Using Goals to Infer Formal Behavioral Models from Web Services Applications

May Haidar, Hicham H. Hallal

***Abstract*—** We propose an integrated framework for the formal analysis of web services based applications involving both static and dynamic analysis techniques. The proposed framework consists of three main components: A library of patterns, representing both recommended and undesired services related properties along with an efficient classification of the compiled patterns according to their effect on the behavior of web services based applications; a set of static analysis techniques that use tools like code inspection, abstraction extraction, and model inference to detect property patterns in the source code of the applications under test; and a set of dynamic analysis techniques directly mainly to verify web services based applications against property patterns that cannot be detected using only the source code of applications. In this paper, we elaborate the work completed on the development of an approach to dynamic analysis of web services based applications, we describe the formal model used to depict the behavior of an application based on it observed execution traces, and we outline the workflow of a goal based inference approach to derive behavioral models.

***Keywords*—**Dynamic Analysis of Web Services, Goal based reengineering, Inference of Behavioral models, Automata Models, Property Patterns, Education.

I. INTRODUCTION

BUSINESSES are increasingly adopting service orientation to shape the architecture of their enterprise solutions and to increase the efficiency of their software applications. At the foundation of this ever more popular paradigm, web services are heavily used to enhance decentralization, platform independence, and language portability. The power of services resides mainly in the high degree of dynamism and flexibility they exhibit throughout their lifecycle: publication, discovery, and binding are all dynamic activities that make a service an evolving entity capable of adapting to continuously changing and new requirements. In addition, compositions of services, which can also be dynamic, have added to the power of services in building larger enterprise solutions for heterogeneous businesses. Examples of such uses of service computing include e-commerce and education, where universities try to take advantage of available web services and cloud based applications to enable their communities to perform business and academic activities and projects.

May Haidar is with the Computer Science Department at Fahad Bin Sultan University, Tabuk, KSA, P.O. Box 15700, 71454. Tel: 00966551827096; (e-mail: mhaidar@ fbsu.edu.sa). She is also with the department of Computer science and Operational Research at the University of Montreal.

Hicham H. Hallal, is with the Department of Electrical Engineering, at Fahad Bin Sultan University, Tabuk, KSA, P.O. Box 15700, 71454 (e-mail: hhallal@fbsu.edu.sa).

However, the fast paced growth of service implementation and deployment in various contexts has resulted in a growing gap between the development and verification of services based applications.

On one hand, static analysis techniques [8, 11, 34] remain insufficient to detect behavioral flaws and defects that are exhibited only when services, especially composite ones, are executed. In particular, such techniques face two major problems: difficulty of generating executable models that can be used in the analysis, and limited coverage of defects that are exhibited only during runtime, e.g., concurrency incurred problems. On the other hand, dynamic and runtime techniques, which depend mainly on monitoring, can only claim to detect errors and flaws in the observable behavior of an application featuring the running of a service of the dynamic composition of several services.

Currently, formal methods have become a reliable solution to automate the analysis of various systems. In particular, formal techniques are being increasingly used to perform different development activities such as requirement definition and elucidation, modeling and model transformation, testing, and property verification [14, 15, 17]. Nowadays, formal verification techniques are used in several domains including communication systems [15], software and program analysis [12], and web based applications [8, 16].

As an example, model checking [6], which is usually used to verify the model of a concurrent system against formally specified properties can be fully automatic and produces counterexamples that point to the violations when a model does not satisfy a given property.

Historically though, the adoption of model checking based techniques on large scales remained relatively limited due mainly to problems like the lack of formal models, the inherent state space explosion problem, and the lack of proper justification for its use especially for classes of properties whose verification does not explore concurrent behavior of the models [6, 14]. However, the recent extensive work on model driven techniques in the development and analysis of systems coupled with the advances realized in the manufacturing of powerful computing devices have contributed to significant alleviation of the historic limitations and made the use of model checking in the verification of distributed applications both practical and justifiable.

