# On the Verification of Behavioral and Probabilistic Web Services using Transformation

Giti Oghabi, Jamal Bentahar
*Engineering and Computer Science*
*Concordia University*
*Montreal, Canada*
*g_ogha@cse.concordia.ca,*
*bentahar@ciise.concordia.ca*

Abdelghani Benharref
*Engineering and Computer Science*
*Abu Dhabi University*
*Abu Dhabi, UAE*
*abdelghani.benharref@adu.ac.ae*

***Abstract*—In this paper, we propose a preliminary approach for automating web service verification. We use Semantic Markup for Web Services (OWL-S) to describe web service behavior. We parse the OWL-S file and transform it automatically to a corresponding Markov chain diagram or Markov decision process, which are then transformed to a PRISM model to be used as input by PRISM, a probabilistic model checker, to verify automatically the web service behavior. We provide an implementation of the transformation algorithm through a developed software tool automating all the transformation and verification activities.**

***Keywords*-Web Service; Verification; Model Checking;**

## I. INTRODUCTION

Web services are used vastly in different areas and they are playing a great role in IT industry nowadays. Similar to any other software system, soundness and correctness are main concerns of web service technology. Web service verification increases the trustworthiness of a web service and decreases its failure rate. The goal of web service verification is to assure that the web service satisfies all the expected requirements. Some of these requirements are general such as safety, liveness and deadlock freedom and some of them are specific, business-logic properties. Web service verification can give confidence to a provider before publishing its web service. It has also a significant usage in web service compositions and in building and managing communities of web services [1]. As web service composition make use of different web services and malfunctioning of one of them can lead to the whole system failure, we can prevent this situation by verifying each single web service. Furthermore, in a community of web services [2], a community master can accept or reject a web service after its verification.

In this paper, we propose a model checking-based approach for automating web service verification. Model checking is a formal and fully automatic technique that aims to check whether or not a system model $M$ satisfies a given property $\phi$ (i.e. $M \models \phi$). Among different model checkers such as Spin, NuSMV, MCMAS, etc., we have chosen PRISM. PRISM is a probabilistic model checker for the modeling and analysis of systems, which can, but not necessarily, behave in a probabilistic way [3], [4]. As argued in [5] and [6], web services can manifest not only deterministic, but also nondeterministic behavior, which justifies our use of the PRISM model checker. In fact, in a process model, some of the processes occure defenitely and some of them may occure with certain probability. For example in an $If - Then - Else$ construct, the probability for $If$ branch to occure may not be equal to the $Else$ branch. In a deterministic model checker, we cannot consider these uncertainty issues, but in PRISM since we can have a probability for each state, we can calculate the probability of reaching a state or satisfying a condition. As each model checker needs the system model $M$ as an input to be verified, we should build a model for the considered web service. This model can be extracted from the web service programming code, but since we are looking for a standard automated method regardless of web service programming languages and platforms, we have chosen OWL-S for web service description. OWL-S ontology is referred to as a language for web service description, which provides a standard vocabulary [7]. Besides OWL-S, there are two other Web Service Description Languages: WSDL and BPEL. WSDL defines the interface of a service as a collection of network endpoints and contains some elements such as types, messages, operations, ports, etc. [8]. However,WSDL does not contain useful information about web service behavior. Thus, it cannot be used to construct the system model to be checked. On the other hand, BPEL (Business Process Execution language) models a business process as a composition of elementary web services [8]. BPEL models then a set of web services, but what we are interested in is a language, which describes single web services independently.

In our approach, the OWL-S file of each web service is used as input to our software tool, which, after parsing the file, makes a probabilistic model (Markov chain diagram or Markov decision process) out of it. Thereafter, the tool automatically creates a PRISM file for the created probabilistic model. The PRISM file is then used as input for the PRISM

978-0-7695-4463-2/11 $26.00 © 2011 IEEE
DOI 10.1109/ICWS.2011.106

model checker to verify, not only the properties the web service is required to satisfy, but also with what probability. The focus of this paper is on the approach of extracting the Markov chain diagram and Markov decision process out of the OWL-S file and the generation of the PRISM source code from the related Markov chain/decision process.

We should mention that in OWL-S standard definition, probablilities are not included. However, as proposed in [9], we can show the probability of each process as a parameter in its definition.

