Spinning the OWL-S Process Model*

Toward the Verification of the 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 a modeling and verification procedure for 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. A Web service interaction protocol may be verified by a service provider to ensure that the advertised protocol 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. A Web service client may want to verify an 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 are delivered to the client.

In this paper, we explore the verification of OWL-S[1] interaction protocols using automatic verification tools, such as SPIN [1]. OWL-S is a well-established language 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 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.

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

Previous work on OWL-S verification is scant. Narayanan et al., in [2], propose a Petri Net-based operational semantics, which models the control flow of a Process Model exclusively[2]. On the basis of this mapping, a number of theorems on the computational complexity of traditional verification problems such as reachability of states and discovery of deadlocks are proven. The results show that the complexity of the reachability problem is PSPACE-complete. This result is not surprising given the complexity of the OWL-S Process Modeling language.

We extend on Narayanan's seminal work in three directions. First we provide a model of data flow in addition to control flow. As a result the verification procedure can detect harmful interactions between data and control flow that would be undetected otherwise. An example of these interactions in given is section 4.3. Second, as part of our modeling methodology, we define a set of abstractions that remove unverifiable details. The use of abstractions results in a simpler model that nevertheless preserve all the essential details that are required for verification. These abstractions were implicitly used in Narayanan's work, but not explicitly specified. 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.

The rest of the paper is organized as follows. In section 2, we will provide a quick overview of OWL-S 1.1 through a running example based on the Amazon Web service. In section 3, we will provide an introduction to verification with Spin. In section 4 we will provide a mapping of the example from OWL-S 1.1 to the PROMELA language which is used to construct models that the SPIN system can analyze. In section 5, we will provide examples of verification using SPIN. Finally, in section 6 we will discuss our results and future work.

2 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 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 sequence to describe processes that follow each other, spawning of concurrent processes, synchronization points between concurrent processes, conditional statements, non-deterministic selections of processes.

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

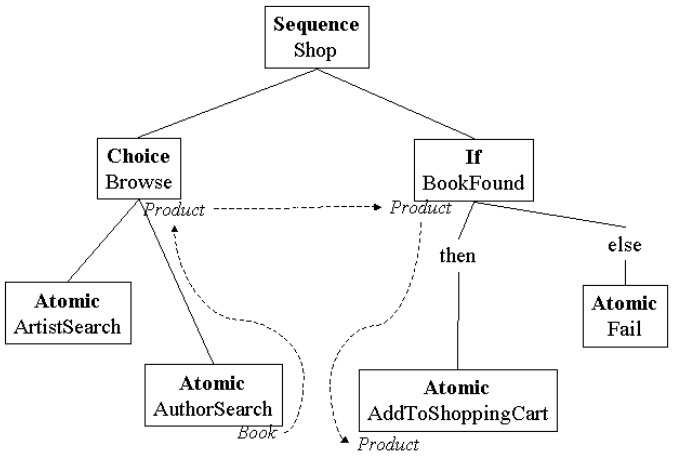

Fig. 1. The Process Model of Amazon.com's Web service

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 fragment of the Process Model adopted by Amazon.com's Web service. The nodes of the tree correspond to composite processes that represent different 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 by that author that have been found.

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.

3 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, a method that has been fairly successful in the verification of distributed systems, such as Web services (e.g. BPEL [4]). 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 three decisions to be made. The first one is to decide *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 to abstract*, in other words which aspects of the protocol are relevant to the claims to be verified, and which ones can be abstracted away as irrelevant. Irrelevant aspects of the protocol may include those that can be verified better in other ways, for instance type safety can be ensured using a type checker. Moreover, eliminating irrelevant or unverifiable aspects of the protocol reduces the complexity of the model, thereby increasing the likelihood that an exhaustive analysis of all possible execution traces can be done successfully. 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.

The third decision to be made is *how to model the protocol*, in such a way that the model preserves the behaviors to be checked. Thus, generating a *verification model* for an interaction protocol entails the translation of the protocol into a formal specification, which encapsulates the modeling decisions, i.e. implements the decided-upon abstractions 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. 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.

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). In the rest of this section we provide an overview of Promela, an insight in LTL and the type of claims can be formulated in Promela. The presentation will necessarily be shallow, concentrating only on the details that are relevant for our work. The readers are referred to Chapters 3 and 6 of [1] for a more comprehensive discussion of Promela and LTL.

3.1 Promela

Promela is a high-level specification language which has been designed for the representation of software models for Spin. A model in Promela consists of a set of asynchronous processes. Processes in Promela are introduced by the `proctype` keyword, as in `proctype A()` and are instantiated with the `run`

operator. Instantiated processes are inherently concurrent. Thus, the sequence `run A(); run B()` spawns off two concurrent threads: one computing `A()`, the other `B()`. If processes are to be executed in a particular order, e.g. in a sequence, they must be explicitly synchronized.

