2011

Model checking single web services using Markov Chains and MDPs

Giti Oghabi  
Concordia University

Jamal Bentahar  
Concordia University

Abdelghani Benharref  
Abu Dhabi University

Publication Details
Model Checking Single Web Services
using Markov Chains and MDPs

Giiti OCHABI a, Jamal BENDAHAR b and Abdelghani BENHARREF c

a Engineering and Computer Science, Concordia University, Montreal, Canada
b Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Canada
c Engineering and Computer Science, Abu Dhabi University, Abu Dhabi, UAE

Abstract. This paper proposes a new verification method for individual web service
specifications using OWL-S, the Ontology Web Language for semantic web Services,
where probabilities are introduced to model uncertainty. The approach makes use of a
probabilistic model checker named PRISM. The web service description is received as an
input in the form of OWL-S, which is analyzed and automatically transformed into a
Discrete Time Markov Chain (DTMC) or a Markov Decision Process (MDP). The obtained
DTMC/MDP is then processed and coded into the
PRISM language. This code is parsed and verified by the PRISM model checker
to determine which required properties are satisfied by the web service. In addition
to the atomic processes, different composite processes are considered including
sequence, choice, if-then-else, repeat-while, repeat-until, split, and split join. We
also prove that the transformation algorithm is sound and complete and introduce a
software tool implementing the approach.

Keywords. Web Service, Verification, Model Checking

Introduction

Fast growing and pervasive use of web services in business settings have led to a crucial
need for their verification. Web services need to be rigorously verified before launch-
ing them on the web to increase the confidence provider and consumers can have on
their behaviors. Verification means checking a web service in terms of safety, liveness,
deadlock freedom, etc. It also means checking if the web service satisfies some specific,
business-logic properties. Web service verification is of great importance not only when
they should interact with each other to provide composite services or within commu-
nities [1], but also when they are considered at the single level prior to any interaction
with other services or applications. Furthermore, since in composite scenarios, different
services are combined to fulfill some complex needs, the complexity of verifying
the composite service increases with the number of interacting services and the complexity
of each service. Consequently, the verification of each individual service decreases the
error rates and the ultimate effort of verifying the whole composite service. What is more
challenging is when the web services can exhibit a probabilistic behavior, which means
depending on the inputs and other circumstances, different choices are likely to be made
and different outcomes are likely to be produced.
This paper addresses the challenge of verifying individual probabilistic web services by extending a framework we have previously proposed in [2]. In the previous framework, we proposed a model checking-based approach for automating web service verification considering a restricted number of control constructs and a hypothetical and relatively small case study. In this paper, we consider two more control constructs, Split and Split-Join. Like in the previous framework, the proposed approach is based on transforming the web service process model described in OWL-S ontology, a language for describing services. However, unlike the previous work, a formal proof of the soundness and completeness of the transformation method is provided in this paper. A more complicated and concrete case study is also discussed. Furthermore, we assume in this paper that OWL-S is extended with probabilities, so we can extract these probabilities directly from the OWL-S file instead of assuming them in an ad-hoc way. Finally, the algorithm which the tool uses is presented.

Our framework is based upon model checking, which is a formal technique to verify if a system model $M$ satisfies a given property $\phi$ (i.e. $M \models \phi$). It has been argued in [3] and [4] that web services can exhibit not only deterministic, but also nondeterministic and uncertain behavior. To account for this uncertain and nondeterministic aspect, we propose to use a probabilistic model checking approach. Consequently, we have selected PRISM, a probabilistic model checker for verifying systems, which can, but not necessarily, behave in a probabilistic way [5,6]. Typically, a model checker needs the system model $M$ as the input to be verified; we should then build a formal model for the considered web service. This model can be extracted from the web service code. By doing so, the verification approach will be dependant on the programming language, platform, and different programming choices. Because our objective is to have a more general approach independent from the implementation details and which can be conducted earlier in the design phase, we have chosen OWL-S for web service specification. 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 is an XML based language which defines a service as a collection of network endpoints and contains some elements such as types, messages, operations, ports, etc. [8]. In fact, WSDL describes web service interface and it doesn’t contain any information about its behavior. Thus, it cannot be used to construct the system model to be checked. On the other hand, BPEL (Business Process Execution language), which is also an XML based language, models a business process as a composition of elementary web services [8]. BPEL models a composite web service, which is composed of a set of web services, but what we are interested in is a language, which describes individual web services prior to any composition.

