VERIFYING WEB SERVICES USING PROBABILISTIC MODEL CHECKING

Giti Oghabi

A thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science(Software Engineering) at

Concordia University

Montréal, Québec, Canada

September 2011

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Giti Oghabi**

Entitled: **Verifying Web Services using Probabilistic Model Checking**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science(Software Engineering) at

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair

Dr. Brigitte Jaumard

_____ Examiner

Dr. Benjamin Fung

_____ Examiner

Dr. Olga Ormandijeva

_____ Supervisor

Dr. Jamal Bentahar

Approved _____

Chair of Department or Graduate Program Director

_____ 20 _____ _____

Robin A.L. Drew, Ph.D.,ing., Dean

Faculty of Engineering and Computer Science

Abstract

Verifying Web Services using Probabilistic Model Checking

Giti Oghabi

In service computing, soundness and correctness are key concerns of web services technology. Verification is among the methods used for checking the design's correctness against specific desired properties. Verification means checking a web service in terms of general properties such as safety, liveliness, deadlock freedom, etc. It also means checking if the web service satisfies some specific business-logic properties. Verification aims at increasing the trustworthiness of a web service and decreases its failure rate.

In this thesis, we propose a model-checking verification approach for individual web services. We use semantic markup for web services (OWL-S), an Ontology Language for Web Services, to describe web services' behaviors. We introduce probabilities as parameters for this language to model uncertain choices. We use PRISM, a probabilistic model checker to verify web services against general and business-logic properties expressed in a propositional probabilistic logic. In this approach, we develop a transformation algorithm for converting the OWL-S to Markov chain diagram and Markov decision process, which are formal probabilistic models that are compatible with PRISM. The obtained diagram is then automatically 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. We implemented this transformation algorithm and these procedures in a software tool.

Acknowledgments

The fulfillment of this work would not have been possible without the support of many people.

I would like to express my sincerest gratitude to my supervisor, Dr. Jamal Bentahar for providing me the opportunity to work in his research team as well as for his kind and endless support, invaluable guidance and precious advice.

I would like to thank all my dear lab mates for their help, useful suggestions, sharing experience and the friendly environment they provided in the workplace.

My warm appreciation goes to my dear sisters, aunt, cousin and friends, Parvaneh and Javad who always were beside me, for their kindness, support and encouragement enlightening my heart.

Last, but not least, special and deepest thanks and love to my dear grandma and dear parents for their unconditional support, continual encouragement throughout my academic and whole life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we introduce our research topic and discuss the motivations behind it. We also present the research questions this thesis aims to answer and outline our contributions. After a discussion of relevant related work, we present the thesis structure.

1.1 Context of Research

One of the fast growing and pervasive technologies in IT industry is the service-oriented computing paradigm. This paradigm suggests designing software as services with a standard interface so that they can be used by and interact with other applications. There are wide ranges of services, from simple ones, which can perform single tasks to more complicated ones, which execute complex business processes. Services that use the Internet as their infrastructure are called web services.

"A web service is a self-describing, self-contained software module available via a network, such as Internet, which completes tasks, solves problems, or conducts transactions on behalf of a user or application" [24]. The reason d'etre of web services is to interact with other services and applications, which explains the emergence of more advanced technologies such as web service composition and communities of web services. Web service composition refers to the combination of many complementary web services to provide a more complete service that no single web service can provide.

Communities are virtual structures aiming to host web services having similar functionalities so it will be easy to localize (i.e. discover) them and they can collaborate with each other, for example, through substitution [5, 6].

Similar to any other software system, soundness and correctness are key concerns of web services technology. Verification is among the methods used for checking the system's correctness against specific desired properties. In fact, verification means checking a web service in terms of safety, liveliness, deadlock freedom, etc. It also means checking if the web service satisfies some specific business-logic properties.

One of the verification methods, which we have used in this research, is model checking. It is a formal, algorithmic, and fully automatic technique that aims to check whether or not a system model M satisfies a given property ϕ (i.e. $M \models \phi$).

In this research, we introduce a model checking-based approach for web service verification and its implementation. Among different model checkers we have chosen PRISM, which is a probabilistic model checking, as it allows verifying both probabilistic and non-probabilistic properties.

1.2 Motivations and Research Questions

The first motivation, which initiates our interest in verifying web services, was having a mechanism in a community setting to asses web services before they get granted acceptance to join a community. The objective behind such a mechanism is to know the percentage of satisfaction of the community's requirements, so the acceptance decision will be made by the community's master (i.e. the manager) based on this value with regard to an established threshold. Automatic web service verification provides a solution towards this objective.

Automatic verification is also extremely useful for web service composition as different web services are involved and the malfunctioning of one of them can lead to the whole system failure. However, before verifying the whole composition, each single web service should be verified independently from the others. Furthermore, before launching a new web service, its provider should be

sure that it satisfies the requirements, for instance deadlock freedom, liveness, safety, etc.

The main research question this thesis aims to answer is how to automatically verify single web service, not only when the design choices are deterministic, but also when some probabilistic choices could be included. In fact, for the motivations listed earlier and particularly for community and composition management, the question is how to verify a web service without having its source code. Which formalism should be used to model web services is another question that should be answered before starting the automatic verification.

1.3 Contributions

In this thesis, we introduce a framework for automatic verification of web services based on their behaviors. As we use model checking as verification technique and each model checker needs the system model M as input to be verified, we provide a technique to build such 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 Ontology Languages for Web Services (OWL-S) for web service description. We will describe OWL-S ontology in more detail in Chapter 2.

In our approach, the OWL-S file of each web service is used as input to a software tool we have developed, which, after parsing the file, produces a probabilistic model (Markov chain diagram). Thereafter, the tool automatically creates a PRISM file for the created 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 main contributions of this thesis have been published in two papers: [22, 23]

1.4 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 the verification of web service composition and are consequently based on BPEL4WS [14, 12]. Only a few initiatives about single web service verification based on OWL-S have been lately launched.

In [19], 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. However, unlike Lomuscio et al.'s proposal, we implemented not only the "sequence" control, but also the other control constructs including "choice", "if-then-else", "repeat-while", etc. In [4], Ankolekar et al. have also investigated automatic verification of web services. However, their main focus is not on a single web service, but rather on the interaction protocols of these web services.

Unlike [19] and [4] 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 [16, 13]. As argued in [7] and [25], 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 occur definitely and some of them may occur with certain probability. For example in an "$If - Then - Else$" construct, the probability for If branch to occur may not be equal to the $Else$ branch. In a deterministic model checker, these type of considerations are not supported, 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.

Choosing a probabilistic model checker also 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 the PRISM model checker, there is a feature for

determining the probability of each state occurrence, so we can show not only if a given state satisfies a given property, but also the probability of such a satisfaction. Having these probabilities in the model, PRISM can calculate and predict the probability of reaching a specific state (i.e. 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 certain probability value.

In [8], 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 [18], 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 from OWL-S model into Markov chain diagram are somehow similar to these rules, but the resulting models are totally different. Markov chain diagrams 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 [21], 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.

1.5 Thesis Overview

The rest of the thesis is organized as follows: In Chapter 2, we present a brief review of all background subjects, which we need to understand the thesis approach, including: service oriented paradigm and web services technology, ontology, Ontology Languages for Web Services (OWL-S), verification and model checking, and finally the PRISM model checker. Chapter 3 discusses our approach in

detail, provides two case studies that have been examined by our tool along with their experimental results and then explains the tool architecture and implementation. In Chapter 4, we conclude our work and describe opportunities for future work.

Chapter 2

Background

Before introducing our verification framework, we will present in this chapter some preliminaries, which are relevant for the rest of the thesis. This chapter is organized as follows. In Section 2.1, we discuss about web services, the difference between web services and web applications, service oriented architecture and web service stack. In Section 2.2, we explain the concepts of ontology and Ontology Web Language (OWL). Then, we discuss the need for ontology for web services. Thereafter, we describe Web Service Ontology Language (OWL-S) and its different parts. In Section 2.3, we describe the verification and model checking problem and technique and present three different logics: PLTL, CTL and PCTL. Finally, in Section 2.4, we introduce PRISM, which is a probabilistic model checker and describe the PRISM model language and PRISM property specification language.

2.1 Web Services

Service oriented computing is a paradigm, which uses different services to develop distributed applications rapidly. Services are self-contained modules that can be described, published and located over a network. They are programmed using XML-based technology. A service can be described and used through a well defined interface. Services provide different types of functionalities from simple requests such as converting a file format to another format, to very complicated business processes.

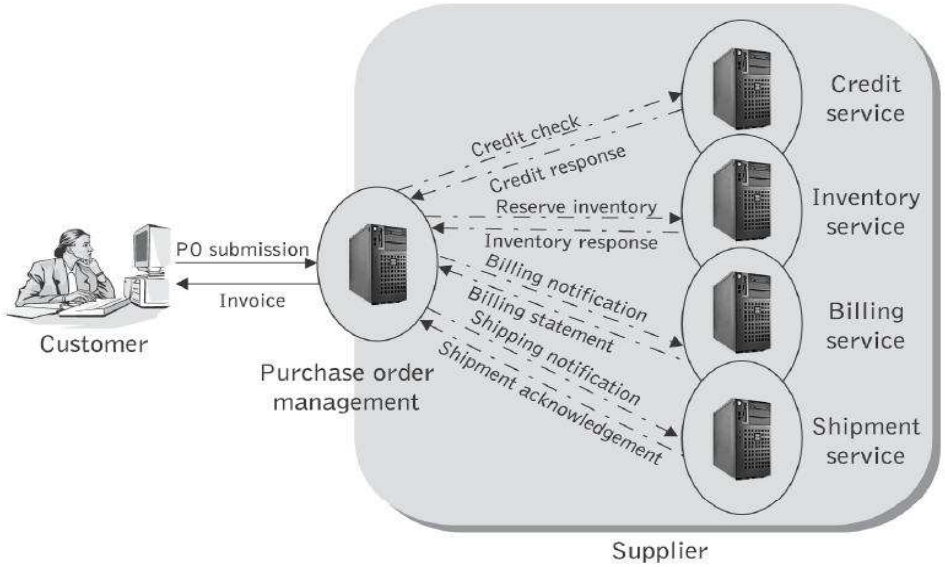

Figure 2.1: A purchase application involving interacting web services

They are independent of the context in which they are used [24].

2.1.1 What is a Web Service?

"A web service is a self-describing, self-contained software module available via a network, such as the Internet, which completes tasks, solves problems, or conducts transactions on behalf of a user or applications" [24]. We can have different forms of web services: simple business web services such as funds withdrawal or funds deposit services, full business services such as automated purchasing services, full applications such as insurance applications or service-enabled resources such as services for access to particular databases.

In Figure 2.1 [24], a purchase order application is shown. In this application, different web services interact with each other and finally a purchase is accomplished. First, a purchase order sent by a customer initiates the application. The tasks of credit checking, stock checking and calculating the bills are done concurrently. Then, some other tasks are performed sequentially, for instance, the item will be shipped after the billing is done. This example shows that some services collaborate with each other to perform a single task through the Internet.

2.1.2 Web Services versus Web-based Application

It is important to mention that web services are different from web-based applications. The main difference is that web-based applications are developed to be used by human users but web services are designed and developed for access by humans as well as automated applications. The main characteristics of web services, which make them different from web-based applications are summarized as follows [24]:

- Web Services act as resources for other applications.
- Web services are modular and self-describing. They can describe their functions and inputs for their users.
- Web services are more manageable than web-based applications. It means the states of a web service can be monitored and managed by using external application management.
- Web services may be brokered. If some web services perform the same task, then some applications may place bids for the opportunity to use the requested service.

2.1.3 Service Oriented Architecture

Service Oriented Architecture (SOA) presents a set of design principles providing services to software applications or other services through a network by publishing interfaces. The main goal of SOA is increasing the interoperability among existing services and technologies. In SOA, systems are mostly modular and flexible rather than monolithic and static. The concept of SOA is strongly related to web services, but SOA and web services are two different subjects. SOA can be implemented without web services but its deployment is much easier by web services [24].

SOA consists of three main building blocks: service provider, service registry and service requester (client). Providers are software agents that provide the services. Providers publish service description on the service registry. Service requesters are the software agents that request the execution of a service. Clients should be able to find the required service description in the service registry.

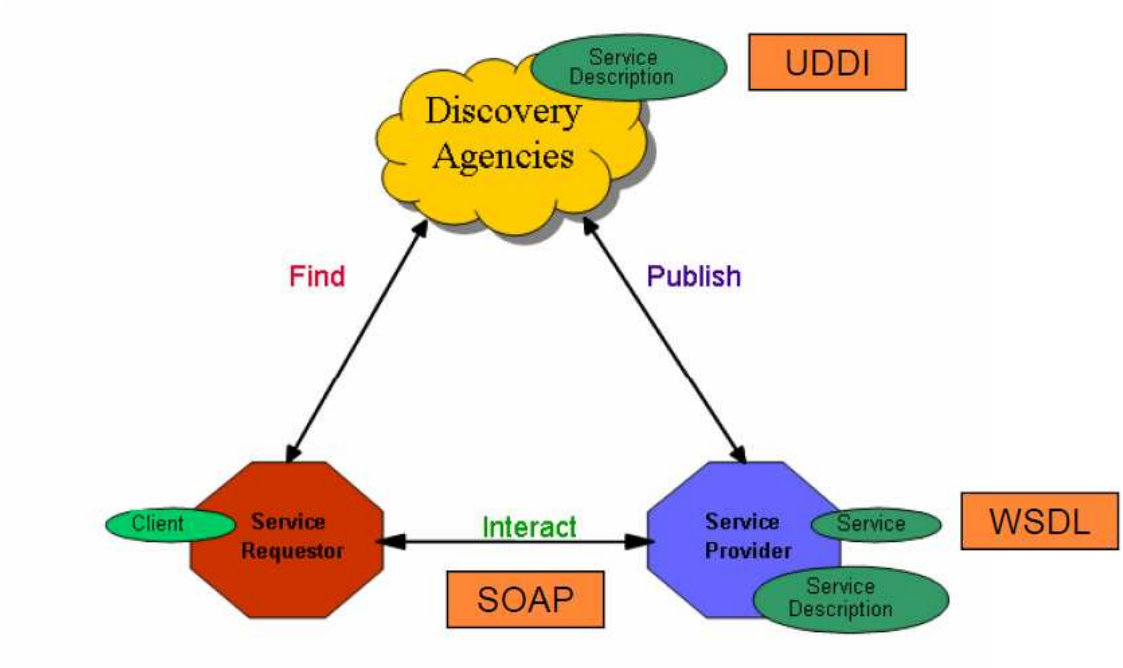

Figure 2.2: Service oriented architecture

Figure 2.2 shows the service oriented architecture and its main parts. If implemented using web services, SOA will consist of:

- Web Service Provider: A web service provider is the organization that the web service belongs to and it specifies and implements the business logic related to the service. Web service provider has to publish the web service in a service registry and a service discovery agency hosts the service registry.