To define processes, PROMELA provides a set of basic control constructs that include selection statements that can be used to model conditionals as well as non-deterministic choices depending on whether a boolean guard condition is provided. In addition, PROMELA provides iteration statements that can be used to model loops; other control constructs also provided are goto statements (with process-local scope) and break statements.

Data flow is supported by two different constructs: variables and message channels. Variables can be used to model data assignment, while channels are used to model data flow between processes. Both variables and channels are either globally scoped or locally scoped within a single process. Furthermore, PROMELA supports variable and channel typing. 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. Therefore, in addition to data flow, channels can also be used to implement multi-process synchronization. As we shall see below, this synchronization mechanism will be exploited to implement OWL-S sequences[3].

All the traditional programming language types such as `int`, `char`, `boolean`, arrays and records are provided. In addition, PROMELA supports a form of enumerated type called `mtype`, which is typically used to describe message types. PROMELA does not support an OOP-style (Object-Oriented Programming type hierarchy. This limitation affects our modeling of OWL-S Process Models, as explained below.

3.2 Expressing claims in PROMELA

While SPIN always perform the verification of basic properties of the model, such as the deadlock detection, it also support the specification of model-specific correctness claims. These claims specify what should be true in a given state of execution of the model, but also the can be used to specify what kind of invariance and state reachability properties should the model have. Claims on states are expressed through the `assert(p)` statement that specifies that a property `p` should be true in a given state. As discussed below in section 4.2, we will use `assert` statements for the verification of data-flow by requiring that inputs that are targets of data-flow links should always be instantiated.

[3] 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 breaks down in certain cases. 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 invariably break down when multiple instances of processes can be spawned dynamically.

Claims on invariance and reachability of states are expressed in Linear-time Temporal Logic (LTL), a formalism for describing sequences of transitions between the states of a reactive system [3]. LTL allows the definition of logic formulae, such as ◇`p` that specify that `p` will *eventually* be true (in a future state); invariance claims such as □`p` to specify that `p` will *always* be true; and error conditions with `never(p)` specifying that `p` will never hold. These claims can be used to verify whether a given process is ever executed, or the relation between processes.

Since these claims are on the states or their reachability, the control flow of the distributed system is emphasized, rather than the data computations performed by the system. In fact, PROMELA is designed to discourage the specification of computations that are internal to a process.

4 Mapping OWL-S Process Models to 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 discuss the abstractions that we propose, and the mapping rules from the OWL-S Process Model to PROMELA statements.

4.1 Abstractions

Preconditions and Effects There is a profound difference between the execution environment of the OWL-S Process Model on one side and that of more traditional programming languages for which Spin has been constructed. Although the mapping of OWL-S Process Models to PROMELA [4] can map processes, inputs and outputs (to variables), and control flow constructs, there is no clear mapping for preconditions and effects. We believe this is because, unlike the case in typical programming languages, the execution of an OWL-S Process Model depends crucially on the knowledge of the agent executing it.

To appreciate the distinction being made, consider a `write` operation on a file. A precondition for this operation is the existence of the file, and the effect is that the argument of the `write` operation will be written to the file. Neither the precondition nor the effect are explicitly stated. Rather, most programming languages make an *implicit* assumption that the precondition is true, and if it is not, an exception will be thrown.

In OWL-S, preconditions are explicitly stated. The agent executing a process, evaluates the preconditions and checks their truth in its knowledge base. If the preconditions are true, the process can be safely executed. Otherwise, the agent may try to make those preconditions true, or it may defer executing the process or it may simply ignore them, hoping for the best. This ability, to make decisions based on preconditions and exploit the effects of other processes, has no counterpart in most programming languages. Specifically, preconditions and effects

[4] or any other programming language

cannot be modeled in PROMELA . As a consequence, we abstract preconditions and effects away in the PROMELA model constructed.

Conditions Besides preconditions and effects, conditions occur in `Result` statements and `if` statements, but these are treated somewhat differently. 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. There is nothing the client can do about it, nor could the condition have been evaluated ahead of time. From the point of view of software verification, such a condition could be considered a random variable which may equally be true or false. We therefore model `Result`s as a non-deterministic choice between the conditional outputs and effects.

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. Such conditions cannot be represented in SPIN . Furthermore, the knowledge state of the agent cannot be known ahead of time nor can it be inferred from the execution trace of previous statements. Therefore, as in the case of `Result`s, we abstract `if` statements as a non-deterministic choice between the `then` and `else` statements.

Data Flow Our final modeling abstraction relates to the content of the information exchanged in data flow links, and more generally, to the information represented by inputs and outputs. For a given data flow link that map 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 simply 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 that can methodically verify the integrity of all data flow links.

The various abstractions detailed above do not hinder the verification of other, more complex, claims that involve the interaction of data and control flow, as explained in section 4.3 below.