In our approach, the OWL-S file of each web service is used as input to our software tool, which, after parsing the file, produces a probabilistic model: Discrete Time Markov Chain (DTMC) or Markov Decision Process (MDP). Thereafter, the tool automatically creates a PRISM file for the obtained probabilistic model. The PRISM file is then used as input for the PRISM 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 transforming the web service specification into DTMC or MDP and the generation of the corresponding PRISM source code.

This paper is organized as follows. In Section 1, we present OWL-S. A brief description of the PRISM model checker is also provided. The main contribution is discussed in
Section 2 through a concrete case study. This section introduces the rules and algorithm
to transform an OWL-S model to DTMC or MDP and then to the PRISM language. The
soundness and completeness of the transformation method are analyzed as well. In Sec-
tion 3, we discuss some common properties, which are checked in our case study. Section
4 focuses on the tool implementation and algorithm. In Section 5, we review relevant
related work. Section 6 concludes the paper and identifies possible directions for future
work.

1. OWL-S and PRISM

1.1. 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 descrip-
tion 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 program-
ning 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.

1.2. PRISM: A Probabilistic Model Checker

PRISM is a tool for modeling, verifying, and analyzing systems, which can exhibit prob-
abilistic behaviors [6]. Probabilistic model checking is a formal verification technique
in 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 automati-
cally.

PRISM supports three types of probabilistic models: Discrete Time Markov Chains
(DTMC), Markov Decision Processes (MDP), and Continuous Time Markov Chains.

G. Oghihi et al. / Model Checking Single Web Services Using Markov Chains and MDFs

2. From OWL-S Model to DTMC and MDP: A Case Study

To consider the transformation of OWL-S into DTMC, let us first give the formal definition of this formalism [9].

Definition 1. A Discrete Time Markov Chain (DTMC) is a tuple $M = (S, P, t_{\text{init}}, AP, L)$ where:

- $S$ is a countable, nonempty set of states;
- $P : S \times S \rightarrow [0, 1]$ is the transition probability function such that for all states $s : \sum_{s' \in S} P(s, s') = 1$;
- $t_{\text{init}} : S \rightarrow [0, 1]$ is the initial distribution, such that $\sum_{s \in S} t_{\text{init}}(s) = 1$;
- $AP$ is a set of atomic propositions and $L : S \rightarrow 2^{AP}$ 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 $t_{\text{init}}(s) > 0$ is considered as initial state. In this section, we describe how we extract this needed information out of the input OWL-S file and produce a DTMC. However, some OWL-S control constructs need more than DTMC because transitions are labeled with actions, which are not supported by DTMC. To account for labeled probabilistic transitions, we propose to use MDP [9]. The formal definition of a DTMC is as follows.

Definition 2. A Markov Decision Process (MDP) is a tuple $M = (S, Act, P, t_{\text{init}}, AP, L)$ where:

- $S$ is a countable, nonempty set of states;
- $Act$ is a set of actions;
- $P : S \times Act \times S \rightarrow [0, 1]$ is the transition probability function such that for all states $s \in S$ and actions $\alpha \in Act : \sum_{s' \in S} P(s, \alpha, s') = 1$;
- $t_{\text{init}} : S \rightarrow [0, 1]$ is the initial distribution, such that $\sum_{s \in S} t_{\text{init}}(s) = 1$;
- $AP$ is a set of atomic propositions and $L : S \rightarrow 2^{AP}$ 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 use a concrete case study: a web service for an online sale system [10]. Since OWL-S is a new topic and there are very few OWL-S examples for small case studies, we have started from the activity diagram of our case study from which we created the corresponding OWL-S specification. Then we transform this specification into the internal representation of our tool. The tool parses and produces an associated DTMC/MDP.

The web service is an online system from which clients can order and buy products. The system checks the client’s credit and the stock amount, if the client has enough credit and the required product is available in the stock, it passes the order to the warehouse, otherwise it contacts the customer to cancel the order or set a new delivery time. Figure 1 shows the activity diagram for this web service.