This paper is organized as follows. In Section II, we review related work and compare the present work with. In Section III, we describe OWL-S, its different parts, and the logic behind it. A brief explanation about the PRISM model checker is also given. The rules and algorithm to transform an OWL-S model to Markov chain diagram and then to the PRISM language, which is the main part of this research, is discussed in Section IV. We have chosen the classic "`CongoProcess.OWL`" [7] as a motivating example to describe the approach. In Section V, we discuss some common properties, which are checked in our example. Section VI focuses on the tool implementation and experiments. Section VII concludes the paper and identifies directions for future work.

## II. RELATED WORK

Many research proposals about automating web service verification have been published in the recent ten years. However, they are mostly focusing on automating web service composition verification and are consequently based on BPEL4WS [10], [11]. Only a few initiatives about web service verification based on OWL-S have been lately launched.

In [12], Lomuscio et al. investigated the transformation from OWL-S to ISPL, a process model language for the MCMAS model checker. They have proposed some transformation rules of OWL-S control constructs and implemented the "sequence" control. We extended and adapted some of these rules to our work to fulfill the needs for transforming OWL-S control constructs into Markov chain diagram and decision process. However, unlike Lomuscio et al's proposal, we implemented not only the "sequence" control, but all the control constructs including "choice", "if-then-else", "repeat-while", etc. In [13], Ankolekar et al. have also applied automatic tools for the verification of web services. However, their main focus is not on individual web services, but rather on the interaction protocols of these web services. Unlike [12] and [13] where the MCMAS and SPIN deterministic model checkers have been used, we use PRISM, which can check not only deterministic behaviors, but also probabilistic properties of nondeterministic systems. This allows analyzing some useful features, for example calculating the probability of satisfying some properties, such as reachability and deadlock freedom, and other business logic properties.

In fact, in PRISM model checker, there is a feature for detemining the probability of each state occurance, so we can show not only the differernt sates, but also the probability of their occurance. By having these probabilities in the model, PRISM can calculate and predict the probability of reaching a specifice state, (reachability analysis), probability of having a deadlock, etc. Having these properties is extremely useful in decision making and evaluation of web services. For example, in a community of web services, the community master can decide to accept a web service if it satisfies one specific property with more than a given probability value.

In [14], Cao et al. presented a methodology for passive testing of behavioral conformance for web services. However, unlike our proposal, the paper only focuses on security issues. In [15], Liu et al. developed a model checking framework for web services based on OWL-S. In their approach, the authors introduce some rules for transforming OWL-S model to a TCPN (Time Constraints Petri Net) model. Our transformation rules for changing OWL-S model into Markov chain diagram and decision process are somehow similar to these rules, but the resulting models are totally different. Markov chain diagram and decision process are probabilistic models that can model deterministic as well as nondeterministic dynamic systems and properties, which makes them richer than TCPN. Furthermore, we present an algorithm and a fully implemented tool to perform the modeling and transformation automatically. In [16], Narayanan et al. presented an interpreter, which takes web service description in form of DAML-S as an input and generates automatically a Petri Net and performs the desired analysis. This work is different from ours as its main focus is mostly on web service composition. In addition, our approach is based on OWL-S, which is an improved version of DAML-S.

## III. PRELIMINARIES

### A. *Web Service Ontology Language*(*OWL-S*)

OWL-S ontology is referred to as a "language for describing services and providing a standard vocabulary that can be used together with the other aspects of the OWL description language to create service descriptions" [7]. An OWL-S consists of three different classes: service profile, service grounding, and service model. The service profile tells "what the service does", so that it becomes suitable for a service-seeking agent. Service grounding specifies the details of how an agent can access a service. It typically contains a communication protocol, message formats, and other service-specific details such as port numbers. Service Model gives details on how to ask for the service and describes what happens when the service is executed [7]. This description is of a great importance as it helps a

service-seeking agent to know whether the service satisfies the requirements or not; to compose individual services to perform a complex task; and to monitor the service execution [7]. Since our concern in this research is the way services behave with clients, we principally use the service model class.

In this class, a service can be viewed as a process. A process specifies how a client interacts with a service [7]. There are two types of processes: **Atomic** and **Composite**. Atomic processes describe a single interaction, but composite processes describe the actions, which require multi-step protocols [7]. Execution of atomic processes are in a single step, but composite ones are executed by decomposition into other(atomic or composite) processes; their decomposition can be specified by using control constructs, namely *sequence*, *choice*, *if-then-else*, *repeat-while*, *repeat-until*, *split*, and *split-join*. These control constructs have almost the same name of control structures in programming languages, but they have an essential difference as they do not describe the service behavior, but the way a client may interact with a service.