In the case of composite Web Services, the reasoning about the use of model checking is similar. While analyzing simple web services does not necessarily require the use of model checking techniques, the use of model checking in the analysis of web services featuring underlying dynamically composite services is clearly needed and justified. The latter is specifically true for services whose composition is specified through WS-BPEL [28] (Web Services Business Process Execution Language) and WSCI [4] (Web Services Choreography Interface) As to the lack of models, especially in the case of inaccessible code, the focus concentrates on inferring behavioral models from observable traces that an application/system produces when it is used. Once a model is available, model checking can be used to verify the model of the application under test against predefined properties. When the specified properties do not require the use of model checking, other less complex techniques like search based methods or even manual inspection are applied to analyze the inferred models.

In this paper, we discuss the development of an integrated formal framework where both static and dynamic analysis techniques complement each other in enhancing the property testing process of an existing web services based application. In particular, we elaborate the work completed on the development of an approach to dynamic analysis of web services based applications, we describe the formal model used to depict the behavior of an application based on it observed execution traces, and we outline the workflow of a prototype toolset to implement the proposed approach along with applicable optimizations.

This paper is organized as follows. Section 2 presents a review of the basic notions and concepts in service computing. Section 3 discusses the formal framework for the analysis of web services based applications. Section 4 focuses on the approach to infer behavioral models of WS applications from executions. In section 5, we discuss the related work in the area of formal analysis of WS applications. Finally, in Section 6, we conclude the paper and discuss potential extensions of this work.

II. Service Computing

Service computing views business entities as service providers and models business processes as compositions of multiple services. Service-Oriented Architecture (SOA) and Web services are two related concepts in Service computing. The SOA/WS triangle represents the common principle of SOA and Web services are depicted in Fig. 1. The service providers first publish their service descriptions into a registration server. The service requestors can look up the registration to choose suitable services. Then the service requestors can directly interact with the service providers by sending messages to them.

A Web service is defined by the W3C as a software system designed to support interoperable machine-to-machine interaction over a network [32]. It has an interface described in a machine-friendly format (mainly WSDL). Other systems (including similar services) interact with the Web service in a manner prescribed in its description using SOAP-messages (Fig. 3), typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

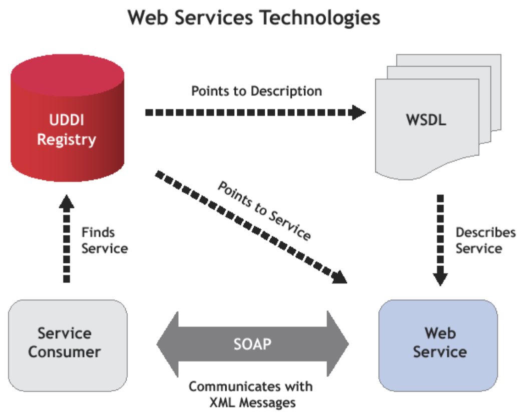


Fig. 1 The SOA/WS triangle

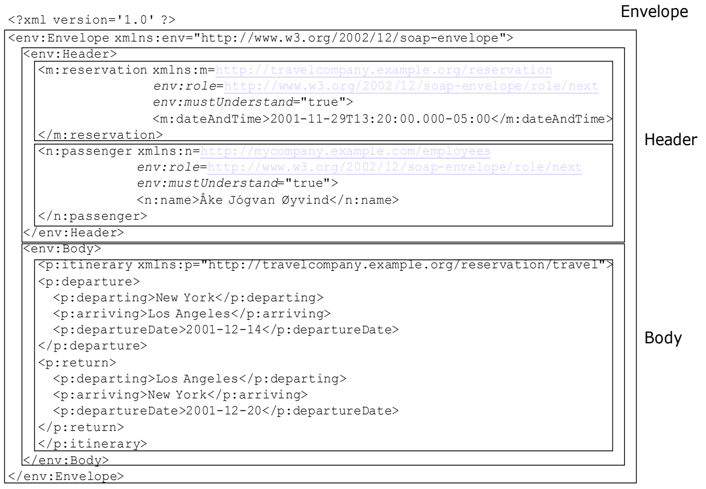
The W3C Web service technical stack in Fig. 2 lists some of the supporting techniques and specifications for Web services. The communication layer normally uses Internet protocols, such as HTTP, SMTP, and FTP. A given message may even involve multiple kinds of message transport. WSDL is exploited at the interface layer to indicate the end point of a service and to describe the operations of a service and the types and number of parameters for invoking an operation. UDDI [28] is a service discovery protocol that supports service registration and look-up.