- Web Service Requester: A web service requestor is the enterprise that looks for some services. It looks for and invokes the service. The web service requester searches the service registry and if it finds the service, it will bind to it.

- Web Service Registry: Web service registry is a searchable directory where the web service description is published. Service requester searches and finds the service description in the web service registry and obtains the binding information for the service.

2.1.4 Web Service Technology Stack

Figure 2.3 depicts the web service stack layers, which consist of transport infrastructure, messaging, description, discovery and processing technologies. Regarding the web services infrastructure, since these services mostly work on the Internet, they take advantage of HTTP and perhaps FTP and SMTP as transport standards. Another technology, which web services use, is *Extensible Markup Language* (*XML*). XML is a widely used format for exchanging data and the related semantics. Particularly, *Simple Object Access Protocol* (*SOAP*) is an XML-based messaging protocol on which web services rely to exchange information. Web services use XML in every layer in the web service stack, except the first one (i.e. transport layer).

WSDL is another layer in the web service stack, which is responsible for service description. *WSDL* is a *Web Service Description Language*, which describes the interface exposed by a web service. Through this service description, a provider can describe the offered service and a requester can invoke it.

Universal Description, Discovery, and Integration (*UDDI*) is another web service layer. *UDDI discovery* is the process of locating the web service definition. Through this definition, a client learns that a particular service exists, what its capabilities are, and how to interact with it.

The last layer of web service stack uses different processing technologies, but we only focus on *BPEL*, which is more relevant to the subject of this thesis. *Business Process Execution Language* (*BPEL*) is a language used for *Service Composition*. BPEL describes the execution logic of a web service by defining the control flows. In this way, a service which is a composition of other services can be described [24].

2.2 Web Service Ontology Language (OWL-S)

Since our research is based on OWL-S, which is an ontology language for web services, we present in this section this language along with related concepts ontology and Web Ontology Language (OWL).

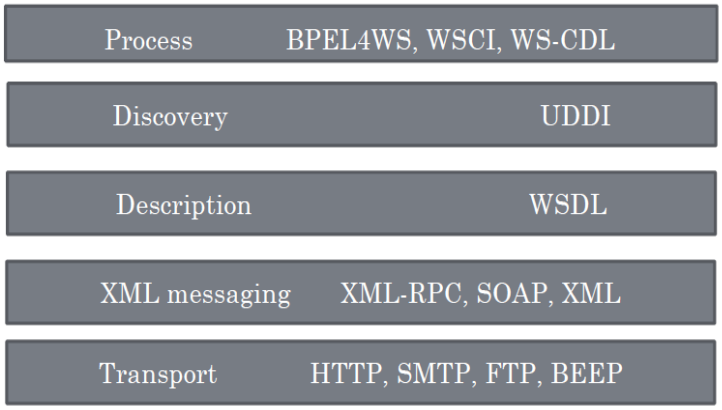

Figure 2.3: Web service stack

2.2.1 Ontology and Web Ontology Language (OWL)

The term ontology is originally borrowed from philosophy and nowadays, it is widely used in different computer areas such as artificial intelligence and semantic web. In fact, the term ontology refers to a document that defines the relations among other terms [26].

"The web Ontology Language (OWL) is a semantic markup language for publishing and sharing ontologies on the World Wide Web" [2]. OWL is an extension for RDF (the Resource Description Framework) and a developed form of DAML+OIL web ontology language. OWL is designed to express a wide variety of knowledge and provide a means to reason with it in order to express more complex knowledge using logical inferences [3]. It can be used to describe classes and relations between classes and establish a domain by defining those classes and their properties, as well as individuals and asserting properties about them. The reasoning about these classes can be conducted through semantics and inference rules [27].

2.2.2 The Need for Ontology for Web Services

To make use of a web service, a software agent has to know the description of the service and the means by which it can be accessed. OWL-S is an extended markup language designed to provide such a description. By having this description besides a web service, users or software agents are

able to discover, invoke, compose and monitor particular web services, and they are able to do that with a high degree of automation. In fact, the ontology structuring mechanisms of OWL provide an appropriate, web-compatible representation language and framework for web service description [1].

In the following, we briefly discuss some motivations for using Ontology for web services:

Automatic Web Service Discovery:

It can help a client to find the location of desired web services, which have particular capabilities. For example, suppose a user wants to find an airline ticket seller service selling tickets for two cities and accepting a particular credit card. Normally, this process can be done by human interaction. The user uses a search engine, read the results, and filter them manually. We can do all these activities by help of OWL-S, it means OWL-S gives some information about web service responsibility and function and it can be located automatically.

Automatic Web Service Invocation:

It means an agent or software programmer can invoke the web service automatically. It is usually a collection of remote procedure calls. OWL-S provides a declarative API that specifies the required arguments for each call. An agent should be able to interpret this declarative markup to know the inputs for invoking the service.

Automatic Web Service Composition and Inter-operation:

It involves the automatic selection and composition of web services for performing a complex task. For example, if the user wants to go to a trip for a conference, the user must select the relevant web services (for example conference registration, hotel booking, ticket booking, etc.) and specify the composition manually. With OWL-S, the information required to select and compose services can be declaratively specified and a software agent can read and use these information for automatic service compositions.

In the next section, we describe the overall structure of OWL-S and its different parts.

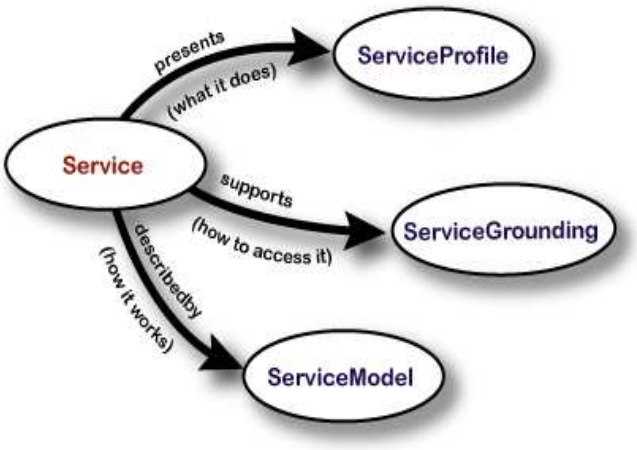

Figure 2.4: Top level of the service ontology

2.2.3 Main Parts of OWL-S

An Otology for web services has three main parts: the service profile, model (or process) and grounding [1]. Figure 2.4 depicts these three parts of OWL-S. In the following, we describe each of these three parts:

- Service Profile: A service profile says "what the service does". It has useful information for the agent, which is seeking a service. These information are useful to advertise the service.
- Service Model: A service model or process says "how the service is used". It gives a detailed perspective on how to interact with a service.
- Service Grounding: A service grounding says "how the access to the service should be performed at the technical level". A grounding provides the information about transport protocols.

In this thesis, we will focus on "Service Process" and the way a service will interact with a user or another service or application.

2.2.4 Service Process

To give a detailed perspective on how to interact with a service, such a service can be viewed as a process. In order to do that, the service process component of OWL-S models the way the service

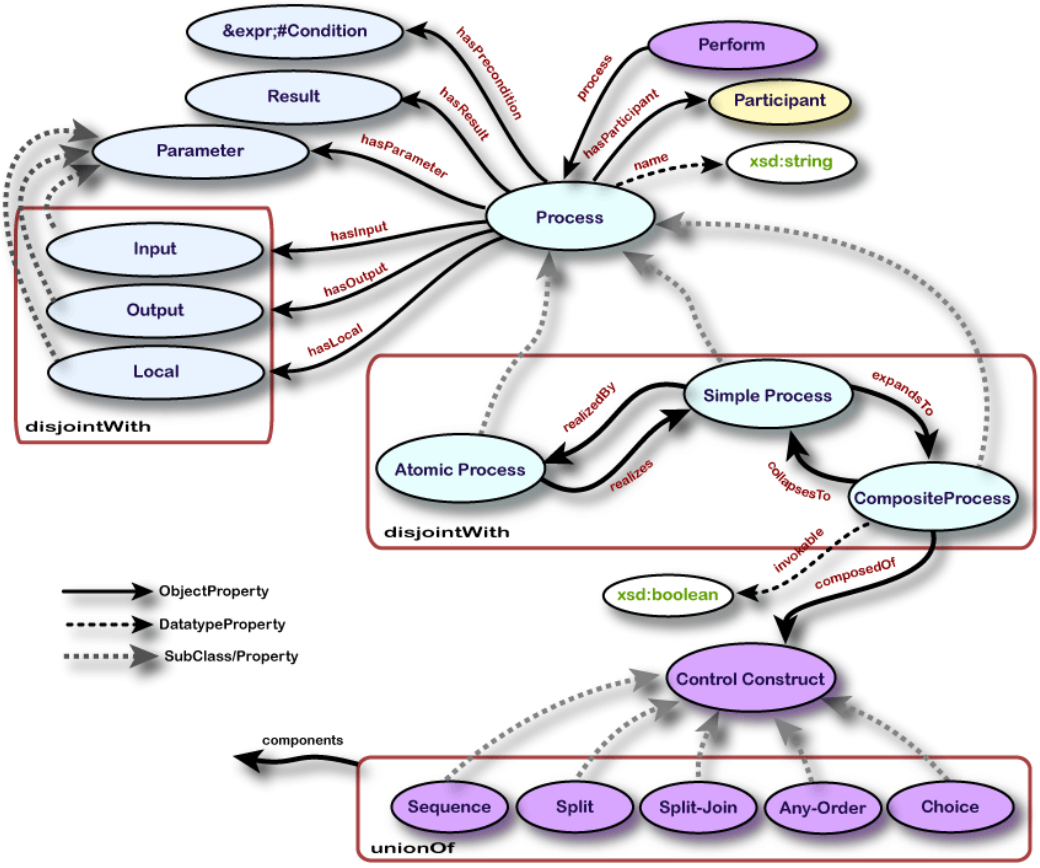

Figure 2.5: Top level of the process ontology

behaves and interacts with users or other services and applications.

There are two different processes: *atomic* and *composite* processes. An *atomic* process is the description of a service, which receives one message and sends one message in response. A composite process is a service, which has some states and each message is forwarded to a specific state. A process can have any number of inputs and outputs. It also can have some preconditions, which hold until the process is invoked.

In the following, we will discuss the different parts of the process model as shown in Figure 2.5.

Parameters:

A parameter is a class in OWL-S file, which contains some subclasses: "Input","Output" and "Locals". The inputs and outputs variables are located in "Input" and "Output" subclasses and in

"Locals" subclass, we have local variables for the process. A very simple example of inputs and outputs is shown as follows where an atomic process named "Purchase", with three input variables: "ObjectPurchased", "PurchaseAmt", and "CreditCard" and one output named: "Confirmation-Num" are described:

```
- <process:AtomicProcess rdf:ID="Purchase"/>
-    <process:hasInput>
-       <process:Input rdf:ID="ObjectPurchased"/>
-    </process:hasInput>
-    <process:hasInput>
-       <process:Input rdf:ID="PurchaseAmt"/>
-    </process:hasInput>
-    <process:hasInput>
-       <process:Input rdf:ID="CreditCard"/>
-    </process:hasInput>
-    <process:hasOutput>
-       <process:Output rdf:ID="ConfirmationNum"/>
-    </process:hasOutput>
- </process:AtomicProcess>
```

Control Constructs:

To make a composite process, there are different control constructs, for example: *Sequence*, *If-Then-Else*, *Repeat-While*, *Repeat-Until*, *Split*, and *Split-Join*. A composite process has a *composedOf* property, which indicates the control construct of composition. In the following, we show this structure:

```
- <owl:ObjectProperty rdf:ID="composedOf">
-     <rdfs:domain rdf:resource="#CompositeProcess"/>
-     <rdfs:range rdf:resource="#ControlConstruct"/>
```

```
- </owl:ObjectProperty>
```

Then, the control construct itself should be described. Each control construct is composed of "control construct list" and each "control construct list" has a *first* part, which shows the first process that should be executed and a *rest* part, which is a control construct list itself. The following shows the control construct for Sequence

```
- <owl:Class rdf:ID="Sequence">
-   <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
-   <rdfs:subClassOf>
-     <owl:Restriction>
-       <owl:onProperty rdf:resource="#components"/>
-       <owl:allValuesFrom rdf:resource="#ControlConstructList"/>
-     </owl:Restriction>
-   </rdfs:subClassOf>
- </owl:Class>

- <owl:Class rdf:ID="ControlConstructList">
- <rdfs:comment> A list of control constructs </rdfs:comments>
-   <rdfs:subClassOf rdf:resource="&shadow-rdf;#List"/>
-   <rdfs:subClassOf>
-     <owl:Restriction>
-       <owl:onProperty rdf:resource="&shadow-rdf;#first"/>
-       <owl:allValuesFrom rdf:resource="#ControlConstruct"/>
-     </owl:Restriction>
-   </rdfs:subClassOf>
-   <rdfs:subClassOf>
```

```
-    <owl:Restriction>
-         <owl:onProperty rdf:resource="&shadow-rdf;#rest"/>
-       <owl:allValuesFrom rdf:resource="#ControlConstructList"/>
-    </owl:Restriction>
-    </rdfs:subClassOf>
- </owl:Class>
```

2.3 Verification and Model Checking

After discussing web services, web service ontology and OWL-S, in this section, we will present the verification problem and model checking technique.