### B. PRISM: A Probabilistic Model Checker

PRISM is a tool for modeling, verifying, and analyzing systems, which exhibit probabilistic behaviors [4]. Probabilistic model checking is a formal verification technique according to which a mathematical model of the system is constructed and then analyzed by the model checker. The properties of the system are expressed formally in a probabilistic or non-probabilistic temporal logic and checked against the constructed model automatically.

PRISM supports three types of probabilistic models: Discrete-Time Markov Chains (DTMC), Markov Decision Processes (MDP), and Continuous-Time Markov Chains (CTMC). Models are described by the PRISM language, a simple, high level modeling language. Properties are written in PRISM property specification language, which is based on several well-known temporal logics: PCTL, LTL, CTL, PCTL* [17]. In our approach we use DTMC and MDP for modeling the systems and PCTL logic for describing the properties.

## IV. TRANSFORMING OWL-S MODEL TO DTMC, MDP, AND THE PRISM LANGUAGE

To consider the transformation of OWL-S into DTMC, first we give the formal definition of this formalism [17].

***Definition** 1:* A discrete-time Markov chain (DTMC) is a tuple $M = (S, P, \iota_{init}, AP, L)$ where:

- $S$ is a countable, nonempty set of states;
- $P : S \times S \to [0, 1]$ is the *transition probability function* such that for all states $s : \sum_{s' \in S} P(s, s') = 1$;
- $\iota_{init} : S \to [0, 1]$ is the *initial distribution*, such that: $\sum_{s \in S} \iota_{init}(s) = 1$;
- $AP$ is a set of atomic propositions and $L : S \to 2^{AP}$ a labeling function.

Each Markov chain has a set $S$ of states and a transition probability function $P$, which specifies for each state $s$ the probability $P(s, s')$ of moving from $s$ to $s'$ in one step by a single transition. Also $\sum_{s' \in S} P(s, s') = 1$, is a constraint for $P(s, s')$, which shows $P(s, s')$ is a distribution. It means if we have $n$ transitions from $s$ to different states, the total sum of their related probabilities will be 1. A state $s$ with $\iota_{init}(s)>0$ is considered as initial state. In this section, we describe how we extract this needed information out of the input OWL-S file and make a Markov chain diagram. However, there are cases of OWL-S control constructs, where transitions are labeled with actions, which cannot be captured by DTMC, but rather by MDP [17].

***Definition** 2:* A discrete-time Markov decision process (MDP) is a tuple $M = (S, Act, P, \iota_{init}, AP, L)$ where:

- $S$ is a countable, nonempty set of states;
- $Act$ is a set of actions;
- $P : S \times \ Act \ \times S \to [0, 1]$ is the *transition probability function* such that for all states $s \in S$ and actions $\alpha \in Act : \sum_{s' \in S} P(s, \alpha, s') = 1$;
- $\iota_{init} : S \to [0, 1]$ is the *initial distribution*, such that: $\sum_{s \in S} \iota_{init}(s) = 1$;
- $AP$ is a set of atomic propositions and $L : S \to 2^{AP}$ a labeling function.

An MDP has approximately the same definition as an DTMC with some differences, which we explain as follows: MDP has an extra element, $Act$, which is a set of actions. Actions can be considered as labels for transitions. Transition probability function $P$ shows on the contrary of MDP that $P(s, \alpha, s')$ is a distribution. It means if we have $n$ transitions from $s$ to different states with action $\alpha$, the total sum of their related probabilities will be 1. This is the reason behind using MDP and not DTMC to model parallel actions since it allows sets of distributions per action and state rather than just a single distribution. Consequently, we use DTMS for modeling all the control constructs except Split and Split-Join, which we model using MDP.