Publication and Discovery: **UDDI**	extends URI
Service Description: **WSDL**	extends HTML
Messaging: **SOAP**	extends HTTP
Transport: HTTP, SMTP, FTTP, …	

Fig. 2. The technical stack for Web services

Web service processes are business processes composed by individual Web services. W3C and OASIS have released several XML-based description languages to model Web service processes. These languages can be classified into orchestration and choreography languages. Orchestration languages hold the point of view of a service requestor (c.f. WS-BPEL[27]). They model how the requestor calls external services and how to process the response internally. In another word, orchestration languages model the internal behavior within a service requestor. Complementarily, choreography languages model how the services interact with each other outside of the services. The behavior of a Web service is strictly bounded in the sense that it is owned and accessible within the corporation, but hidden from any other corporation. Therefore, the internal behavior of a Web service is not observable for another, except through its emitted messages and communication ports. The choreography languages describe the observable behaviors among a group of Web services. It can be from the individual service point of view that the global model is projected on a single service (c.f. WSCI [4]), or from the global system point of view (c.f. WS-CDL [24]). Both choreography and orchestration languages describe the data flow and the control flow of a business process. Control flow expresses the execution order of the actions in constructs of sequence, branching, parallel, synchronization etc. Data flow is about process relevant data and how these data are interchanged and manipulated by the actions. The two flows are intertwined such that the control flow is triggered by certain status of data and data is manipulated by the actions defined in the control flow.

As a particular type of Web services, we consider the REST Web services, which have become one of the most important technologies for Web applications [14, 29]. REST, which stands for Representational State Transfer, represents an architectural style for networked hypermedia applications. It is primarily used to build Web services that are lightweight, maintainable, and scalable.



```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/12/soap-envelope">
 <env:Header>
  <m:reservation xmlns:m=http://travelcompany.example.org/reservation
                 env:role=http://www.w3.org/2002/12/soap-envelope/role/next
                 env:mustUnderstand="true">
                 <m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n=http://mycompany.example.com/employees
               env:role=http://www.w3.org/2002/12/soap-envelope/role/next
               env:mustUnderstand="true">
               <n:name>Åke Jógvan Øyvind</n:name>
  </n:passenger>
 </env:Header>
 <env:Body>
  <p:itinerary xmlns:p="http://travelcompany.example.org/reservation/travel">
  <p:departure>
    <p:departing>New York</p:departing>
    <p:arriving>Los Angeles</p:arriving>
    <p:departureDate>2001-12-14</p:departureDate>
  </p:departure>
  <p:return>
    <p:departing>Los Angeles</p:departing>
    <p:arriving>New York</p:arriving>
    <p:departureDate>2001-12-20</p:departureDate>
  </p:return>
  </p:itinerary>
 </env:Body>
</env:Envelope>
```

Fig. 3 Example of a SOAP message.

The architecture of REST based application follows the client/server model, where the communication between the components of the application uses mainly stateless HTTP as the underlying protocol. In the REST architectural style, data and functionality are considered resources accessible using Uniform Resource Identifiers (URIs), typically links on the Web (Fig. 4). Clients and servers exchange representations of resources by using a standardized interface and protocol [14]. Services based on REST are called a RESTful services and are bound by major constrains such as the uniform interface, which induces desirable properties including performance, scalability, and modifiability. Such properties enable services to work best on the Web.