We illustrate the OWL-S structure by the following BNF grammar where atomic processes are enclosed in “ ”, composite processes are enclosed in <> and the names of control constructs are enclosed in [ ]:

1. <OrderingProcess> ::= “PlaceOrder”[Seq] <Ordering>
2. <Ordering> ::= “CancelOrder”[Choice] <OrderCreditChecking>
3. <OrderCreditChecking> ::= “ProcessOrder”[Seq] <CreditChecking>
4. <CreditChecking> ::= [IF - Then] <StockChecking> [Else]
7. <NewDeliveryTime> ::= “ChangeDeliveryTime”[Choice]
8. <WarehouseProcess> ::= “OrderToWarehouse”[Seq]
9. <ShippingReqProcess> ::= “CheckNewDeliveryTime”[Seq]
13. <ShippingReq> ::= “RequestShipping”[Seq] “OrderConfirmed”
15. <DataCorrection> ::= [IF - Then] “RaiseSendInvoice” [Else]
16. <RaiseSendInvoice> ::= “RaiseInvoice”[Seq] “SendInvoice”
17. <RetData> ::= “RetData”[Seq] “RaiseSendInvoice”

This OWL-S form includes control constructs: Sequence (Seq), Choice, IF-Then-Else and Split-Join. 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.
2.1. Modeling Atomic Processes

Since each atomic process represents a programming unit, which receives some inputs and produces some outputs, this process can be transformed into a DTMC as a state, which will be reached based on the control structure of the program. In our example, we consider each atomic process as a single state. Thus, PlaceOrder, CancelOrder, OrderToWarehouse, etc. are DTMC states.
2.2. 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, ShippingReq is a sequence composite process, which is made of two atomic processes: RequestShipping and OrderConfirmed. 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.

Finally, 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 4.

2.3. Modeling Control Constructs

2.3.1. Modeling Sequences

A sequence is 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, RaiseSendInvoice, a composite sequence process, is shown by a DTMC having two states named RaiseInvoice and SendInvoice and a transition between them. Since there is only one possible transition from RaiseInvoice, the probability of this transition is obviously 1. The corresponding DTMC is shown in Figure 2.

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

```plaintext
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.
2.3.2. Modeling Choices

In this construct, the caller process has different processes as options to select from, but only one is to be selected. Thus, we substitute a composite choice construct with its inner processes.

In [2] we simply supposed that the processes are equally chosen and thus the probability of reaching each state is $\frac{1}{Np}$, where $Np$ is the number of available processes. However, the probability of selecting each process is better determined as a parameter in the OWL-S structure according to the web service business logic, so that we can extract it. In our example, PlaceOrder, which is a choice composite process is replaced by an atomic process named $Ci$ ($i$ shows the number of choice control construct) which we add to the structure and this atomic process goes to CancelOrder and OrderCreditChecking. Thus, PlaceOrder will be transited to the $Ci$ atomic process, which in turns transit to CancelOrder and OrderCreditChecking, which are substitutions for the PlaceOrder state. The associated probabilities are $\frac{1}{4}$ and $\frac{3}{4}$ respectively (see Figure 3). It should be mentioned that since OrderCreditChecking is a composite process, it will be decomposed and substituted with other processes.

2.3.3. Modeling 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; 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, named Fi which i is the number of If-Then-Else control constructs 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 4. The probabilities of these transitions which should be extracted from the OWL-S file are $\frac{2}{3}$ and $\frac{1}{3}$. 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:

```
ax: [0..2] init 0;
```

---

http://site.ebrary.com/lib/uow/Doc?id=10524253&ppg=41
Copyright © 2011. IOS Press. All rights reserved.
May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.
2.3.4. Modeling Repeat-While and 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 transitioned to the first process of while-body, which itself is a state. Otherwise, the state is transitioned 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 W1 and the while-condition as Con. The while body consists of processes named: s1, s2, and s3, and the next process after the loop body is s4. Again, we suppose that all the processes are atomic. The resulting DTMC is illustrated in Figure 5.

![Figure 5. Modeling Repeat-While into Markov Chain](image)

Associated with repeat-while is the following PRISM code:

```
s: [0..4] init 0;
[] Con & s = 0 → (s' = 1);
[] s = 0 → 2/3; (s' = 1) + 1/3; (s' = 2);
```

It means since we have 3 states F1, (1, s1), we define a variable named s with 3 different values. Value 0 represents state F1, values 1 and 2 represent states t1 and s1. s = 0 shows we are in state F1, and s = 1 means we are in state t1 and s = 2 means we are in state s2. Furthermore, since the rest of transitions are of type sequence, the equivalent code could be written as explained earlier.
G. Oghalti et al. / Model Checking Single Web Services Using Markov Chains and MDFs

10 of 18 31/01/2014 12:23 PM

\[
\begin{align*}
\text{\texttt{1}} & \text{ !Con \& s=0 \rightarrow (s'=1);} \\
\text{\texttt{1}} & \text{ s=0 \rightarrow 3/4:(s=1) + 1/4:(s=4);} \\
\end{align*}
\]

A similar algorithm can be applied to the repeat-until control. The associated DTMC is shown in Figure 6, and its corresponding PRISM code is as follows:

\[
\begin{align*}
\text{s: } & \text{[0..4] init 0;} \\
\text{[1] & Con \& s=3 \rightarrow (s'=4);} \\
\text{[1] !Con \& s=3 \rightarrow (s'=0);} \\
\text{[1] s=3 \rightarrow 2/5:(s=4) + 3/5:(s=0);} \\
\end{align*}
\]

2.3.5. Modeling Splits and Split-Joins

Split shows processes, which are executed concurrently, and split-join shows processes, which are executed concurrently, but in addition, when all of them are executed, another process follows. It means, all the processes join a common process after execution. Logically, when processes are parallel, the probability of the execution for each one of them is 1, which can be modeled, as argued earlier, using MDP rather than DTMC. Let us consider a prototypical example: Suppose we have an atomic process named (0), which is in sequence with a composite process of split type and this composite process is composed in turn of three different composite processes named \( p, q, \) and \( r. \) Suppose \( p, q, \) and \( r \) are sequence processes and are respectively composed of \( p_0, p_1, q_0, q_1, \) and \( r_0, r_1. \) The corresponding MDP is shown in Figure 7, where the probability of each parallel transition is 1. The transitions are also labeled using fictive actions.

Using the aforementioned transformation technique, the resulting PRISM code is as follows:

\[
\begin{align*}
\text{s: } & \text{[0..?] init 0;} \\
\text{[Actt] s=0 \rightarrow (s'=1);} \\
\text{[Acts1] s=1 \rightarrow (s'=2);} \\
\text{[Acts2] s=1 \rightarrow (s'=3);} \\
\text{[Acts3] s=1 \rightarrow (s'=4);} \\
\text{[Actp] s=1 \rightarrow (s'=5);} \\
\text{[Actq] s=2 \rightarrow (s'=6);} \\
\text{[Actr] s=3 \rightarrow (s'=7);} \\
\end{align*}
\]

Copyright © 2011. IOS Press. All rights reserved.
May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.
As we can see in this piece of code, each statement has a label. Since we model split and split-join with MDP, each transition from one state to another state has a name, which is represented by a label in the corresponding PRISM statement. The possibilities of each transitions are 1, it means all transitions will be occurred and this guarantees the synchronization of all branches of the split statement.

A similar algorithm is used for the transformation of the split-join control, except that all split processes will join to reach a given process. Figure 8 shows the MDP associated with the above prototypical example after adding a new atomic joining process (1).

**Theorem 1.** The transformation technique from OWL-S to DTMC and MDP described above is sound and complete.

**Proof.** On one hand, the soundness of the technique can be proved by induction on the control construct types. The algorithm works recursively by considering all these types individually, and then combining the resulting DTMC and MDP. Since MDP is a generalization of DTMC, where the actions can simply be replaced by null, the final diagram is an MDP. As the technique parses the OWL-S sequentially, all the individual MDPs obtained share one or more states. Consequently, their combination is simply obtained by merging the similar states in one state and keeping all the incoming and outgoing transitions. It results that the final MDP is a sequential combination of all the MDPs, which guarantees its soundness. On the other hand, the completeness is straightforward from construction as all the possible control constructs are considered. Consequently, all the possible behaviors captured by OWL-S are captured as well by the resulting combined MDP.

3. Verification and Experimental Results

The verification process using the PRISM model checker needs the PRISM specification of the web service to be verified and the desired properties to check if the web service is design so that the properties are satisfied. In this section, we will specify some of these properties for our case study using PCTL logic [9]. In this logic, the operator $F$ means in the future of some computations, and the probability is expressed as follows: $P_{\Delta} x$, where $x$ is a real number, and $\Delta \in [<, >, \leq, \geq]$. For readability reasons, we define labels for each state so that we can use these labels instead of the state numbers. This, instead of using the state numbers, for instance $s = 20$, we use "ContactCustomer" as its label.