To explain our approach in more details, we have chosen a concrete example of OWL-S file named `CongoProcess.owl` from OWL-S standard web site [7]. It is about a web service for buying online books. First, we illustrate the file structure by the following BNF grammar where atomic processes are enclosed in "{}", "+" shows the *sequence* control construct, and "," shows the *choice* control construct:

1) $FullCongoBuy \to \{LocateBook\} + CongoBuyBook$
2) $CongoBuyBook \to BuySequence + \{SpecifyDeliveryDetails\} + \{FinalizeBuy\}$
3) $BuySequence \to \{PutInCart\} + SignInAlternatives + \{SpecifyPaymentMethod\}$
4) $SignInAlternatives \to CreateAcctSequence, SignInSequence$

5) $CreateAcctSequence \rightarrow \{CreateAcct\} + \{LoadUserProfile\}$
6) $SignInSequence \rightarrow \{SignIn\} + \{LoadUserProfile\}$

All control constructs except "$SignInAlternatives$", which is a *choice* control, are *sequence* control. Associated with the *choice* control is an *if-then-else* control. We can also include *repeat-while* and *repeat-until* by simply looping on "$FullCongoBuy$". We use this example and some other prototypical examples to describe how we model each of the OWL-S processes and control constructs into DTMC and MDP in the following subsections.

### A. Modeling Atomic Processes

Since each atomic process represents a programming unit, which receives some inputs and produces some outputs, this process can be transformed in Markov chain as a state, which will be reached based on the control structure of the program. In our `CongoProcess` example, we can consider each atomic process as a single state. Thus, $LocateBook$, $PutInCard$, $SignIn$, $CreateAcct$, $LoadUserProfile$, etc. are DTMC states.

### B. Modeling Composite Processes

Each composite process is made of some other processes, atomic or composite, which are arranged according to their relevant control constructs. For example, $SignInSequence$ is a *sequence* composite process, which is made of two atomic processes: $SignIn$ and $LoadUserProfile$. To transform composite processes into a DTMC diagram, we replace the current process with its composed processes and the transition between the composed processes depends on the related control construct. In the following subsection, we describe in detail how each control construct defines the transition between its composed processes.

An important point to be mentioned here is if the new replaced processes are composite too, similarly they should be replaced by their composed processes and this algorithm should be repeated until all present processes are atomic. At this point, we can consider each process as a DTMC state and the whole structure is our final DTMC diagram. This recursive algorithm will be presented later in Section VI.

### C. Modeling Control Constructs

*1)* ***Sequence****:* It shows a finite set of processes, which are executed in a sequential order. Thus, we can model a sequence of processes using states sequentially related to each other through transitions. For example, $SignInSequence$, a composite sequence process, is shown by a DTMC having two states named *SignIn* and *LoadUserProfile* and a transition between them. Since there is only one possible transition from *SignIn*, the probability of this transition is 1. The corresponding DTMC diagram is shown in Figure 1.

To produce the related PRISM code for each DTMC diagram, we use Algorithm 1. According to this algorithm,

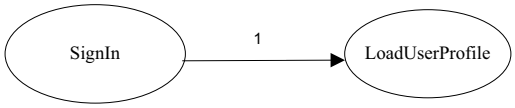


Figure 1. Modeling Sequence into Markov Chain

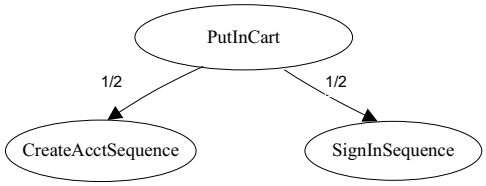


Figure 2. Modeling Choice into Markov Chain

the PRISM code is as follows:

```
s: [0..1] init 0;
[] s=0 → 1:(s'=1);
```

**Algorithm 1** *From DTMC to PRISM*

***a***) *Define an integer variable having a maximum value equal to the number of states, then initialize it to* 0 *using the PRISM syntax.*

***b***) *Show all transitions from one state to some other states by a PRISM command, which starts with* `[]` *and comprises a guard and one or more updates. In the guard part, make the source state true. Then, create an update part for each destination state separated by* "`+`"*. In each update part, first put the related probability followed by* "`:`"*, then make the destination state true and connect these two states by* "`&`"*.*

***c***) *If there is a condition for a transition, then make the condition true in the guard part of the command.*

*2)* ***Choice****:* In this construct, the caller process has different processes as options to select from, but only one is chosen. Thus, we substitute a composite choice construct with its inner processes and we suppose that the probability of each process is determined by a parameter in OWL-S, as suggested in [9]. In our example, $SignInAlternatives$, which is a *choice* composite process is replaced by $CreateAcctSequence$ and $SignInSequence$. So, $PutInCart$ will be transited to two states: $CreateAcctSequence$ and $SignInSequence$, which are substitutions for $SignInAlternatives$ state. Here we suppose the probabilities of 1/2 for $CreateAcctSequence$ and $SignInSequence$ processes (see Figure 2). It should be mentioned that since $CreatAcctSequence$ and $SignInSequence$ are composite process, they will be decomposed and substituted with other processes.