2.3.1 System Verification

System verification is used to make sure that the related design or product possesses specific properties (see Figure 2.6). The properties can be elementary properties such as deadlock freedom or obtained from the system specification. If the system does not satisfy one of the required properties, it is a bug or defect. If the system fulfills all required properties, it is considered as a correct system with regard to the checked properties. There are many verification techniques such as peer reviewing, testing, and model checking. This thesis focuses on model checking, a formal and automatic verification method [10, 9].

2.3.2 Model Checking

Model checking is a form of verification, which uses formal methods. Using formal methods consists in applying logic and mathematical models and techniques for analyzing and modeling software, hardware and communication systems. The goal of using formal methods is to verify the system correctness against desired properties in a systematic and rigorous way.

There are model-based verification techniques, which use formal methods. In these techniques,

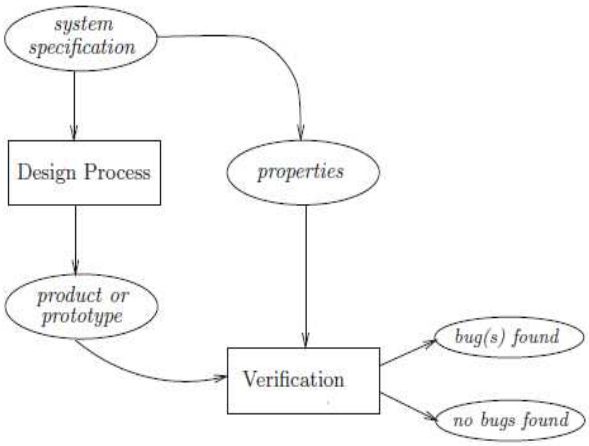

Figure 2.6: System verification schema

the system is modeled in a mathematically precise manner and the verification can explore ambiguities or inconsistencies in the system specification. Model checking is one of these techniques. Model checking explores all possible system states and examines all possible system scenarios. In this way, it can be checked whether a given model satisfies a given property or not [10, 9].

Figure 2.7 depicts the model checking procedure. As shown in this picture, the system should be modeled and the required properties should be formalized into property specification. The system model and properties specification are used as inputs to the model checker. Then, the verification shows the properties that are satisfied and counter examples for those that are not satisfied. To put the idea into a nutshell, model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds in that model [9].

Different model checking techniques and model checkers use different temporal logics for modeling the system and specifying the properties. In the following sections we briefly discuss about some of the most common logics, which we use in our approach, but before that we define the term logic and some related terms:

- **Logic:** Logic is a language for representing knowledge and information so that conclusions can be driven. Each logic has a syntax and semantics.

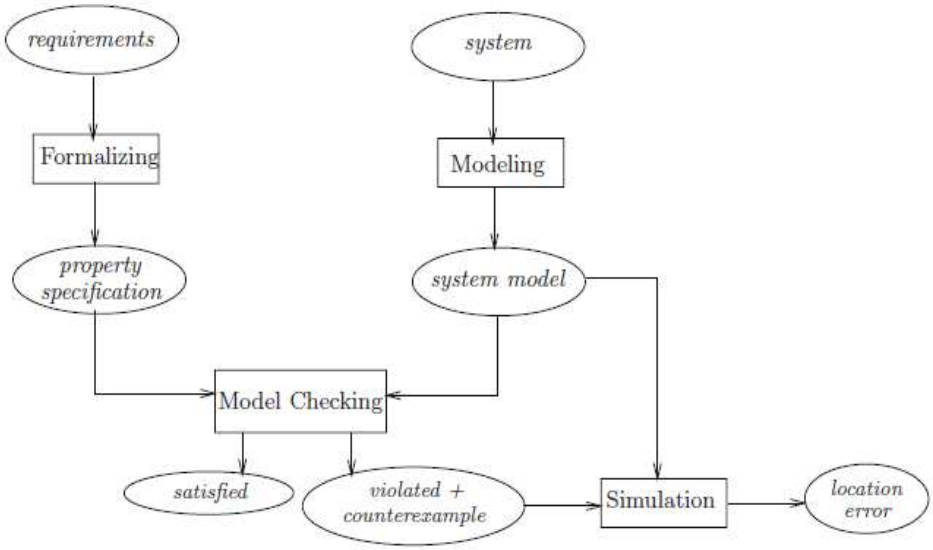

Figure 2.7: Schematic of model checking

- **Syntax:** It defines how a sentence should be written in a language.
- **Semantic:** It defines the meaning of a sentence in a language.

2.3.3 Propositional Linear Temporal Logic (PLTL)

"Temporal logic provides a very intuitive but mathematically precise notation for expressing properties about the relation between the state labels in executions" [9]. There are two forms of temporal logic: *linear* and *branching*[10]. In the linear form, at a given moment there is just a single next moment, but in the branching form, there is a branching, tree-like structure, so any given moment may have many next moments. The syntax and semantic rules of Propositional Linear Temporal Logic (PLTL) are given in the follow subsections.

PLTL Syntax

- **Elements:** PLTL has the following elements:
  - A set of atomic propositions: AP ($p \in AP$)

  - Boolean operators: $\vee$, $\neg$
  - Additional operators: $\wedge$, $\Rightarrow$, $\Longleftrightarrow$
  - Two constants: True and False
  - Temporal operators X (next), U (until)

- **Syntax:** The Syntax of PLTL is given by the following BNF grammar:

  $\phi ::= p \mid \neg\ \phi \mid \phi \vee \phi \mid \mathrm{X}\phi \mid \phi\ \mathrm{U}\ \phi$

- **Temporal Operators:** The followings are the temporal operators of PLTL:

  Xp: in the next state p is true

  Gp: Always p

  Fp: Eventually p

  pUq: p Until q

PLTL Semantics

Let σ be a path in the model to be verified, which means an infinite sequence of states related by transitions, $Label : S \to 2^{AP}$ an interpretation function, which assigns a set of atomic propositions to each state $s \in S$, I the set of initial states, p an atomic proposition, s a state, and ϕ, ψ formulas in PLTL. Then the semantics is given recursively over paths where $\sigma \models \phi$ means the path σ satisfies the formula ϕ. The set of all paths emanating from a given state s is denoted by $Path(s)$ and a path starting at a given state s_i is denoted by σ_i. When a state s (resp. a model M) satisfies a formula ϕ, we write $s \models \phi$ (resp. $M \models \phi$). The semantic is given by the following rules:

- $\sigma \models p$ iff $p \in Label(\sigma[0])$ where $\sigma[0]$ denotes the first state of the path σ
- $\sigma \models \neg\ \phi$ iff not($\sigma \models \phi$)
- $\sigma \models \phi \vee \psi$ iff ($\sigma \models \phi$) or ($\sigma \models \psi$)
- $\sigma \models \mathrm{X}\phi$ iff $\sigma_1 \models \phi$

- $\sigma \models \phi$ U ψ iff $\exists\ j \geq 0$ s.t. $(\sigma_j \models \psi$ and $\forall\ k,\ 0 \leq k < j\ (\sigma_k \models \phi))$
- $s \models \phi$ iff $\forall\ \sigma \in Path(s).\ \sigma \models \phi$
- $M \models \phi$ iff $\forall s \in I, \forall\ \sigma \in Path(s).\ \sigma \models \phi$

2.3.4 Computation Tree Logic (CTL)

CTL is a branching temporal logic for specifying system properties. In this section, we describe the syntax and semantics of this logic.

CTL Syntax

CTL is defined by the following syntactic rules:

- Propositional atoms are state formulas, that is to say formulas evaluated on the model's states.
- ϕ, ψ are state formulas $\Rightarrow \neg\ \phi$, $\phi \vee \psi$ are state formulas.
- ϕ is a path formula $\Rightarrow$ Eϕ, Aϕ are state formulas, where path formulas are evaluated through paths, infinite sequences of states, and E and A are existential and universal path quantifiers.
- ϕ, ψ are state formulas $\Rightarrow$ Xϕ ,ϕ U ψ are path formulas

Other operators are introduced as abbreviations as follows:

- AFp = A(true U p)
- AGp = $\neg$E(true U $\neg p$)
- EFp = E(true U p)
- EGp = $\neg$A(true U $\neg p$)
- EFp = E(true U p)
- AXp = $\neg$EX$\neg p$
- AGp = $\neg$EF$\neg p$

CTL Semantics

- $s \models p$ iff $p \in Label(s)$
- $s \models \neg\ \phi$ iff not$(s \models \phi)$
- $s \models \phi \vee \psi$ iff $(s \models \phi)$ or $(s \models \psi)$
- $s \models \mathrm{E}\phi$ iff $\sigma \models \phi$ for some $\sigma \in Path(s)$.
- $s \models \mathrm{A}\phi$ iff $\sigma \models \phi$ for all $\sigma \in Path(s)$.
- $\sigma \models \mathrm{X}\phi$ iff $\sigma_1 \models \phi$
- $\sigma \models \phi\ \mathrm{U}\ \psi$ iff $\exists\ j \geq 0$ s.t. $(\sigma[j] \models \psi$ and $\forall\ k,\ 0 \leq k < j\ (\sigma[k] \models \phi))$

2.3.5 Probabilistic Computation Tree Logic (PCTL)

Probabilistic Computation Tree Logic (PCTL) is a branching-time temporal logic based on the logic CTL [9]. A PCTL formula is a condition on a state of a Markov chain and the state either satisfies the formula or not. PCTL is like CTL with a major difference. In CTL we have universal and existential path quantifiers, but in PCTL we have the probabilistic operator $P_J(\varphi)$ where φ is a path formula and J is an interval of $[0, 1]$. In the following we describe the syntax and semantics of PCTL.

PCTL Syntax

As in CTL, PCTL distinguishes between state and path formulas. State formulas over the set AP of atomic propositions are defined by the following BNF grammar [9]:

$$\phi ::= \text{true} \mid p \mid \phi \vee \phi \mid \neg\ \phi \mid P_J(\varphi)$$

In this grammar, $p \in AP$, φ is a path formula and $J \subseteq [0, 1]$ is an interval with rational bounds.

Path formulas in PCTL are defined by the following grammar:

$$\varphi ::= \mathrm{X}\ \phi \mid \phi\ \mathrm{U}\ \phi \mid \phi\ \mathrm{U}^{\leq n}\ \phi$$

ϕ is a state formula and $n \in$ N. The path formula ϕ_1 $U^{\leq n}$ ϕ_2 has the same meaning as the until formula in CTL, but $U^{\leq n}$ is step-bounded. It means the ϕ_1 event holds and after n steps, the event ϕ_2 will hold.

PCTL Semantic:

Before giving the semantics of PCTL formulas, let us define the formalism of discrete-time Markov chain:

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

- S *is a countable, non empty 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.*

Let $p \in AP$ be an atomic proposition, $M = (S, P, \iota_{init}, AP, L)$ a Markov chain, s a state in S, ϕ, ψ PCTL state formulas, and φ a PCTL path formula. The satisfaction relation $\models$ is defined for state formulas as follows:

- $s \models a$ iff $a \in L(s)$,
- $s \models \neg \phi$ iff $s \not\models \phi$,
- $s \models \phi \wedge \psi$ iff $s \models \phi$ and $s \models \psi$,
- $s \models P_J(\varphi)$ iff $Pr(s \models \varphi) \in J$

Here, $Pr(s \models \varphi) = Pr_s\{ \pi \in Paths(s) \mid \pi \models \varphi\}$. (see [9] for more details)

Suppose a path π in M, the satisfaction relation for path formulas is defined as follows:

- $\pi \models \mathrm{X}\phi$ iff $\pi[1] \models \phi$,
- $\pi \models \phi \ \mathrm{U} \ \psi$ iff $\exists\ j \geq 0$ s.t. $(\pi[j] \models \psi \wedge (\forall\ 0 \leq k < j.\pi[k] \models \phi))$,
- $\pi \models \phi \ \mathrm{U}^{\leq n} \psi$ iff $\exists\ 0 \leq j \leq n.\ (\pi[j] \models \psi \wedge (\forall\ 0 \leq k < j.\pi[k] \models \phi))$

where for path $\pi = s_0 s_1 s_2 \ldots$ and integer $i \geq 0$, $\pi[i]$ denotes the $(i+1)^{th}$ state of π, i.e., π[i] $= s_i$.

2.4 The PRISM Model Checker

Prism is a probabilistic model checker. It supports several types of probabilistic models including discrete-time Markov chains (DTMCs) and Markov decision processes (MDPs). Models are implemented by the PRISM language, which is a state-based language. It also has a property specification language, which uses different temporal logics including PLTL, CTL, PCTL, and PCTL* (an extension of PCTL). PRISM is a free and open source software [16, 17, 20].

In the following sections, we briefly describe the syntax of the PRISM language and PRISM specification language. It should be mentioned that we do not go through all the details and we only focus on the parts, which we mostly use in our approach.

2.4.1 The PRISM Language

The building block of the PRISM language is a module. A described model by PRISM consists of a number of modules. Each module contains some local variables and the values of each of these variables are the state of the module at any given time.