4.2 Modeling OWL-S Processes

In this section, we describe the mapping of OWL-S Process Models to PROMELA . Using the organization of processes in OWL-S, we first present the mapping of generic composite processes, highlighting the basic data flow and control mechanisms. We then describe the mapping rules for the different types of control

```
(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

flow statements realized by composite processes, concluding with a description of atomic processes. Throughout this section, the Amazon process example (Fig. 1) will be used to illustrate the mapping rules.

Modeling Composite Processes OWL-S Processes map naturally onto processes in PROMELA. For example, Fig. 2 shows the result of the translation of the top-level `Shop` process to PROMELA. Since `Shop` is a top-level process, it does not take any arguments. Other process definitions shown in Fig. 4, such as `Browse`, require as input a channel for control flow, `syncChan`, and optionally an additional channel for data flow, named `dataChan`.

In OWL-S, Composite Processes have the responsibility for orderly execution of their child processes. Each parent process, therefore, creates a `syncChan` and a `dataChan` to be used by its child processes. 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. 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`, as shown in lines 5 and 8 respectively.

Modeling `Split` and `SplitJoin` Since processes in PROMELA are intrinsically concurrent, `Split` and `SplitJoin` 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 `SplitJoin` process would wait for the termination of the processes it spawned.

Since there are no `Split` and `SplitJoin` statements in the Amazon example, Fig. 3 shows a prototypical implementation of a `SplitJoin` 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 `SplitJoin` 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.

```
(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

Modeling Sequences While concurrent processes can be implemented in a relatively straightforward way, the modeling of OWL-S sequences requires explicit synchronization, much like the `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.

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 in which all guard conditions are true. The implementation of the `Browse` process, 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`. An OWL-S conditional would be implemented similarly, but with

the `if` condition as a guard to the `then` statement and an `else` guard to the `else` statement. According to PROMELA semantics, the `else` guard is only true, if all other guards are false.

```
(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. The `Browse` Process

Modeling Atomic Processes Finally, we present the mapping of an *atomic* process, which produces different results, to PROMELA. We abstract the condition associated with each result, rather 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 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 data flow have been represented in the samples code showed 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,

```
(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. The `AuthorSearch` Process

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 4) specifies a claim on the state reached, namely that the value of product should not be zero. In other words, it verifies that the input is instantiated to some value.

```
(1) proctype AddToShoppingCart (chan syncChan, dataChan) {
(2)   /* check whether there is something on the data channel */
(3)   int product; dataChan?product;
(4)   assert(product);
(5)   syncChan!_pid,done;
(6) }
```

Fig. 6. The `AddToShoppingCart` Process

Modeling Loops Although PROMELA supports loops, their interaction with the data flow mechanism introduced in OWL-S 1.1. is still unclear. Therefore, we defer the modeling of loops to future work.

4.3 Verifying Interaction between Data and Control Flow

Data and control flow can often interact in unexpected ways. The simple process model depicted in Fig. 7 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.

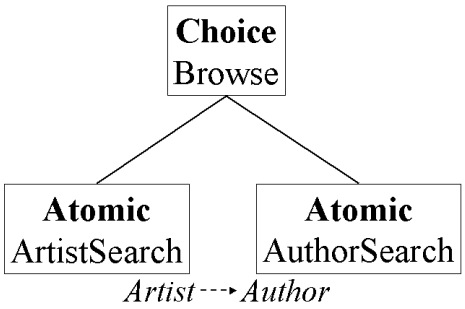

Fig. 7. An example of interaction between data and control flow in OWL-S

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 would report 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. [2], would not detect the flaw in the process model described above.

5 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 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].

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

	#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 1. Performance of OWL-S verification using SPIN (time in seconds).

1. *Simple Amazon*: In the first case, the PROMELA specification of the Amazom.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 follows:

   $$\Diamond \texttt{Done\_ShopBook}$$

   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 follows:

   $$\Box(\texttt{productAvailable} \rightarrow X(\Diamond \texttt{Done\_AddToShoppingCart}))$$

   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`. Narayan et al. [2] shows that the complexity the verification of the OWL-S Process Model with loops is PSPACE while the complexity of the same model without loops is NP-complete. Although we do not officially model OWL-S loops because their interaction with the data flow is yet to be completely worked out, we tried to estimate the effect of adding loops to our model by forcing the search process to loop until a product is found. 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.

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.

6 Conclusions and Future Work

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.

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.

References

1. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
2. 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.
3. E. M. Clarke, O. Grumberg and D. A. Peled. *Model Checking*. The MIT Press, 2000.
4. X. Fu, T. Bultan and J. Su. Analysis of interacting BPEL web services. In *Proceedings of the Thirteenth International World Wide Web Conference (WWW-13)*, May 2004.