*3)* ***If-Then-Else****:* In this control construct, there are three sections: *if-condition*, *then*, and *else*. If *if-condition* is true, all processes specified in *then* section will be executed;

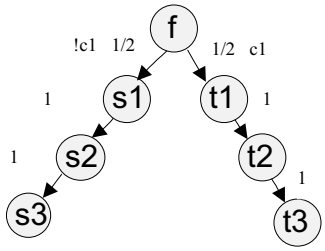


Figure 3. Modeling If-Then-Else into Markov Chain

otherwise, all processes specified in *else* section will be executed. To transform this construct to DTMC diagram, we create a state in which the *if-condition* is checked, and we add two transitions from this state: one reaches the first process specified in *then* part and the other reaches the first process in "else" part. As mentioned before, each atomic process is associated with a single state and each composite processes is replaced by a composite set of states. To explain this procedure, let us consider a prototypical example. Suppose we have a condition named $c1$ and if it holds, we want to execute the sequence processes of $t1$, $t2$, and $t3$, and if it doesn't hold, the sequence of $s1$, $s2$, and $s3$ should be executed. Then the resulting DTMC is as shown in Figure 3.

It should be mentioned that all the processes are considered atomic; otherwise, each composite process should be replaced with its related composed processes.

The equivalent PRISM source for transition from $f$ to $s1$ and $t1$ of this DTMC diagram is as follows:

```
s: [0..2] init 0;
[] c1=true & s=0 → (s'=1);
[] c1=false & s=0 → (s'=2);
```

Furthermore, since the rest of transitions are of type *sequence*, the equivalent code could be written as explained in the Sequence subsection.

*4)* ***Repeat-While, Repeat-Until****:* There are two types of loops in OWL-S: *repeat-while* and *repeat-until*. Their logics are the same as the while and repeat-until loops in programming languages. In *repeat-while*, the *while-condition* will be checked at first and if it holds, the *while-body* will be executed repeatedly until the *while-condition* becomes false. To transform this construct into DTMC, the *while* statement is associated with a state, which can transit to two different states (processes). If *while-condition* is true, this state is transited to the first process of *while-body*, which itself is a state. Otherwise, the state is transited to the next process after *while-body*, which is a single state (atomic process) or a composition of states (composite process). As a prototypical example, suppose we have a *repeat-while* loop. We name the *while* state as $w0$ and the *while-condition* as $c1$. The *while body* consists of processes named: $s1$, $s2$, and $s3$, and the next process after the loop body is $s4$. Again, we suppose

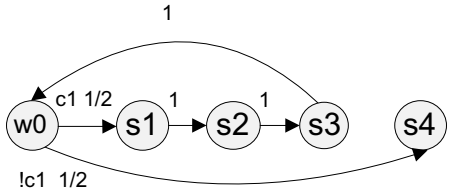


Figure 4. Modeling Repeat-While into Markov Chain

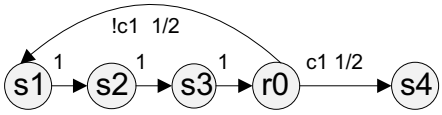


Figure 5. Modeling Repeat-Until into Markov Chain

that all the processes are atomic. The resulting DTMC is illustrated in Figure 4.

Associated with *repeat-while* is the following PRISM code:

```
s: [0..4] init 0;
[] c=true & s=0 → (s'=1);
[] s=3 → s'=0;
[] c=false & s=0 → (s=4);
```

A similar algorithm can be applied to the *repeat-until* control. The associated DTMC is shown in Figure 5, and its corresponding PRISM code is as follows:

```
s: [0..4] init 0;
[] c=true & s=3 → (s'=4);
[] c=false & s=3 → (s'=0);
```

## V. PROPERTIES AND VERIFICATION

Once the PRISM file is created as described in the previous section, the next step is to verify it against desirable properties through the PRISM Model Checker. To specify those properties, we use Probabilistic Computation Tree Logic (PCTL). The syntax and semantics of this language are given in [17]. In this section, we will give examples of some properties we have checked for the `CongoProcess` scenario. To make these properties readable, we define labels for each state so that we can use these labels instead of the state numbers. For example, for the state $s = 3$, instead of using 3, we use "SignIn" as its label.