Modules and Variables

The following syntax defines a module in PRISM:

```
module name ... endmodule
```

A variable in PRISM may have different values, each value shows the module state at a given time.

The syntax for defining a variable is as follows:

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

This statement defines a variable name `x`, with three possible values 0, 1, and 2 and it has been initiated to 0.

PRISM Command

The behavior of each module is described by a set of PRISM commands. The syntax of a command in PRISM is as follows:

```
[] guard → prob1 :  update1 + ...  + probn :  updaten;
```

The guard shows the present state and each update shows a transition, which can be done if the guard is true. Each update has a probability, which can be assigned to that specific transition. Here is an example of a module:

```
module M1

x :  [0..2] init 0;

[] x=0 → 0.8:(x'=0) + 0.2:(x'=1);

[] x=2 → 0.5:(x'=2) + 0.5:(x'=0);

endmodule
```

Model Type

As discussed earlier, the PRISM language supports three types of probabilistic models: discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs) and Markov decision processes (MDPs). To specify the model type, we can use one of the keywords: *dtmc*, *ctmc* or *mdp*.

Constants

We can also define constants in PRISM code:

```
const double pi = 3.141592;

const bool yes = true;
```

Initial State

We can specify the initial state in PRISM by a specific statement show as follows:

```
init x=0 endinit
```

We can also indicate more than one initial state.

Module Renaming

PRISM allows module renaming, which is a kind of module duplication and can be used in simulations and experiments.

```
module M2 = M1 [ x=y, y=x ] endmodule
```

Expressions

In PRISM, expressions contain literal values (numbers, true, false, etc.), identifiers (variables, constants, etc.), operators, which are listed as follows:

- - (unary minus)
- *, / (multiplication, division)
- +, - (addition, subtraction)
- <, <=, >=, > (relational operators)
- =, != (equality operators)
- ! (negation)
- & (conjunction)
- | (disjunction)
- ⇒(implication)
- ? (condition evaluation: condition ? a : b means "if condition is true then a else b")

Formulas and Labels

We can define formulas in PRISM. A formula contains a name and an expression. Then, the formula's name can be used instead of that expression. The following is an example of formula:

```
formula lfree = p2 = 0..4, 6, 10;
```

In this expression the formula *lfree* is defined and can be used in an expression as follows:

```
[] p1=2 & lfree → (p1'=4);
```

PRISM also contains labels. Sometimes, there are some states, which are of particular interest. By defining a label, we can specify these states. For example, in the following example *safe* is an state in which temp<= 100 or *alarm* is true.

```
label "safe" = temp<= 100 | alarm = true;
```

2.4.2 The PRISM Property Specification Language

Once the formal model is built in the PRISM language, we need to specify the properties against which the model will be checked. The PRISM property specification language is used for this purpose. This language is based on several well-known probabilistic temporal logics, including PLTL, CTL, PCTL, and PCTL*. In this section, we briefly describe this language.

The P Operators

The P operator is one of the most important operators, which is used for extracting the probability of the occurrence of an event. The syntax of using this operation is as follows:

```
P bound [ pathprop ]
```

example: `P>0.98 [ pathprop ]`

Path Properties

A path property is a formula, which is true or false for a single path in a model. The following operators are used to express path properties:

- **X :** "neXt", example: `P< 0.01 [ X y=1 ]`
- **U :** "Until", example: `P>0.5 [ z<2 U z=2 ]`
- **F :** "eventually" or "Future", example: `P<0.1 [ F z>2 ]`
- **G :** "always" or "Globally", example: `P>=0.99 [ G z<10 ]`

"Bounded" Variants of Path Properties :

"Bounded" variant means a time bound is imposed on the property. All of the above operators, except X, have "bounded" variants. The syntax of the time interval specification is "$<= t$" where t is a PRISM expression having a non-negative value. Some example of this would be:

`P>=0.98 [ F<=7 y=4 ]`

It means the probability that y becomes 4 after 7 time steps is equal or greater than 0.98.

`P>=0.98 [ G<=7 y=4 ]`

It means the probability that y staying equal to 4, 7 time steps is equal or greater than 0.98.

PLTL-Style Path Properties :

PRISM also supports the temporal logic PLTL. PLTL path properties are richer since PLTL permits combination of different temporal operators. Here, are a few examples of PLTL-style path properties:

`P>0.99 [ F ( "request" & (X "ack") ) ]`

It means the probability that a request is being received in the future and immediately acknowledged is 0.99.

`P>=1 [ G F "send" ]`

It means the probability that a message will be always sent in the future is 1.

Quantitative Properties

Sometimes, it is important to know the exact or approximate probability of a given event, rather than the probability range (i.e. the probability is below or above a specific bound). The P operator with the following syntax helps find this probability.

`P=? [ pathprop ]`

`Pmin=? [ pathprop ]` (for the minimum probability)

`Pmax=? [ pathprop ]` (for the minimum probability)

example: `P=? [ F x=5 & y=5 ]`

Chapter 3

Proposed Approach and Implementation

3.1 Introduction

As discussed in the introductory chapter, 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 rigourously verified before lunching 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, liveliness, deadlock freedom, etc. It also means checking if the web service satisfies some specific, business-logic properties. In a community setting, the community's master should verify each web service if it satisfies the community requirements before accepting it as new member. Web service verification is of a great importance not only when they should interact with each other to provide composite services or within communities [5, 15], 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 ultimate effort of verifying the whole composite service. What is more challenging is when web services can exhibit probabilistic behaviors, which means depending on the inputs and other circumstances, different choices are likely to be made and different outcomes are likely to be produced [28].

In Chapter 2, we briefly discussed the OWL-S language and PRISM model checker. In this chapter, we discuss the automation of web service verification. We present our approach of building automatically the model to be checked by transforming an OWL-S file, which contains web service description to Markov chain diagram or Markov Decision Process (MDP) and then to the PRISM code. The PRISM code is then verified by the PRISM model checker. This allows us to check web services in terms of safety, liveliness and deadlock freedom as well as if they satisfy other required properties, and if so with how many percentage.

This chapter is organized as follows. In Section 3.2, we describe our proposed approach. We give a formal definition of Markov chain diagram and MDP and describe our transformation method from OWL-S to these two formal models. To put this transformation into practice, the method is explained through a first case study. The algorithm producing the PRISM code is also presented. In Section 3.3, the properties and verification of the first case study are discussed and experimental results are shown. Section 3.4 discusses the approach through a second case study, more complete than the the first one. The implementation of our tool is described in Section 3.5.

3.2 The Proposed Approach

3.2.1 Markov Chain and MDP

Hereafter, the formal definitions of the formal models we consider in our approach, namely Discrete-Time Markov Chain (DTMC) and Markov Decision Process (MDP).

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

- S *is a countable, non empty 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.*

A Markov chain is a tuple consisting of 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 and L indicates the atomic proposition that are satisfied in a given sates. 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 an MDP is as follows.

Definition 3 *A 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* $\alpha \in Act : \sum_{s' \in S} P(s,\alpha,s') \in \{0,1\}$*;*
- $\iota_{init} : S \to [0,1]$ *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 a 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 DTMC that $P(s, \alpha, s')$ is a distribution. It means if we have n transitions from s to different states with action α, the total sum of their related probabilities will be 1. This is the reason behind using MDP instead of DTMC to model parallel actions since it allows sets of distributions per action and state rather than just a single distribution.

After presenting these two definitions, in the following we will explain our transformation approach.

3.2.2 Transforming OWL-S to Markov Chain and MDP

To describe our approach in more detail, we have chosen a simple case study described by an OWL-S file named CongoProcess.owl from OWL-S standard website [1]. It models a web service, which is used for buying books online. The flowchart of this web service is shown in Figure 3.1. We have chosen a BNF grammar to show the content of this file, where atomic processes are enclosed in " ", composite processes in < > and the names of control constructs in { }. For example, the following line:

$< S1 > \rightarrow$ "$S2$"$\{Sequence\} < S3 >$

means the composite process $S1$ is composed of a sequence of two other processes $S2$ and $S3$, where $S2$ is an atomic process and $S3$ is a composite process. Using this grammar, CongoProcess.OWL is as follows:

1. $< FullCongoBuy > \rightarrow$ "$LocateBook$"$\{Sequence\} < IF0 >$
2. $< IF0 > \rightarrow \{IF - Then\} < CongoBuyBook > \{Else\}$"$LocateBook$"
3. $< CongoBuyBook > \rightarrow < BuySequence > \{Sequence\}$"$SpecifyDeliveryDetails$"$\{Sequence\}$"$FinalizeBuy$"
4. $< BuySequence > \rightarrow$ "$PutInCard$"$\{Sequence\} < SignInAlternatives > \{Sequence\}$"$SpecifyPaymentMethod$"
5. $< SignInAlternatives > \rightarrow < CreateAcctSequence > \{Choice\} < SignInSequence >$
6. $< CreateAcctSequence > \rightarrow$ "$CreateAcct$"$\{Sequence\}$"$LoadUserProfile$"
7. $< SignInSequence > \rightarrow$ "$SignIn$"$\{Sequence\}$"$LoadUserProfile$"

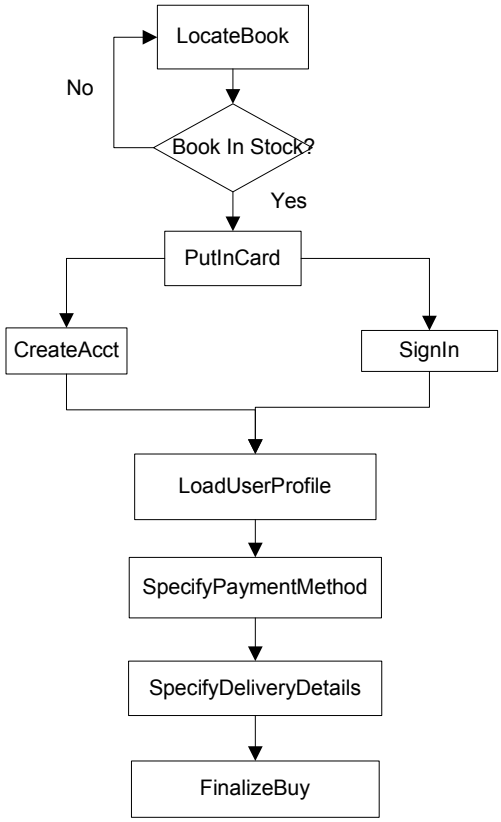

Figure 3.1: CongoProcess flow chart

Considering this file, we explain how we transform each of its elements into Markov chain elements. We should mention here that since Markov chain is a probabilistic model, which includes a probability distribution over transitions, we should have the related probability distribution in OWL-S. However, although some suggestions of extending OWL-S with probabilities using a Bayesian framework for probabilistic ontologies have been recently launched [11], the current OWL-S standard does not support stochastic events yet. To solve this problem, in this case study we suppose that the system is fair so that if we have N branches, the probability of reaching each one of them is $1/N$. This assumption will be relaxed in the second case study.

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 `CongoProcess` example, we consider each atomic process as a single state. Thus, $LocateBook$, $PutInCard$, $SignIn$, $CreateAcct$, $LoadUserProfile$, etc. are transformed to DTMC states.

Modeling Composite Processes

Each composite process is made of some other processes, atomic or composite. Such processes are connected to each other by a control construct and each control construct has its own definition, which will be discussed later. For example in the `CongoProcess` OWL-S file, $SignInSequence$ is a composite process, which is made of two atomic processes: $SignIn$ and $LoadUserProfile$ and the related control construct is sequence.

To transform composite processes into a DTMC diagram, we replace the current process with its composed processes and the transitions among the composed processes depend on the related control constructs. In the following subsection, we describe in detail how each control construct defines transitions among its composed processes.

Finally, if the new replaced processes are composite too, then they are replaced by their composed processes in a similar way, and this algorithm is 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. The details of this recursive algorithm will be presented later when we discuss the implementation of our tool in Section 3.5.2.

Modeling Control Constructs

Sequence: A sequence is a finite set of processes, which are executed in a sequential order. Thus, we model a sequence of processes using DTMC states sequentially related to each other through transitions. In our example, *SignInSequence*, which is 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 3.2.

Figure 3.2: Modeling sequence into Markov chain

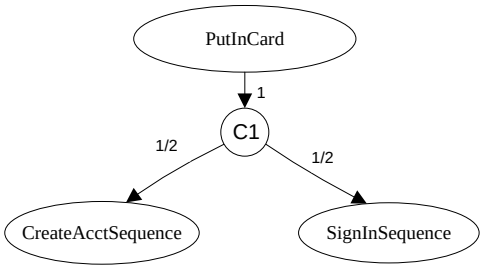

Figure 3.3: Modeling choice into Markov chain

Choice: 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 fact, the composite process transits first to an atomic process named Ci, where i is number of choice constructs in the diagram (the first choice construct is model by $C1$). This allows substituting the choice process by an atomic one. Then, the atomic process is connected to the inner processes. The probability of choosing each process is $1/NP$, where NP is the number of processes. In our example, *SignInAlternatives* is a choice composite process, which is replaced by two processes: *CreateAcctSequence* and *SignInSequence* and since $NP = 2$, the probability of reaching each state is 1/2 (see Figure 3.3). It should be mentioned that since these two processes are still composite processes, they should be decomposed into inner processes.

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 f 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 earlier, each atomic process is associated with a single state and each composite process is replaced by a composite set of states. To explain this procedure, let us consider a prototypical

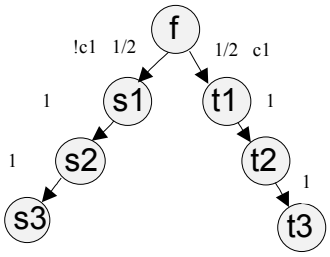

Figure 3.4: Modeling if-then-else into Markov chain

example. Suppose we have a condition named $c1$ so if it holds, we want to execute the sequence processes of $t1$, $t2$, and $t3$, if not, the sequence of $s1$, $s2$, and $s3$ should be executed. Then the resulting DTMC is as shown in Figure 3.4. If we suppose the system fair, the probability of each transition is 1/2. It should be mentioned that all the processes are considered atomic; otherwise, each composite process should be replaced with its related composed processes.