III. Formal Framework for the Analysis of Web Services Based Applications

In this section, we discuss the formal framework proposed to enhance the hybrid (static and dynamic) analysis of web services based applications. The intended framework encloses three main components that are deemed essential to automation in the field of formal analysis (verification, validation, or testing) of software applications.

A. *Library of Property Patterns*

Patterns are commonly used in the development and analysis of software applications, and service oriented architectures as well, since they introduce clever and insightful ways to solve common problems. Along with patterns, the term antipattern is also defined as the solution to a problem that does not work as intended (in terms of correctness and/or efficiency) [12].

```
HTTP Request
POST http://MyService/Person/
Host: MyService
Content-Type: text/xml; charset=utf-8
Content-Length: 123
<?xml version="1.0" encoding="utf-8"?>
<Person>
  <ID>1</ID>
  <Name>M Vaqqas</Name>
  <Email>m.vaqqas@gmail.com</Email>
  <Country>India</Country>
</Person>
```

```
HTTP Response
HTTP/1.1 200 OK
Date: Sat, 23 Aug 2014 18:31:04 GMT
Server: Apache/2
Last-Modified: Wed, 01 Sep 2004
13:24:52 GMT
Accept-Ranges: bytes
Content-Length: 32859
Cache-Control: max-age=21600, must-
revalidate
Expires: Sun, 24 Aug 2014 00:31:04
GMT
Content-Type: text/html;
charset=iso-8859-1
<!DOCTYPE html PUBLIC "-//W3C//DTD
XHTML 1.0 Strict//
EN"
"http://www.w3.org/TR/xhtml1/DTD/xht
ml1-strict.dtd">
<html
xmlns='http://www.w3.org/1999/xhtml'
>
<head><title>Hypertext Transfer
Protocol -- HTTP/1.1</title></head>
<body>
...
```

Fig. 4. Request/Response Pair of a RESTful Application.

Following their definition, existing works [12] have documented antipatterns in catalogs (similar to design patterns) so that they can be avoided. In the proposed framework, we intend to build on existing work in [12, 17, 18, 19] and compile a library of web services properties (patterns and antipatterns) along with a classification that can make the analysis of an application a more structural process. The classification of properties will be hierarchical following the categorization:

1) static/dynamic,
2) correctness/functional, and
3) style/performance.

Such classification should help developers identify the antipatterns to better avoid them, and testers detect them in the application using the appropriate techniques. On the other hand, documented properties, which would include BPEL4WS and WISCI requirements in the form of property patterns, can be instantiated in different contexts and for different purposes like verifying correctness, security, and performance related issues. The property library will be based on an easy to use template that depicts mainly the type, formal model, and example of a property.

For example, in a previous work [17,18], a pattern template is defined and a set of 119 patterns and property specifications are identified for the verification of Web applications (WAs). Fig. 4 shows an example of such patterns. Each pattern is specified in Linear Temporal Logic (LTL), which makes it directly usable in many model checkers.

ID	FGS6
Pattern description	Banking information is entered no more than once before submitting form
Category	Functional – General – Security and Authentication
Page Attributes	*Banking_info*: Boolean identifying the presence of fields for banking information *Submit*: identification of page where form submit action exists
LTL Mapping	PrecedenceGlobally (($\neg(\mathit{banking\_info})\,\mathrm{W}\,(\mathit{banking\_info}\,\mathrm{W}\,(\mathrm{G}\neg(\mathit{banking\_info})))), \mathit{submit})$
Comments	
Source	Newly introduced

Fig. 5 Example of a Web Applications Pattern

B. *Static Analysis Techniques*

The proposed techniques target mainly code and/or existing specifications or textual descriptions of web services (choreographies and orchestrations). Such techniques are usually independent of specific input data sets or individual execution paths, and are classified into:

a) Direct code inspection techniques, where suspicious code segments are directly identified in the code (through linear scanning for example).

b) Abstraction based techniques, where code representations e.g., class diagrams are used to detect the exhibition of certain predefined patterns (or antipatterns).

c) Model based techniques, where a model is extracted from the code of an application to describe the expected behavior of the application during runtime.