- $P <= 0$ [$F$ "$deadlock$"]

In PRISM, there is a predefined "$deadlock$" label and the above property means: the probability of having deadlock in future ($F$) from the initial state of the model is less than or equal to 0 (which technically means equal to 0 and thus impossible). This is an example of safety property for the web service (absence of deadlock). The property is satisfied in our model.

- $P >= 1$ [$F$ "$terminate$"]

We define the label "$terminate$" in PRISM as follows: $Label$ "$terminate$"$= s = 7$;

Thus, this property means the probability of terminating the execution of the web service process by reaching "$finalizBuy$" state in the future is greater or equal to 1 (which technically means equal to 1 and thus certain). This property, satisfied in our model, captures the liveness property of the `CongoProcess` scenario.

- $!SpecifyPaymentMethod \mid !SpecifyDeliveryDetails \Rightarrow P <= 0\ [F\ \text{"}terminate\text{"}]$

This formula expresses the following business logic property: if the payment method or delivery details are not specified, then the probability of terminating by reaching an acceptance (final) state is less than or equal to 0. The property is satisfied in our model.

- $!BookInStock \Rightarrow P <= 0\ [Fs = 1]$

This business logic property, satisfied in our model, says: if there is no book in the stock, then no book can be added to the cart.

- $Pmax =?\ [PutInCart\ U\ SignIn]$

PRISM has also the ability to compute the probability of satisfying a property. For example, in the above formula, PRISM computes the maximum probability that one customer adds books in his cart continuously until ($U$) he signs in through the system. PRISM estimates this probability to be 0.5.

- $Pmax =?\ [F\ (SignIn\ \&\ CreateAcct)]$

This property estimates the probability that a user can sign in and at the same time create an account. This property is not satisfied in our scenario as it is an impossible situation. Consequently, the calculated probability is 0.

By specifying the whole desired business logic properties, we can check all possible paths (executions) in a specified model, calculate their occurrence probabilities, and verify the correctness of the model.

## VI. Implementation

We have implemented, using Java, the above approach in a software tool taking a web service description in form of an OWL-S file as input and producing DTMC/MDP and the PRISM source code as output. From the architectural perspective, the tool is composed of three different parts:

**Parser:** It reads the OWL-S file, parses it and recognizes its different components and control constructs to make an internal representation form of the file in the memory. This internal representation, which is made mostly by linked list structures is easily readable by the other parts. Our parser works for the OWL-S latest version 1.2, but since it is well parameterized, it can be easily adapted if new versions emerge. Furthermore, the internal representation, which we have chosen is independent from the input file format.

**DTMC and MDP Generator:** It reads the internal data structures created by the parser, analyzes it and makes a new and simpler representation, which we call "Internal Representation II". This representation is used later by the PRISM Code Generator. In addition to this, the generator creates the corresponding DTMC or MDP depending on the constructs included in the input file as explained in Section IV.

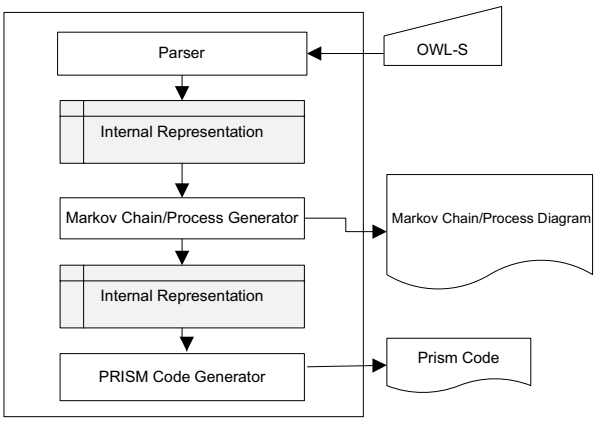


Figure 6. Tool Internal Architecture

**PRISM Code Generator:** It is the last part and produces the final PRISM code. It reads and analyzes the "Internal Representation II" created by the DTMC and MDP generator and makes a respective PRISM file. This is the final output, which is used by the PRISM model checker for verification. Figure 8 depicts the tool architecture showing its different parts. The system implementation is discussed next.