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 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 is itself 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 $w0$ and the *while-condition* $c1$. The *while body* consists of processes named: $s1$, $s2$, and $s3$, and the next process after the loop body is $s4$. Again, we suppose that all the processes are atomic. The resulting DTMC is illustrated in Figure 3.5.

A similar algorithm can be applied to the *repeat-until* control. The associated DTMC is shown in Figure 3.6.

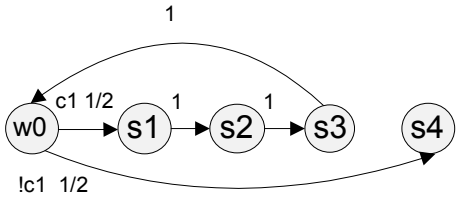

Figure 3.5: Modeling repeat-while into Markov chain

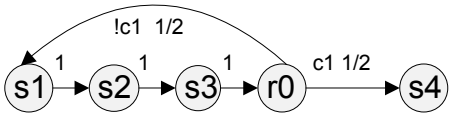

Figure 3.6: Modeling repeat-until into Markov chain

Split, Split-Join: *Split* shows processes, which are executed concurrently, and *split-join* shows processes, which also 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 $t0$, 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 $p0$, $p1$, $q0$, $q1$, and $r0$, $r1$. The corresponding MDP is shown in Figure 3.7, where the probability of each parallel transition is 1. The transitions are also labeled using fictive actions.

A similar algorithm is used for the transformation of *split-join*, except that all split processes will join to reach a given process. Figure 3.8 shows the MDP associated with the above prototypical example after adding a new atomic joining process $u0$.

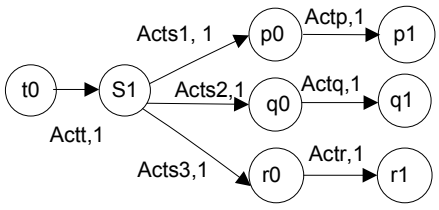

Figure 3.7: Modeling split into Markov decision process

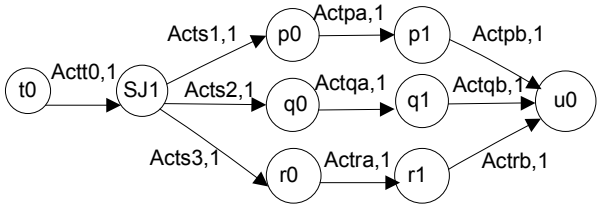

Figure 3.8: Modeling split-join into Markov decision process

Now we present the soundness and complexness results of our transformation technique.

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

Proof. *On the 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 straithforward 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.* $\diamondsuit$

3.2.3 Transforming OWL-S to the PRISM Code

After transforming OWL-S into DTMC or MDP, the approach considers these two formalisms to produce the PRISM code. The algorithm is as follows:

a) First, we define an integer variable and its value range from zero to the maximum number of states, and then initialize it to 0. For example the following PRISM code defines and initializes the variable s, which ranges from 0 to 5, where each value shows a different state.

```
s:  [0..5] init 0;
```

b) After defining the states, for each "if-then-else" control construct, we define the related "if-condition" as a boolean variable and initialize it to `true`. For example if we have an if-condition named `BookInStock`, we define it as follows:

```
BookInStock :  bool init true ;
```

c) After defining the state variables, we define the transitions between those states using PRISM commands. Each PRISM command starts with `[]` and has two parts separated by $\rightarrow$. The left part of $\rightarrow$ is called guard and the right part is called update. Each command might have one or more updates. To define a transition in DTMC or MDP into the PRISM code, we simply set the defined integer variable s, which shows the states, to the number assigned for the source state in the guard part. Then, we create an update part and set the s variable to the number assigned to the destination state in this part.

d) If there is a condition for a transition, we update the guard part of the command regarding this transition by specifying if the condition is true or false.

e) If there are more than one transition from the source state, we will have one guard and different updates separated by "+". Since in this case each transition has a related probability, we simply use the PRISM syntax and specify the different probabilities using the delimiter ":".

Using this algorithm, the produced PRISM code for the different control constructs discussed in the previous section is as follows:

Sequence

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

Choice

In the above example for choice if we suppose that *CreateAcctSequence* and *SignInSequence* are atomic processes, then we obtain:

```
s:  [0..3] init 0;
```

```
[] s=1 → 1/2:(s'=2) + 1/2:(s'=3) ;
```

If-Then-Else

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

[] c1 = true & s=0 → (s'=1);

[] c1 = false & s=0 → (s'=2);

[] s=0 → 1/2:(s'=1) + 1/2:(s'=2) ;
```

Repeat-While

```
s:  [0..4] init 0;

[] c1 = true & s=0 → (s'=1);

[] s=3 → s'=1;

[] c1 = false & s=1 → (s=4);

[] s=0 → 1/2:(s'=1) + 1/2:(s'=4) ;
```

Repeat-Until

```
s:  [0..4] init 0;

[] c1 = true & s=3 → (s'=4);

[] c1 = false & s=3 → (s'=0);

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

Split

```
s:  [0..6] init 0;

[Actt] s=0 → (s'=1);

[Acts1] s=1 → (s'=2);

[Acts2] s=1 → (s'=3);

[Acts3] s=1 → (s'=4);
```

```
[Actp] s=2 → (s'=5);

[Actq] s=3 → (s'=6);

[Actr] s=4 → (s'=7);
```

Split-Join

```
s:  [0..8] init 0;

[Actt0] s=0 → (s'=1);

[Acts1] s=1 → (s'=2);

[Acts2] s=1 → (s'=3);

[Acts3] s=1 → (s'=4);

[Actpa] s=2 → (s'=5);

[Actqa] s=3 → (s'=6);

[Actra] s=4 → (s'=7);

[Actpb] s=5 → (s'=8);

[Actqb] s=6 → (s'=8);

[Actrb] s=7 → (s'=8);
```

The automatic DTMC output for the CongoProcess.owl example is shown in Figure 3.9 and the related PRISM code is as follows:

```
module x

s:[0..9] init 0 ;

BookInStock :  bool init true ;

[] s=0 → (s'=1);

[] s=1 & BookInStock=true → (s'=2);

[] s=1 & BookInStock=false → (s'=0);

[] s=1 → 1/2:(s'=2) + 1/2:(s'=0) ;

[] s=2 → (s'=3);

[] s=3 → 1/2:(s'=4) + 1/2:(s'=9) ;
```

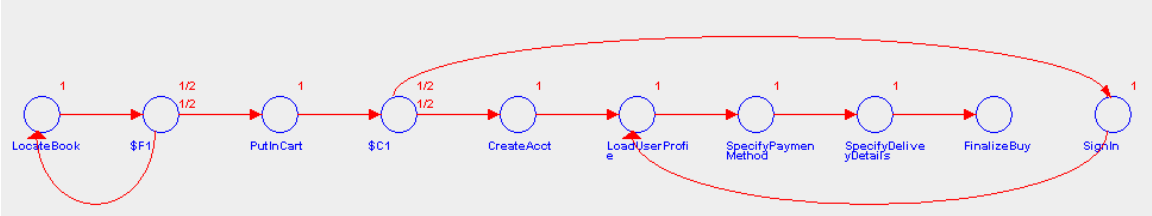

Figure 3.9: DTMC for CongoProcess example

```
[] s=4 → (s'=5);

[] s=5 → (s'=6);

[] s=6 → (s'=7);

[] s=7 → (s'=8);

[] s=9 → (s'=5);

[] s=8 → (s'=8);

endmodule
```

3.3 Properties and Verification Results

Once the PRISM code is produced, we use the PRISM model checker to run the verification. Different properties such as safety, liveness and deadlock freedom can be checked. To specify those properties, we use Probabilistic Computation Tree Logic (PCTL) introduced in Chapter 2. We consider our example of the CongoProcess system to illustrate examples of properties. To make these properties readable, we define labels for the states, so that we can use labels instead of state numbers. For example, we label the state $s = 9$ as "$SignIn$".

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

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 "$terminate$"$= s = 8$;

Thus, this property means the probability of terminating the execution of the web service process by reaching the "$finalizeBuy$" 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.

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

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 cartd

- $SpecifyPaymentMethod$ & $SpecifyDeliveryDetails \Rightarrow P >= 1$ [F "$terminate$"]

This formula expresses the following business logic property: if the payment method or delivery details are specified correctly, then the probability of terminating by reaching an acceptance (final) state is greater than or equal to 1. The property is satisfied in our model.

- $BookInStock \Rightarrow P >= 1$ [F "$terminate$"]

This formula expresses the following business logic property: if there is a book in the stock, then the probability of terminating by reaching an acceptance (final) state is greater than or equal to 1. The property is satisfied in our model.

- $Pmax =?$ [$PutInCard\ U\ F\ SignIn$]

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.

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

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.

- $Pmax =? [F\ CreateAcct]$

This property estimates the probability that a user creates an account. The maximum of this property is 0.5 in this system.

- $Pmax =? [F\ SignIn]$

This property estimates the probability that a user signs in through an account. The maximum of this property is 0.5 in this system.

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.

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 3.1 and some properties are shown in Figure 3.10. PRISM also has the capability of drawing graphs for each property and its percentage for different N values, where N is the number of times the experiment happens. In this example, we draw the graph for this property: $Pmax =? [PutInCard\ U\ F\ SignIn]$ and since the percentage is the same (0.5) for all the experiments, the graph is a straight line (see Figure 3.11).

Table 3.1: CongoProcess Verification Results Using PRISM

	Experiment 1 24 Processes	**Experiment 2 32 Processes**	**Experiment 3 40 Processes**	**Experiment 4 48 Processes**
Number of States	512	4,096	32,768	262,144
Number of Transitions	1,728	18,432	184,320	1,769,472
Construction Time (sec.)	0.001	0.032	0.047	0.062
Verification Time (sec.)	20.53	31.43	47.3	62.32