- $\neg \text{CreditCheckingOk} \Rightarrow P = 1 \ [F \text{ContactCustomer}]$
This business logic property, satisfied in our model, says: if a client has a good credit history, the probability that the system will contact him is great or equal to 1, which practically means 1.

- !$CreditCheckingOk \implies P <= 0 [FItemInvoicePacking]$

This business logic property, satisfied in our model, says: if a client does not have a good credit record, the probability that item will be packed for him is 0.

- !$StockCheckingOk \implies P >= 1 [FCheckNewDeliveryTime]$

This property says: if there is no product in the stock, the system will check if a new delivery time is possible, which is the case of our system.

- !$CreditCheckingOk \mid !StockCheckingOk \implies P <= 0 [Faccept]$  

We define the label "accept" in PRISM as follows:

Label "accept" := s = 11;  
This formula expresses the following property: if a client does not have a good credit record or there is no product in the stock, the web service will not reach an accept state.

- !$DataIsCorrect \implies P <= 0 [FSendInvoice]$

This business logic property, satisfied in our model, says: if the data is not correct for a given client, an invoice will be never sent to this client. This is an example of safety property.

- $RequestShipped \implies P >= 1 [Faccept]$  

If a given request was shipped, the probability of terminating the execution of the web service by reaching "itemInvoiceShipping" state in the future is greater or equal to 1 (which technically means equal to 1). This property is an example of liveness property of the OnlineSale system.

- $P_{min} = ? [FCancelOrder]$

This property estimates the probability that an order will be cancel either by the customer or because of a bad credit history.

- $P_{max} = ? [FOrderToWarehouse]$

This property estimates the probability that an order is passed to the warehouse.

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.

Figure 9 shows the MDP that our tool produced for the OnlineSale 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 T5850, 2.17GHz. 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 (more than $7 \times 10^8$ states and transitions can only be checked in less than 92 sec.)
4. Proof of Concepts

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 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 2.

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 10 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. It reads the elements of XML based OWL-S file and analyzes them one by one. If the element represents an atomic process, the algorithm adds the process to the list of atomic processes. If the element represents a composite process, the algorithm adds it to the list of composite processes and then analyzes all its inner processes. For the inner processes, the same procedure is repeated. If there is an atomic process, the algorithm adds it to the atomic process list, and if there is a composite process, it adds it to the composite process list. Thereafter, the algorithm adds these inner processes as children for the composite parent process. This recursive procedure is repeated for each nested composite process. By executing this algorithm, a linked list structure named Internal Representation I is created to be used by the DTMC/MDP generator.

DTMC/MDP Generator. The task of this part is generating DTMC/MDPs. It reads 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 in memory. Then, the algorithm stores the paths into the txttextInternal Representation II, a multi dimensional linked list. It reads one composite process 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 2 are followed. If the type is sequence, the algorithm puts all composed processes in equivalentList and then merges this list with resultList. For the types repeat while and repeat-until, the algorithm fol-
follows the same steps. If the type is if-then-else, the algorithm puts all processes related to if in ifList and all processes related to else in elseList. At the end, the algorithm merges these two lists into resultList. If the type is choice, it creates a multidimensional list named choiceList, and puts each inner process as a row in it to create multi paths. The algorithm merges then this list with resultList. As split and split-join have also different branches, the algorithm uses 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, the procedure traverses the list and makes 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. The pseudo-code of this algorithm is depicted in Algorithm 2.

PRISM Code Generator. This part reads the Internal Representation II and produces the respective PRISM code. The algorithm used here is the one we introduced in Section 2.

