We expect that these sites provide a blue print for e-commerce Web services.

OWL-S Process Model with loops is PSPACE while the complexity of the same model without loops is NP-complete. Consistent with Narayanan's claim, the search complexity increases greatly, when the OWL-S Process Model is augmented with additional loops. However, it should be pointed out that the loops we constructed are among the most difficult to verify since they spin off two concurrent executions of the Amazon's Process Model. Sequential executions of Process Models would certainly exhibit less interaction.

The exponential increase in number of states and verification time, while troublesome, seems to be manageable since checking more than two concurrent instances of `ShopBook` is superfluous and violates the requirement that the verification model be the minimum sufficient model to perform the verification successfully. Verifying two concurrent instances of `ShopBook` reveal all the dangerous interaction effects just as well as three concurrent instances do. Therefore, we do not gain in verification power by checking more than two instances. In our future research we will search for a better modeling of loops that will minimize the state explosion that has been revealed by our experiments.

7 Conclusions

In this paper we proposed a procedure for the verification of correctness claims about OWL-S Process Models. We described a mapping of OWL-S statements into equivalent Promela statements that can be evaluated by the Spin model checker. In the process, a number of abstractions were presented for OWL-S Process Models. The abstractions reduce the complexity of verification while producing a model that is sufficiently rich to be able to make useful claims about OWL-S Process Models.

The work presented here is a starting point and we see numerous possible extensions to it. For instance, we intend to relax some of the modeling abstractions to report a richer output. In particular, we would like to specify not only the reachability of states, but also under which conditions a state is reachable. This information is important for a Web service client because it typically needs to know what information must be sought in order to guarantee a correct execution of the Process Model and what kind of commitments it will have to make. To this extent we are currently exploring the use of a different verification system, specifically NuSMV [3] which may allow a natural representation of conditions.

Another extension of this work that we would like to pursue is the automatic generation of liveness claims. Based on the OWL-S markup and an appropriate services ontology, a Web service client should be able to reason about processes in an OWL-S Process Model, generating claims on-the-fly, such as "the `Delivery` process always executes after the `Buy` process." These claims can then be verified before the client decides to invoke the Web service. There are multiple sources of liveness claims; in this paper we tested the reachability of one particular state, but the client of a service may also want to verify the correctness with respect to policies that the client has to satisfy.

Finally, this work does not include any modeling of the interaction between the client and the server. We intend to extend the verification to the data mappings specified in the OWL-S Grounding. Such verification may provide guarantees on the data that processes will receive from the Server. In this direction the work proposed in [13, 6] is of particular interest since it may provide a representation of the mapping between the XML data that Web services exchange with the OWL based data representation used in the OWL-S Process Model.

References

1. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Specification: Business process execution language for web services version 1.1. http://www.ibm.com/developerworks/library/ws-bpel/, 2003.
2. A. Ankolekar, M. Paolucci, and K. Sycara. Spinning the OWL-S Process Model– Towards the verification of the OWL-S Process Models. Presented at the *Semantic Web Services: Preparing to Meet the World of Business Applications* workshop at the *International Semantic Web Conference (ISWC 2004)*, Hiroshima, Japan, 2004.
3. A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proceeding of International Conference on Computer-Aided Verification (CAV 2002)*, Copenhagen, Denmark, 2002.
4. A. Cimatti and M. Roveri Conformant Planning via Symbolic Model Checking. In *Journal of Artificial Intelligence Research*, 31, pg. 305–338, 2000.
5. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, Cambridge, MA, USA, 2000.
6. X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *Proceedings of the 13th International World Wide Web Conference (WWW'04)*, New York, NY, USA, 2004. ACM Press.
7. H.Foster, S. Uchitel, J. Kramer, and J. Magee. Model-based verification of web service compositions. In *Proceedings of the Automated Software Engineering (ASE) Conference 2003*, Montreal, Canada, October 2003.
8. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, 2003.
9. S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the Eleventh International World Wide Web Conference (WWW-11)*, May 2002.
10. M. Paolucci, A. Ankolekar, M. Srinivasan, and K. Sycara. The DAML-S virtual machine. In *Second International Semantic Web Conference*, Sanibel Island, Florida, USA, 2003.
11. K. Schmidt and C. Stahl. A petri net semantic for bpel4ws - validation and application. In *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets (AWPN '04)*, Paderborn, 2004.
12. C. Walton. Model checking multi-agent web services. In *Proceedings of the 2004 Spring Symposium on Semantic Web Services*, Stanford, CA, USA, March 2004.
13. T. B. X. Fu and J. Su. Wsat: A tool for formal analysis of web services. In *Proceedings of the 16th International Conference on Computer Aided Verification*, 2004.