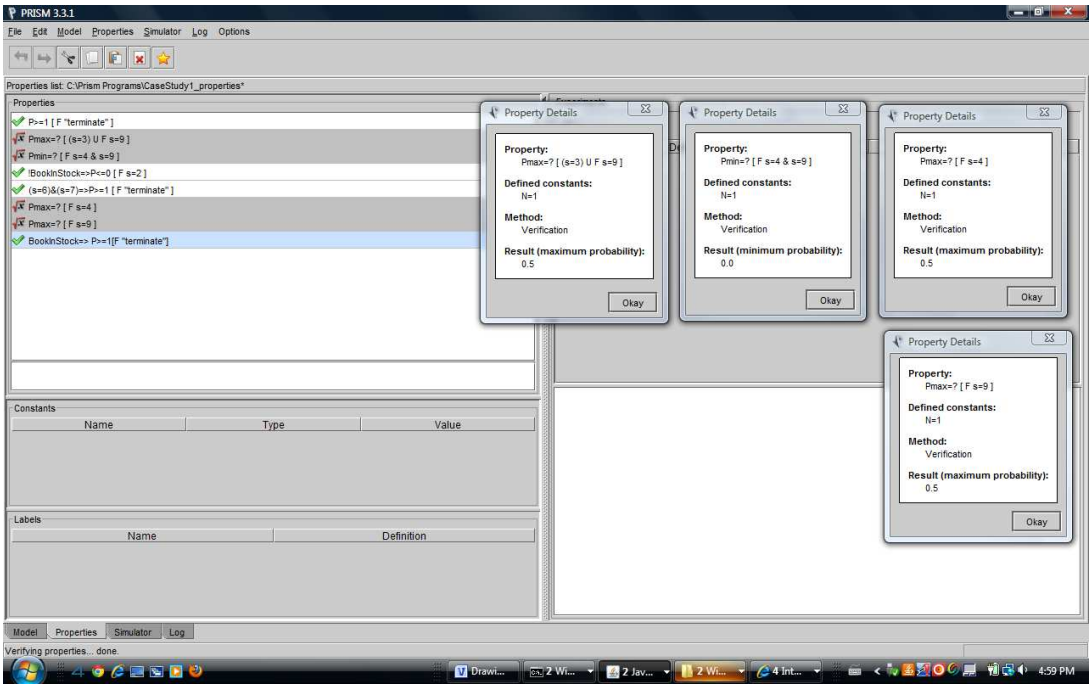

Figure 3.10: PRISM properties verification for CongoProcess

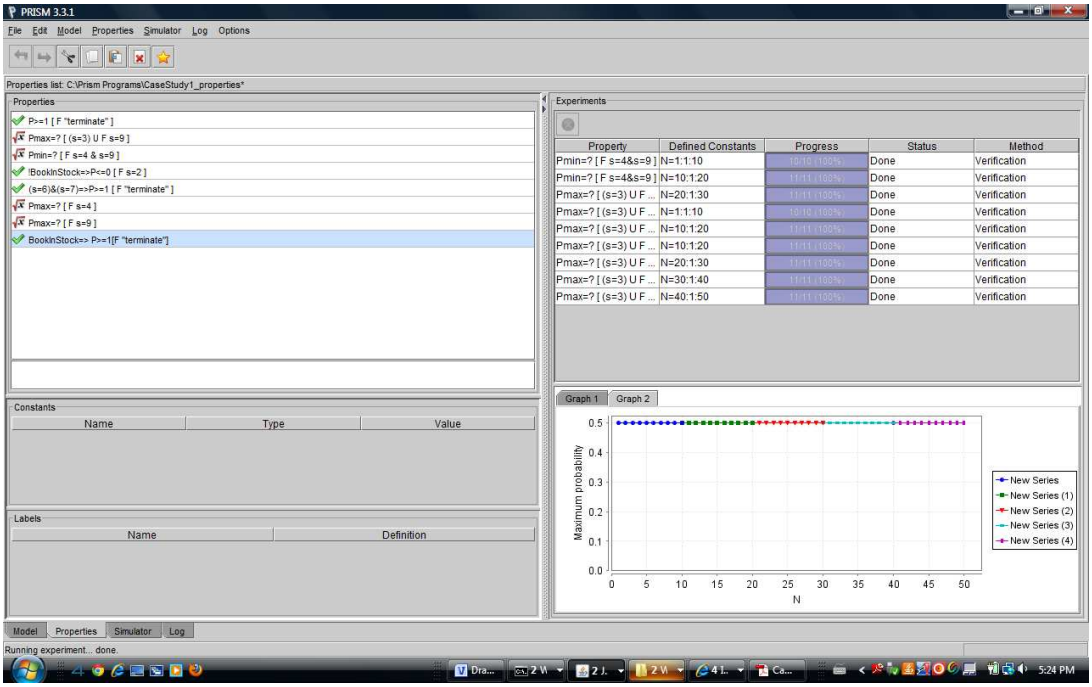

Figure 3.11: PRISM properties graph for CongoProcess

3.4 Detailed Case Study

To explain our approach in more details, we have chosen another case study. Since OWL-S is a new topic and there are very few OWL-S examples, which are very small case studies, we have chosen an activity diagram for an online sale system implemented as a web service. The corresponding OWL-S specification is then created from the activity diagram. 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 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 3.12 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. $< OrderingProcess > \rightarrow$ "$PlaceOrder$"$\{Seq\} < Ordering >$
2. $< Ordering > \rightarrow$ "$CancelOrder$"$\{Choice\} < OrderCreditChecking >$
3. $< OrderCreditChecking > \rightarrow$ "$ProcessOrder$"$\{Seq\} < CreditChecking >$
4. $< CreditChecking > \rightarrow \{IF - Then\} < StockChecking > \{Else\} < OrderCanceling >$
5. $< StockChecking > \rightarrow \{IF - Then\} < WarehouseProcess > \{Else\} < NewDeliveryTime >$
6. $< OrderCanceling > \rightarrow$ "$ContactCustomer$"$\{Seq\}$"$CancelOrder$"
7. $< NewDeliveryTime > \rightarrow < ChangeDeliveryTime > \{Choice\} < WarehouseProcess >$
8. $< WarehouseProcess > \rightarrow$ "$OrderToWarehouse$"$\{Seq\} < ShippingReqProcess >$
9. $< ChangeDeliveryTime > \rightarrow$ "$CheckNewDeliveryTime$"$\{Seq\}$"$InformCustomerNewData$"
10. $< ShippingReqProcess > \rightarrow < RequestProcess > \{Seq\} < ShippingProcess >$
11. $< RequestProcess > \rightarrow < ShippingReq > \{SplitJoin\} < InvoiceProcessing >$
12. $< ShippingProcess > \rightarrow$ "$ItemInvoicePacking$"$\{Seq\}$"$ItemInvoiceShipping$"

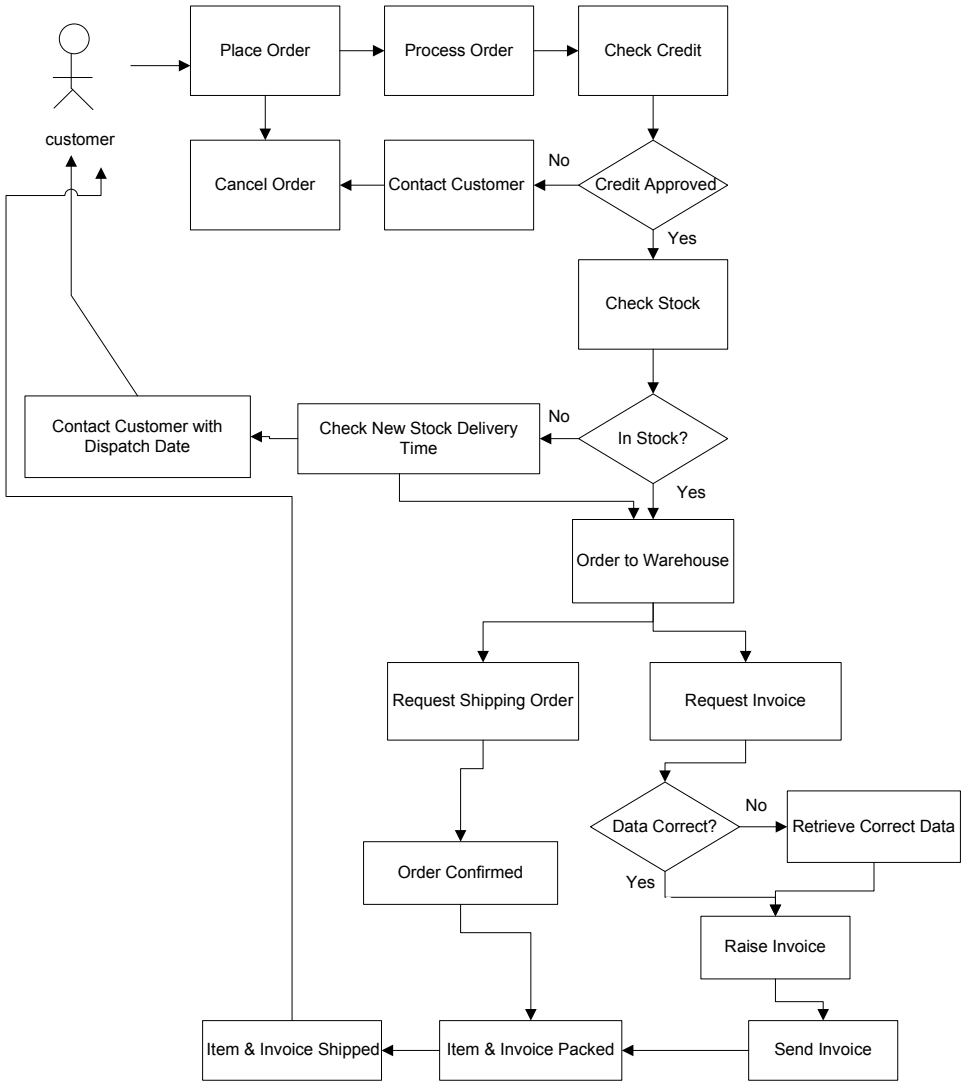

Figure 3.12: Online sale chart

13. $< ShippingReq > \rightarrow$ "$RequestShipping$"$\{Seq\}$"$OrderConfirmed$"

14. $< InvoiceProcessing > \rightarrow$ "$RequestInvoice$"$\{Seq\} < DataCorrection >$

15. $< DataCorrection > \rightarrow \{IF - Then\} < RaiseSendInvoice > \{Else\} < RetrieveData >$

16. $< RaiseSendInvoice > \rightarrow$ "$RaiseInvoice$"$\{Seq\}$"$SendInvoice$"

17. $< RetrieveData > \rightarrow$ "$RetData$"$\{Seq\} < RaiseSendInvoice >$

Figure 3.13 shows the DTMC produced by our tool for this case study. The related PRISM code generated by the tool is as follows:

```
module a
```

```
s:[0..20] init 0 ;

CreditChecknigOk :  bool init true ;

StockCheckingOk :  bool init true ;

DataIsCorrect :  bool init true ;

[] s=0 → (s'=1);

[] s=1 → 0.25:(s'=2) + 0.75:(s'=3) ;

[] s=3 → (s'=4);

[] s=4 & CreditChecknigOk = true → (s'=5);

[] s=4 & CreditChecknigOk = false → (s'=20);

[] s=4 → 0.66:(s'=5) + 0.34:(s'=20) ;

[] s=5 & StockCheckingOk = true → (s'=6);

[] s=5 & StockCheckingOk = false → (s'=17);

[] s=5 → 0.6:(s'=6) + 0.4:(s'=17) ;

[] s=6 → (s'=7);

[] s=7 → (s'=8);

[] s=8 → (s'=9);

[] s=9 → (s'=10);

[] s=10 → (s'=11);

[] s=7 → (s'=12);

[] s=12 → (s'=13);

[] s=13 & DataIsCorrect = true → (s'=14);

[] s=13 & DataIsCorrect = false → (s'=16);

[] s=13 → 0.8:(s'=14) + 0.2:(s'=16) ;

[] s=14 → (s'=15);

[] s=15 → (s'=10);

[] s=16 → (s'=14);
```

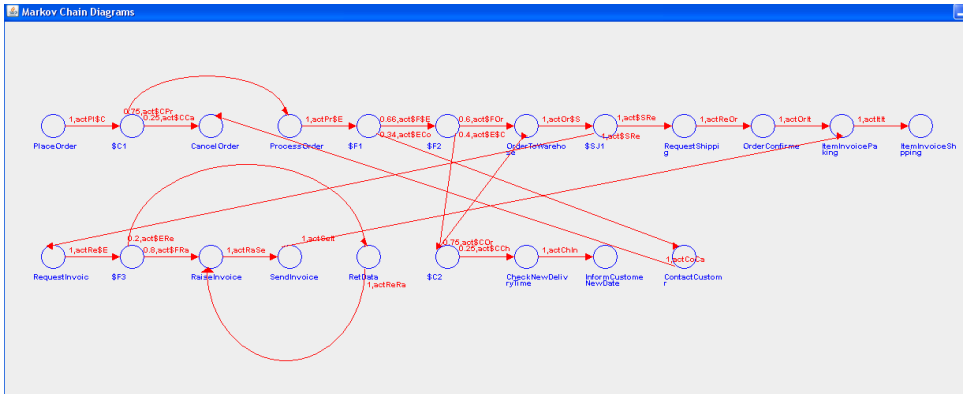

Figure 3.13: Online sale Markov chain diagram

```
[] s=17 → 0.25:(s'=18) + 0.75:(s'=6) ;

[] s=18 → (s'=19);

[] s=20 → (s'=2);

endmodule
```

3.4.1 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. In this section, we will give examples of some properties we have checked for the online sale scenario. To make these properties readable, like previous case study, we define labels for each state so that we can use these labels instead of the state numbers.

- $!CreditCheckingOk \Rightarrow P >= 1\ [FContactCustomer]$

This business logic property, satisfied in our model, says: if a client doesn't have any credit problem, the probability that the system contacts this client is 1.

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

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.

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

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.

- $!CreditCheckingOk \mid !StockCheckingOk \Rightarrow P <= 0\ [F\ \text{“}accept\text{”}]$

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.

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

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.

- $RequestShipped \Rightarrow P >= 1\ [F\ \text{“}accept\text{”}]$

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 `Online Sale` system.

- $Pmax =?[F\text{“accept”}]$

This property estimates the probability that the algorithm will terminate in an acceptance state. The system reveals that this probability is 0.75. It should be mentioned that this value is computed based on the assumption that credit checking process is satisfactory, there is enough products in stock, and the entered data is correct.

- $Pmin =?[FCheckNewDeliveryTime]$

This property estimates the minimum probability that a new delivery time will be checked for a given product. The related probability is 0, meaning that in certain computations, no new delivery time is checked for certain products.

- $Pmin = ?[FCancelOrder]$

This property estimates the minimum probability that an order will be cancelled either by the customer or because of a bad credit history. This probability is 0.25.

- $Pmax = ?[FContactCustomer|InformCustomerNewDate]$

This property estimates the probability that a customer will be contacted in the future. The system computes this probability to be 0.3045.

- $Pmax = ?[FOrderToWarehouse]$

This property estimates the probability that an order is made to the warehouse. The computed probability is 0.75.

- $Pmax = ?[FOrderConfirmed]$

This property estimates the probability that an order is confirmed. The estimated value in our simulation is 0.75.

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.

3.4.2 Experimental Results

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 3.2. Snapshots from the verification process in PRISM are shown in Figures 3.14 and 3.15. The graphs for the following properties: $Pmin = ?[FCancelOrder]$, $Pmax = ?[FContactCustomer|$ $InformCustomerNewDate]$, and $Pmax = ?[FOrderToWarehouse]$ are shown in Figure 3.16.