5. 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 [11][12]. Only a few initiatives about web service verification based on OWL-S have been lately launched.

In [13], 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 [14], 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 [13] and [14] 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. Another feature, which is added to this approach and does not exist in [2,12,14], is working with probabilistic OWL-S. In the current approach, we suppose that OWL-S contains a parameter named probability for each process. By doing so we can extract the probability from the OWL-S file rather than making some ad-hoc assumptions for its value. The project of extending OWL-S with probabilities has already been launched by some research groups [13].

In [16], 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 [17], 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 TCSPN (Time Constraints Petri Net) model. Our transformation rules for changing an OWL-S model into Markov chain diagram and decision process are somewhat 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 TCSPN. Furthermore, we present an algorithm and a fully implemented tool to
Algorithm 2 decomposeCompositeProcess: resultList % is a sequence of atomic states
Variables: choiceList, equivalentList, ifList, elseList, innerList

if compositeProcess.Type = "Sequence" then
    for each process in composedProcessList do
        if process.Type = "Atomic" then equivalentList.add(process)
        else resultList.add(equivalentList)
        innerList = decomposeCompositeProcess(process)
        resultList = merge (resultList, innerList)
    end for
else if compositeProcess.Type = "Choice" then
    for each process in composedProcessList do
        if process.Type = "Atomic" then equivalentList.add(process)
        else choiceList.add(equivalentList)
        innerList = decomposeCompositeProcess(process)
        resultList = merge (choiceList, innerList)
    end for
else if compositeProcess.Type = "Split" then
    for each process in composedProcessList do
        if process.Type = "Atomic" then equivalentList.add(process)
        else choiceList.add(equivalentList)
        innerList = decomposeCompositeProcess(process)
        resultList = merge (choiceList, innerList)
    end for
else if compositeProcess.Type = "Split-Join" then
    for each process in composedProcessList do
        if process.Type = "Atomic" then equivalentList.add(process)
        else choiceList.add(equivalentList)
        innerList = decomposeCompositeProcess(process)
        resultList = merge (choiceList, innerList)
    end for
else if compositeProcess.Type = "If-Then-Else" then
    for each process in ifProcessList do
        if process.Type = "Atomic" then equivalentList.add(process)
        else innerList = decomposeCompositeProcess(process)
        ifList = merge (ifList, innerList)
    end for
    for each process in elseProcessList do
        if process.Type = "Atomic" then equivalentList.add(process)
        else innerList = decomposeCompositeProcess(process)
        elseList = merge (elseList, innerList)
        resultList = add (ifList, elseList);
    end for
else if compositeProcess.Type = "Repeat-While" then
    for each process in composedProcessList do
        if process.Type = "Atomic" then equivalentList.add(process)
        else innerList = decomposeCompositeProcess(process)
        resultList = merge (resultList, innerList)
    end for
else if compositeProcess.Type = "Repeat-Until" then
    for each process in composedProcessList do
        if process.Type = "Atomic" then equivalentList.add(process)
        else innerList = decomposeCompositeProcess(process)
        resultList = merge (resultList, innerList)
    end for
end if

return resultList
perform the modeling and transformation automatically. In [18], 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.

6. 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 transforms an OWL-S file to a DTMC/MDP and creates the equivalent PRISM code. The simulation results show 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 [1], where communication protocols between the community master and slaves are characterized by their uncertainty, which fits well with our probabilistic-based approach.

On the other hand, in the current approach we use equal probabilities for each control construct since OWL-S doesn’t contain any probability. But the idea of probabilistic entropy for web services is developing. If this idea is developed in near future, we will have OWL-S files which carry an occurrence probability for each process. So we can extract the real probability for each control construct like if-Then-Else or Choice and our model will be based on real probabilities.

References


http://site.ebrary.com/lib/uow/doc?id=10524253&ppg=50
Copyright © 2011. IOS Press. All rights reserved.
May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.