**Parser Implementation.** To implement the Parser part, we use a recursive algorithm. The whole idea of the algorithm is as follows. We read the elements of XML based OWL-S file and analyze them one by one. If the element represents an atomic process, we add the process to the list of atomic processes. If the elements represents a composite process, we add it to the list of composite processes and then analyze all its inner processes. For the inner processes, we repeat the same procedure. If there is an atomic process, we add it to the atomic process list, and if there is a composite process, we add it to the composite process list. Thereafter, we add these inner processes as children for the composite parent process. This recursive algorithm is repeated for each nested composite process. By executing this algorithm, a linked list structure named `Internal Representation I` is getting created to be used by the DTMC/MDP generator.

**DTMC/MDP Generator Implementation.** This part reads the `Internal Representation I`, analyzes it, and creates the `Internal Representation II` from which it creates DTMC/MDP diagrams. The algorithm of this part works as follows. It reads the composite processes list nodes one by one and extracts all paths that the composite process list keeps track of. Then, the algorithm stores the paths into the `Internal Representation II`, a multi dimensional linked list. It reads one composite pro-

cess node and applies `decomposeCompositeProcess` function on it. This function replaces the process with its equivalent processes. For example, if the type of composite process is *sequence*, it will be replaced by its inner processes, and if each of the inner processes is composite too, it will be substituted by composed processes analogously. This recursive procedure is repeated until all inner composite processes are replaced with atomic processes.

The returning parameter of this function is `resultList`, a multidimensional list, which stores all the paths. In this function, composite processes are analyzed based on their types, and all the steps described in Section IV are followed. If the type is *sequence*, we put all composed processes in `equivalentList` and then merge this list with `resultList`. For the types *repeat-while* and *repeat-until*, we follow the same steps. If the type is *if-then-else*, we put all processes related to *if* in `ifList` and all processes related to *else* in `elseList`. At the end, we merge these two lists into `resultList`. If the type is *choice*, we create a multidimensional list named `choiseList`, and put each inner process as a row in it to create multi paths. We merge then this list with `resultList`. As *split* and *split-join* have also different branches, we use the same steps for them. It should be mentioned that if each of the inner processes is composite, it should be decomposed similarly, so the function `DecomposeCompositeProcess` is invoked recursively. When the `resultList` is completed, we traverse the list and make a DTMC/MDP out of it. Each node of `resultList` shows one state of the created DTMC/MDP and the whole structure is our final MDP diagram.

**PRISM Code Generator Implementation.** This part reads the `Internal Representation II` and produces the respective PRISM code. The algorithm used here is the one we introduced in Section IV.

Figure 9 shows a part of the MDP that our tool produced for the `CongoProcess` scenario. We have also conducted many simulations for model checking this scenario by increasing the number of processes in each experiment. Results for $4$ experiments are shown in Table 1. The simulations have been conducted using a laptop running 32-bit Windows Vista with 3GB of RAM and Processor Core(TM)2 Duo CPU T$5850$, $2.17$Ghz. These results show that even with a large web service having $48$ processes, the model construction and verification times are still short even if the model size (number of states + number of transitions) is large.

## VII. Conclusion and Future Work

In this paper, we proposed a new approach towards verifying web services using a probabilistic model checking technique. we provided a complete and sound algorithm that automatically transform an OWL-S file to a DTMC/MDP and creates the equivalent PRISM code. The simulation results shows that the whole approach is promising in terms of execution time.

In terms of future work, we are investigating the extension of the present framework to consider composite web services, where nondeterministic choices and behaviors are important. We are also planning to apply this new technique to check communities of web services [18], where communication protocols between the community master and slaves are characterized by their uncertainty, which fits well with our probabilistic-based approach.

As mentioned earlier, in the current approach we assume that the OWL-S contains the probability for each process as a parameter, but we may design PR-OWL-S, which can be an extension of OWL-S similar to PR-OWL, which is the probabilistic ontology [19]. Developing and extending PR-OWL-S is another direction for future work.

## References


[1] J. Bentahar, Z. Maamar, W. Wan, D. Benslimane, P. Thiran, and S. Subramanian, "Agent-based communities of web services: An argumentation-driven approach," *Service Oriented Computing and Applications*, vol. 2, no. 4, pp. 219–238, 2008.

[2] B. Khosravifar, J. Bentahar, A. Moazin, Z. Maamar, and P. Thiran, "Analyzing communities vs. single agent-based web services: Trust perspectives," in *Proceedings of the 7th IEEE International Conference on Services Computing (SCC 2010)*. IEEE Press, July 05 - 10 2010, pp. 194–201.