Table 3.2: Online Sale Verification Results Using PRISM

	Experiment 1 63 Processes	**Experiment 2 84 Processes**	**Experiment 3 105 Processes**	**Experiment 4 126 Processes**
Number of States	9,261	194,481	4,084,101	85,766,121
Number of Transitions	35,748	1,000,269	26,255,178	661,625,091
Construction Time (sec.)	0.046	0.11	0.125	0.25
Verification Time (sec.)	25.18	42.13	63.99	91.29

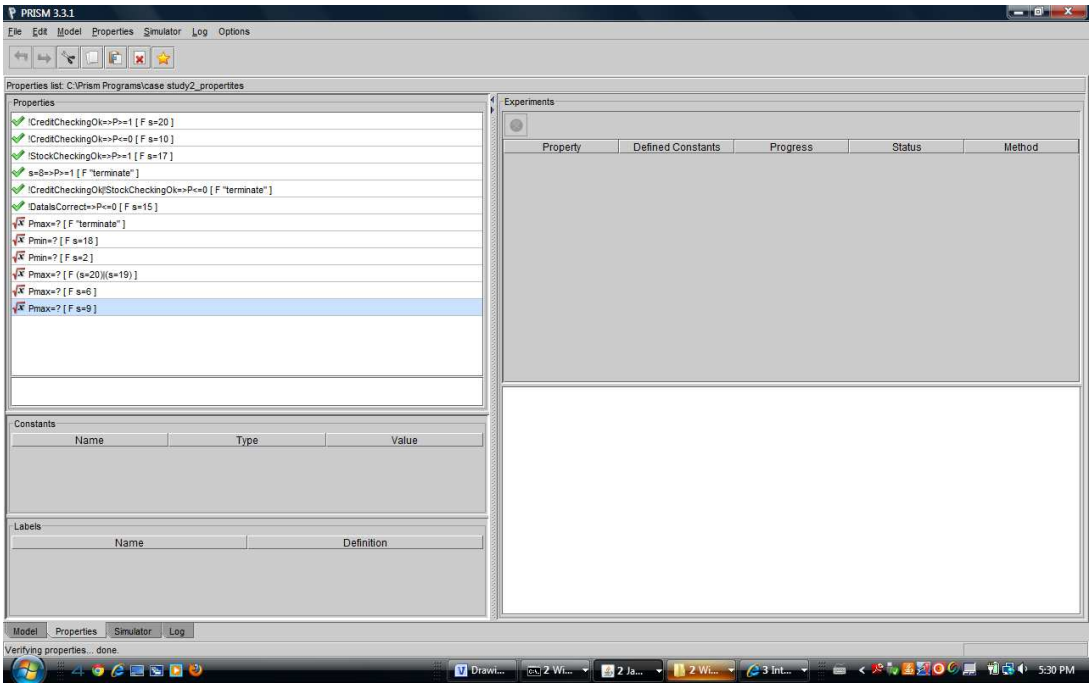

Figure 3.14: Properties verification for the online sale scenario

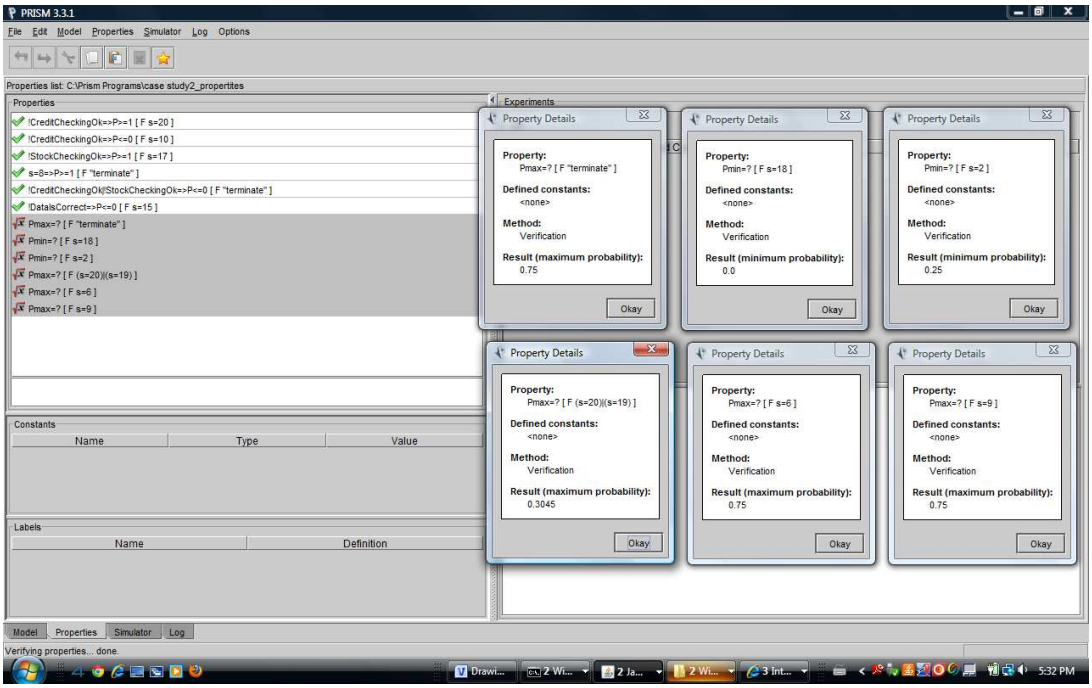

Figure 3.15: Properties satisfaction percentages for the online sale scenario

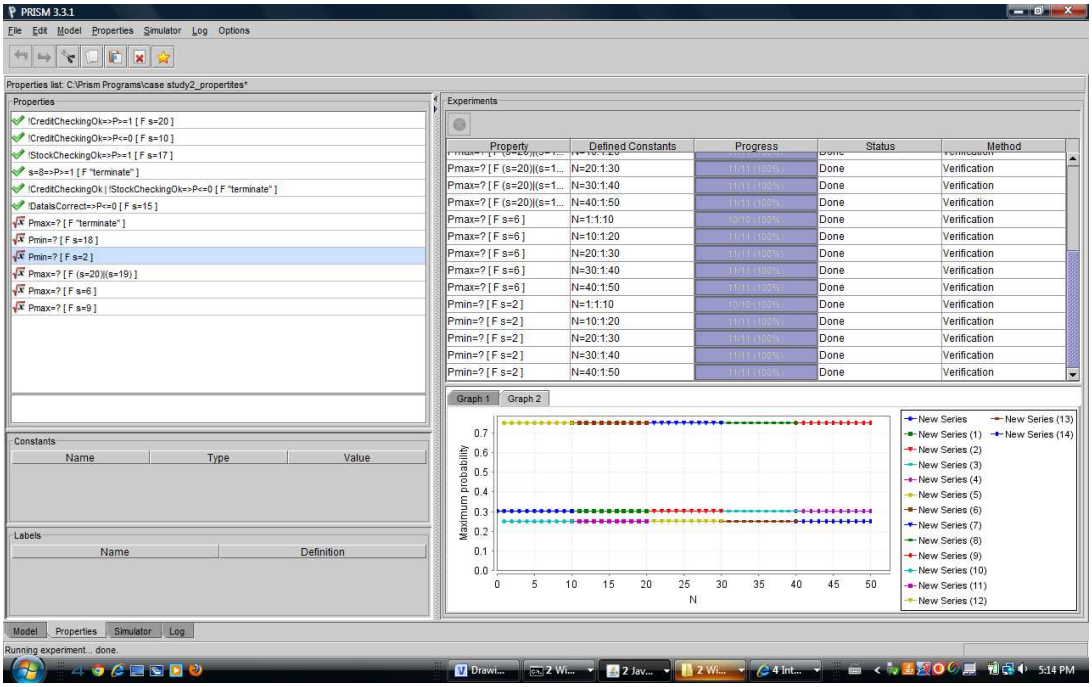

Figure 3.16: PRISM properties graph for the online sale scenario

3.5 Implementation

In this section, first, we introduce the tool architecture and its different parts, then we describe the details of implementing each part. We discuss the main classes and their specific roles, which are implemented with Java in the Eclipse environment. Finally, we explain the main algorithms of the program.

3.5.1 Architecture of the Tool

As described before, the tool takes a web service description in form of OWL-S file as input and produces DTMC/MDP and the PRISM source code as outputs. From the architectural perspective, the tool is composed of three different parts: Parser, DTMC and MDP Generator and PRISM Code Generator.

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 and called `Internal Representation I` 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 OWL-S versions emerge. Furthermore, this internal representation 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. Furthermore, the generator creates the corresponding DTMC or MDP depending on the constructs included in the input file.

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 3.17 depicts the tool architecture showing its different parts. The system implementation is discussed next.

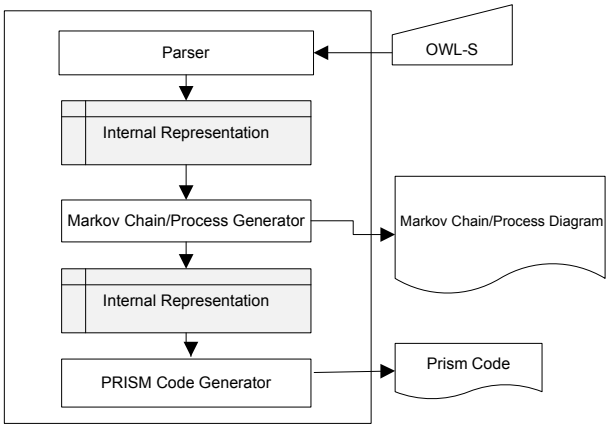

Figure 3.17: Tool internal architecture

3.5.2 Tool Implementation

In this section, we describe the implementation of the three main parts of our tool: Parser, DTMC and MDP Generator and PRISM Code Generator.

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 element 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. The outcome of executing this algorithm is a linked list structure named `Internal Representation I`, which is then used by the DTMC/MDP generator. Figure 3.18 shows the `Internal Representation I` for the CongoProcess Case Study.

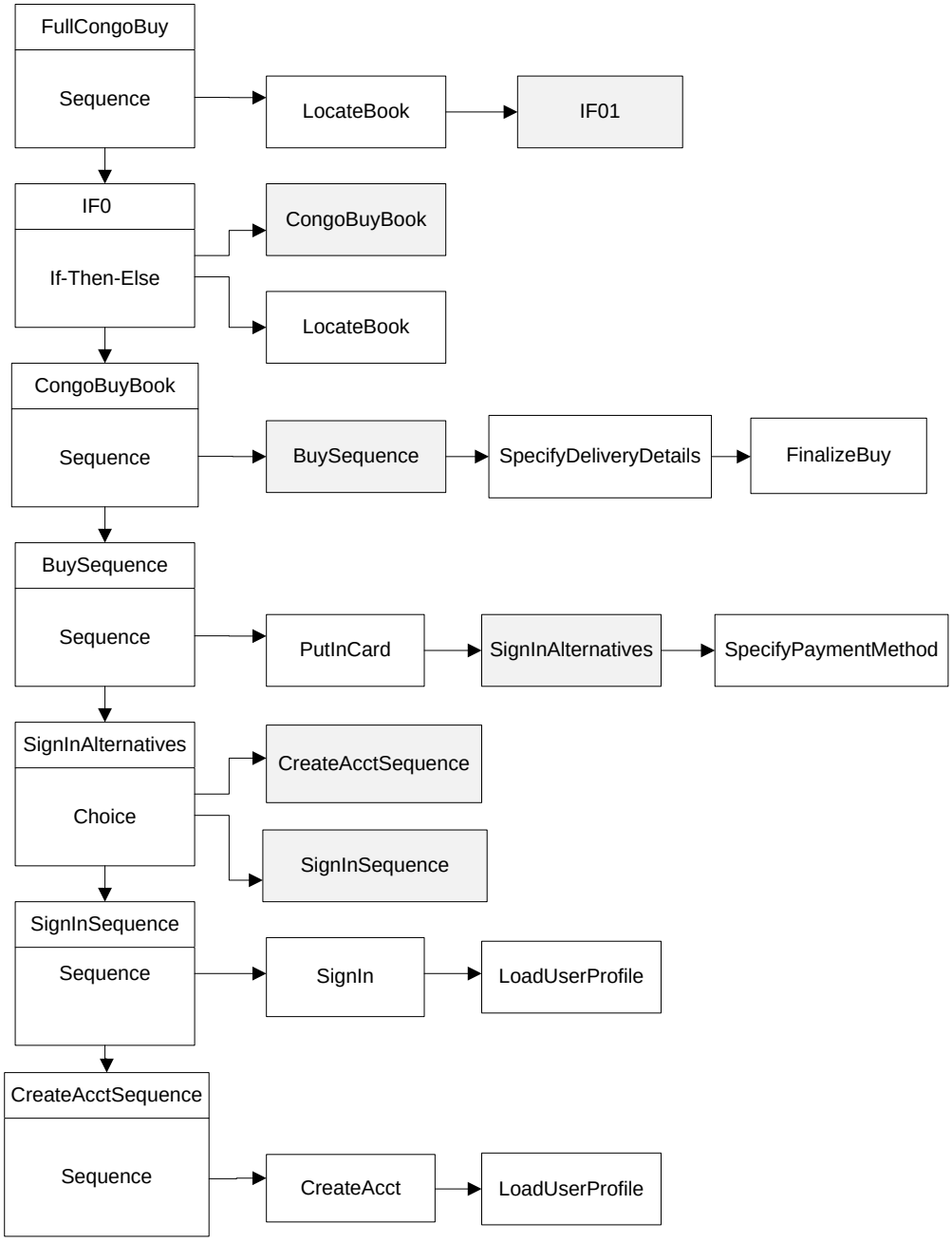

Figure 3.18: Internal representation for the CongoProcess scenario

DTMC and MDP Generator Implementation

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 track of. Then, the algorithm stores the paths into the `Internal 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 3.2.2 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 inner process 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 DTMC/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 3.2.3.

3.5.3 Program Structure

The program has been written in Java under the Eclipse environment. A package named `Parser` containing different classes has been implemented. The main classes are:

Process:

This class provides a representation for each process. When we want to create a process, atomic or composite, we create an object of this class, along with the following attributes:

name: It keeps the name of process, which we extract from the OWL-S file.

type: It shows the type of process, for isntance *atomic*, *If-Then-Else*, *Choice*, etc.

title: For some processes, like *Choice* we should add an extra process, which does not exist in OWL-S, so it has just a title.

id: Each process is referenced by a unique id.

X,Y: It keeps the position of each DTMC/MDP state on the screen.

splitProcesses: It is an ArrayList to keep the inner processes names for the current process if the process type is: *Sequence*, *Repeat-While* or *Repeat-Until* .

choiceProcesses: It is a two dimensional ArrayList where each of its rows contains the name of each branch.

ifCondition: If the process is *If-Then-Else*, this property keeps the *if condition*.

ifProcesses: It is an ArrayList to keep the inner processes names, which exist in the If body, if the process is of type *If-Then-Else*.

elseProcesses: It is an ArrayList to keep the inner processes names, which exist in the Else body, if the process is of type *If-Then-Else*.

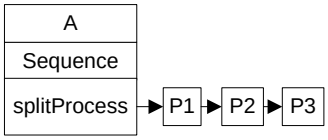

Figure 3.19: Internal representation for sequence process

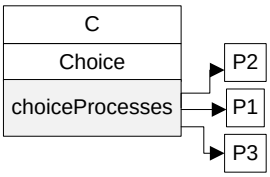

Figure 3.20: Internal representation for choice process

loopCondition: If the process is of type *loop*, such as *Repeat-While* or *Repeat-Until*, this property keeps the related condition.

counted: When we traverse a node, we set this boolean property to `true`, which prevents us counting a node twice.

The methods of this class are getters and setters for these properties. The internal representations for *Sequence*, *Choice* and *If-Then-Else* are shown in Figures 3.19, 3.20 and 3.21. It should be mentioned that the *Repeat-While* and *Repeat-Until* control constructs have the same internal representation as *Sequence*, and *Split* and *Split-Join* control constructs have the same internal representation as *Choice*.

Parser:

As discussed earlier in this chapters, the parser reads the OWL-S file and changes it into the `Internal Representation I`, then the DTMC/MDP generator reads the `Internal Representation I` and creates the `Internal Representation II`. These actions are implemented in the *Parser* Class.

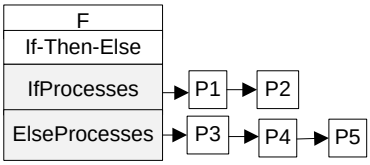

Figure 3.21: Internal representation for if-then-else Process

The class *guiFrame* reads the `Internal Representation II` and draws the related DTMC/MDP Diagram. The key attributes of the *Parser* class are:

AtomicProcesses: It is an ArrayList where the parser records the atomic processes extracted from the OWL-S file.

CompositeProcesses: It is an ArrayList where the extracted composite processes are recorded.

ResultList: It is a two dimensional ArrayList, which shows different paths in the DTMC/MDP Diagram. It is created by a method called `processingMainArrays`.

Transitions: It is a two dimensional ArrayList, which stores all transitions in a DTMC/MDP with their probabilities. In fact `Transitions` represents the `Internal Representation II`.

In the following, we describe some of the important methods of the *Parser* Class:

readOWLFile: This method reads and parses the OWL-S file so that if there is an atomic element, it adds it to the `AtomicProcesses` ArrayList, and if there is a composite element, it adds it to the `CompositeProcesses` ArrayList along with its type: sequence, choice, etc.

readCompositeProcess: It is a recursive function called by `ReadOWLFile` if a composite element is found. It reads the composite element and identifies its type: sequence, choice, etc. and add a child node for the parent composite process, which has been created relative to a composite element. If the inner element is still composite, this function will be called recursively.

processingMainArrays: This method reads the `CompositeProcesses`, which represents the `Internal Representation I` and builds the `resultlList`. It reads each node of `CompositeProcesses` and processes it by calling `decomposeCompositeProcess` and creates `resultList`. Then the method `extractTransitions` reads the `totalList` and builds the `Transitions` ArrayList, which is the `Internal Representation II`. The function `drawMarkovChainDiagram` uses this ArryList and draws the DTMC/MDP diagram. In fact, the functions: `processingMainArrays`, `extractTransitions` and `drawMarkovChainDiagram` are the core of the DTMC and MDP Generator part of the tool.

decomposeCompositeProcess: It is called by `processingMainArrays` method and processes each node of CompositeProcesses ArrayList. Its algorithm is recursive. If the process is atomic,

the algorithm adds it as a state of a path to the `equvalentList`. If the process is composite, the algorithm adds it to a specific list according to the process type. The function is recursively called for inner processes. The recursion continues until all composite processes are decomposed to atomic processes. Then by merging all the current lists, resultList will be created, which is a two dimensional ArrayList showing different paths in the DTMC/MDP diagram. The algorithm of this method is shown in Algorithm 3.1.

extractTransitions: It reads `resultList` and processes it to create the two dimensional ArrayList Transitions (`Internal Representation II`).

drawMarkovChainDiagram: It reads the `Transitions` ArrayList and draws the corresponding DTMC/MDP.

MakePrismFile: It reads `Transitions` ArrayList and generates the PRISM source code. In fact, this method is the core of the "PRISM Code Generator" part of the tool.

guiFrame:

This class produces the graphical interface of the program. It is an extension of the JFrame class and has a method named `paint`, which draws the DTMC/MDP. In fact, the method reads the `Internal Representation II` created by Parser methods and draw the DTMC/MDP diagram. The way it works is as follows: It reads the `AtomicProcesses` list produced by the Parser class and draws a circle corresponding to each atomic process and stores the X and Y of the atomic process. As a second stage, it reads the `Transitions` list and draws a line between two atomic processes, which are connected to each other. Figure 3.22 shows the mapping between the implementation methods and tool architecture parts.

Algorithm 3.1 *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)
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)
         resultLits = merge (choiceList, innerList)
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)
         resultLits = merge (choiceList, innerList)
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)
         resultLits = merge (choiceList, innerList)
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)
    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);
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)
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)
return resultList
```

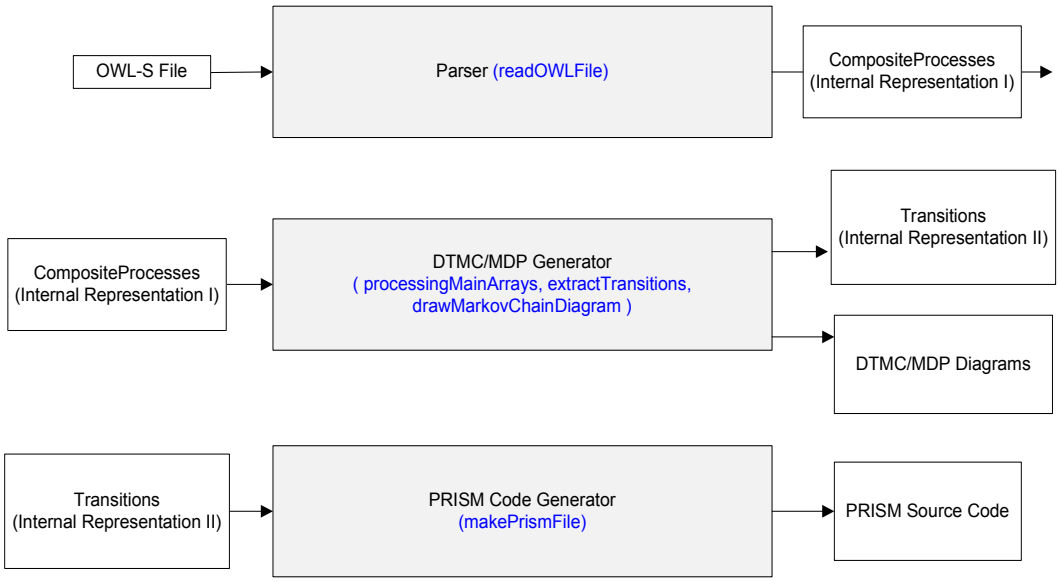

Figure 3.22: Mapping between the tool parts and implementation methods

Chapter 4

Conclusion and Future Work

4.1 Conclusion

In this thesis, we proposed a new approach towards verifying web services using a probabilistic model checking technique, where probabilistic and non-probabilistic properties can be checked. In this approach, we use OWL-S as web service description. The main idea of the approach is to read and parse the OWL-S file of a web service and transform it to a stochastic model, namely Markov chain diagram and Markov Decision Process. Thereafter, this formal model representing the system to be checked is transformed into the PRISM code. The resulting code together with the required properties against which the system will be checked are used as inputs to PRISM, a probabilistic model checker that supports CTL, PLTL, and PCTL logics. The PRISM model checker shows then the properties that are satisfied and with how many percentage.

In this thesis, first we introduce some rules for transforming OWL-S model into Markov chain diagram and Markov decision process and an algorithm to automatically generate the PRISM code from the Markov model. We implemented the algorithm in a tool where the whole verification process can be performed automatically. We have discussed our approach using two case studies and the simulation results shows that the whole approach is promising in terms of execution time.

4.2 Limitations and Future Work

One of the limitations of this work is the common problem of model checking, which is state explosion problem. The other limitation is the extraction of probabilities out of OWL-S. In fact, in the current approach we assume that OWL-S contains the probabilities for each process as parameter, but we may design PR-OWL-S, which can be an extension of OWL-S similar to PR-OWL, an ongoing project for probabilistic ontology [11]. In PR-OWL-S, we can consider the probabilities that the system has as well as external events which may affect those probabilities. Developing and extending PR-OWL-S is one of the directions for future work.

On the other hand, 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 [15], where communication protocols between the community master and slaves are characterized by their uncertainty, which fits well with our probabilistic-based approach. Collaboration and substitution protocols among slave web services within the same community and between communities also need to be rigourously checked before implementation. Here again extending our probabilistic verification framework to account for the stochastic nature of these protocols seems promising.

Bibliography

[1] W3c: Owl-s: Semantic markup for web services web site, 22 November 2004.

[2] W3c: Owl web ontology language guide, 12 November 2009.

[3] W3c: Owl web ontology language use cases and requirements, 12 November 2009.

[4] Anupriya Ankolekar, Massimo Paolucci, and Katia P. Sycara. Towards a formal verification of OWL-S process models. In *Proc. of the International Semantic Web Conference (ISWC)*, volume 3729 of *Lecture Notes in Computer Science*, pages 37–51, 2005.

[5] Jamal Bentahar, Zakaria Maamar, Djamal Benslimane, and Philippe Thiran. An argumentation framework for communities of web services. *IEEE Intelligent Systems*, 22(6):75–83, 2007.

[6] Jamal Bentahar, Zakaria Maamar, Wei Wan, Djamal Benslimane, Philippe Thiran, and Sattanathan Subramanian. Agent-based communities of web services: An argumentation-driven approach. *Service Oriented Computing and Applications*, 2(4):219–238, 2008.

[7] Daniela Berardi, Giuseppe De Giacomo, Massimo Mecella, and Diego Calvanese. Composing web services with nondeterministic behavior. In *Proc. of the 2006 IEEE International Conference on Web Services (ICWS)*, pages 909–912, 2006.

[8] Tien-Dung Cao, Trung-Tien Phan-Quang, Patrick Felix, and Richard Castanet. Automated runtime verification for web services. In *Proc. of the IEEE International Conference on Web Services (ICWS)*, pages 76–82, 2010.

[9] Joost-Pieter Katoen Christel Baier. *Principles of Model Checking*. MIT Press, Cambridge, Massachusetts, 2008.

[10] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 2000.

[11] Paulo C. G. Costa. *Bayesian Semantics for the Semantic Web*. PhD thesis, School of Information Technology and Engineering, George Mason University. Fairfax, VA, USA, 2005.

[12] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Model-based verification of web service compositions. In *Proc. of the 18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 152–161, 2003.

[13] Andrew Hinton, Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: A tool for automatic verification of probabilistic systems. In *Proc. of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444, 2006.

[14] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming bpel to petri nets. In *Proc. of the 3rd International Conference on Business Process Management (BPM)*, volume 3649 of *Lecture Notes in Computer Science*, pages 220–235. Springer, 2005.

[15] Babak Khosravifar, Jamal Bentahar, Ahmad Moazin, and Philippe Thiran. Analyzing communities of web services using incentives. *International Journal of Web Services Research (IJWSR)*, 7(3):30–51, 2010.

[16] Marta Kwiatkowska, Gethin Norman, and Dave Parker. Prism: Probabilistic symbolic model checker. In *Proc. PAPM/PROBMIV'01 Tools Session*, pages 7–12, 2001.

[17] Marta Kwiatkowska, Gethin Norman, and Dave Parker. Prism: Probabilistic symbolic model checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Proc. TOOLS 2002*, volume 2324 of *Lecture Notes in Computer Science*, 2002.

[18] Rujuan Liu, Changjun Hu, and ChongChong Zhao. Model checking for web service flow based on annotated OWL-S. In *Proc. of the 9th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 741–746, 2008.

[19] Alessio Lomuscio and Monika Solanki. Mapping OWL-S processes to multi-agent systems: A verification-oriented approach. In *Proc. of the International Conference on Advanced Information Networking and Applications Workshops (AINA)*, pages 488–493, 2009.

[20] Zhongdan Ma. Modelling with prism of intelligent system. Master's thesis, University of Oxford, 2008.

[21] Srini Narayanan and Sheila A. McIlraith. Simulation, verification and automated composition of web services. In *Proc. of the 11th International Conference on World Wide Web (WWW)*, pages 77–88, 2002.

[22] Giti Oghabi, Jamal Bentahar, and Abdelghani Benharref. On the verification of behavioral and probabilistic web services using transformation. In *Proc. of the 9th IEEE International Conference on Web Services (ICWS)*, Washington DC, USA, July 04 - 09 2011. IEEE press.

[23] Giti Oghabi, Jamal Bentahar, and Abdelghani Benharref. Verifiying web services using probabilistic model checking via markov chains and processes. In *Proc. of the 10th International Conference on Software Methodologies, Tools and Techniques (SoMet)*, Saint Petersburg, Russia, September 28-30 2011. IOS Press.

[24] Michael P. Papazoglou. *Web Services: Principles and Technology*. Prentice Hall, 2008.

[25] Yuji Sakata, Kazutoshi Yokoyama, and Shigeyuki Matsuda. A method for composing process of non-deterministic web services. In *Proc. of the 2004 IEEE International Conference on Web Services (ICWS)*, pages 436–443, 2004.

[26] Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. The semantic web revisited. *IEEE Intelligent Systems*, 21(3):96–101, 2006.

[27] Jay ven Eman. Owl exports from a full thesaurus. *Bulltein of the American Society for Information Science and Technology*, 32(1):22–26, October/November 2005.

[28] Haibo Zhao and Prashant Doshi. A hierarchical framework for composing nested web processes. In *Proc. of the 4th International Conference on Service-Oriented Computing (ICSOC)*, volume 4294 of *LNCS*, pages 116–128, 2006.