[3] Prism model checker website. [Online]. Available: http://www.prismmodelchecker.org/

[4] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, "Prism: A tool for automatic verification of probabilistic systems," in *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, vol. 3920, 2006, pp. 441–444.

[5] D. Berardi, G. D. Giacomo, M. Mecella, and D. Calvanese, "Composing web services with nondeterministic behavior," in *Proceedings of the 2006 IEEE International Conference on Web Services (ICWS)*, 2006, pp. 909–912.

[6] Y. Sakata, K. Yokoyama, and S. Matsuda, "A method for composing process of non-deterministic web services," in *Proceedings of the 2004 IEEE International Conference on Web Services (ICWS)*, 2004, pp. 436–443.

[7] OWL-S: Semantic markup for web services web site. [Online]. Available: http://www.w3.org/Submission/OWL-S/

[8] M. Papazoglou, *Web Services: Principles and Technology*. Prentice Hall, 2008.

[9] H. Zhao and P. Doshi, "A hierarchical framework for composing nested web processes," in *International Conference on Service Oriented Computing*, ser. Lecture Notes in Computer Science, vol. 4294/2006, 2006, pp. 140–144.

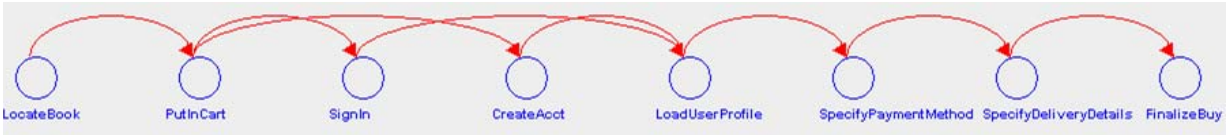


Figure 7. CongoProcess Markov Chain Diagram

Table I
OWL-S VERIFICATION RESULTS USING PRISM

| | **Experiment 1 24 Processes** | **Experiment 2 32 Processes** | **Experiment 3 40 Processes** | **Experiment 4 48 Processes** |
|---|---|---|---|---|
| **Number of States** | 512 | 4096 | 32768 | 262144 |
| **Number of Transitions** | 1728 | 18432 | 184320 | 1769472 |
| **Construction Time (sec.)** | 0.0070 | 0.012 | 0.046 | 0.094 |
| **Verification Time (sec.)** | 20.14 | 48.93 | 75.97 | 103.71 |

[10] S. Hinz, K. Schmidt, and C. Stahl, "Transforming BPEL to Petri nets," in *Proceedings of the 3rd International Conference on Business Process Management (BPM)*, ser. Lecture Notes in Computer Science, vol. 3649. Springer, 2005, pp. 220–235.

[11] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "Model-based verification of web service compositions," in *18th IEEE International Conference on Automated Software Engineering (ASE)*, 2003, pp. 152–161.

[12] A. Lomuscio and M. Solanki, "Mapping OWL-S processes to multi-agent systems: A verification-oriented approach," in *Proceedings of the International Conference on Advanced Information Networking and Applications Workshops*, 2009, pp. 488–493.

[13] A. Ankolekar, M. Paolucci, and K. P. Sycara, "Towards a formal verification of OWL-S process models," in *Proceedindg of the International Semantic Web Conference (ISWC)*, ser. Lecture Notes in Computer Science, vol. 3729, 2005, pp. 37–51.

[14] T.-D. Cao, T.-T. Phan-Quang, P. Felix, and R. Castanet, "Automated runtime verification for web services," in *Proceedings of the 2010 IEEE International Conference on Web Services (ICWS)*, 2010, pp. 76–82.

[15] R. Liu, C. Hu, and C. Zhao, "Model checking for web service flow based on annotated OWL-S," in *Proceedings of the Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, 2008, pp. 741–746.

[16] S. Narayanan and S. A. McIlraith, "Simulation, verification and automated composition of web services," in *Proceedings of the 11th International Conference on World Wide Web (WWW)*, 2002, pp. 77–88.

[17] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, Cambridge, Massachusetts, 2008.

[18] B. Khosravifar, J. Bentahar, A. Moazin, and P. Thiran, "Analyzing communities of web services using incentives," *International Journal of Web Services Research (IJWSR)*, vol. 7, no. 3, pp. 30–51, 2010.

[19] Z. Ding and Y. Peng, "A probabilistic extension to ontology language owl," in *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 4*, 2004.