In the case of web services based applications; static analysis techniques would be applied to the available documents containing the descriptions of individual and composite services. In doing this, we follow in the steps of the work in [12]; the main deviation being the customization of the antipattern library developed to handle mutlithreaded Java applications to the context of web services and web services compositions. In addition, the library will be extended to cover patterns/antipatterns like the one shown in Fig. 5. However, some complex faults cannot be detected with static analysis approaches or only at a high cost (like deadlocks and other errors that cannot be exhibited except when exploring the concurrent behavior of the application). Moreover, static analysis techniques are prone to producing significant numbers of false warnings (mainly false positives) while not being able to detect some behavioral errors like in the case of exception handling. This justifies the need for the third component, a set of dynamic analysis techniques.

C. *Dynamic Analysis Techniques*

Dynamic analysis techniques do not necessarily rely on existing specifications or textual descriptions of an application under test. Instead, they are applied to executable behavioral models that are derived from the application's observed executions (traces or logfiles). Such approach to analysis is particularly efficient in the case of web services based applications; often characterized by their readiness to compose web services, especially dynamically. Moreover, such applications usually feature large architectural structures of applications, which make writing complete specifications inefficient and rather impractical, along with high degrees of concurrency in the behavior of the composite applications. Dynamic analysis techniques include extracting behavioral models of applications from observed executions and verifying them (mainly using exhaustive simulation like in model checking) against behavioral properties specifying defects that cannot be detected using static analysis techniques. Existing dynamic approaches are of two types:

1) Offline (postmortem), where recorded executions of an application are stored and later used in modeling and verifying the application under test.

2) Online (runtime), where an application under test is analyzed in real time as the executions are generated.

In this work, we focus on an offline dynamic analysis approach, which we elaborate in the following section. Nevertheless, the readiness of the proposed framework to handle online analysis of applications is guaranteed given several existing solutions including adopting a sliding window approach that involves taking snapshots of the application during consecutive time periods and building different models for each period so that verification is performed on the built models; or an incremental approach to constructing the behavioral model of the application so that verification can be performed on the model after each iteration.

It is also important to note that the dynamic approach in this paper relies on model checking, where models are derived from the observed behavior of the application. Thus, the approach could be seen as passive testing. Since results of verification could be compromised when a WSUT does not meet the assumptions described previously, this approach does not eliminate the need for traditional testing and should be considered as a complimentary activity rather than an alternative. For instance, a potential enhancement of the approach consists of testing the application using test cases derived from the model checking counter-examples. This helps ensure verifying whether properties are indeed violated. Also, behavioral models derived by this approach enable model based test generation [14].

Consequently, the approach proposed in this paper is based on the use of model checking to test user-defined properties of applications built using web services compositions whose source codes are inaccessible. The model of an application under test is obtained from traces of the web services execution while properties of interest relate to both the business logic and ergonomics of the web services. More specifically, the proposed approach breaks down into the following main steps:

1) Modeling the Web services composition in a language acceptable by a chosen model checker. As described earlier, we use the execution traces of the web services composition recorded using an appropriate monitoring tool, e.g., a proxy server that is capable of intercepting HTTP and SOAP communications. The traces are then converted into a communicating automata model representing the behavior of all the components of the web services based application. This decision to use an automata based model goes in line with the choice of the model checker Spin [21, 22] as the verification core of the proposed approach. Spin is an open source model checker that has been used for verification of systems on both the design and implementation levels. The language used in Spin is Promela, which is a C-like high level programming language used to describe executable models depicted as finite state automata.
2) Specifying properties of interest. These properties can represent both desired and undesired behaviors of the web services. Properties will be mainly user defined and expressed in the property specification language of Spin, LTL. The use of the Spin model checker provides an added flexibility to specify properties for verification. Spin supports writing properties in Linear Temporal Logic (LTL). Our approach consists of providing LTL formal representations of the patterns in the library of property patterns described in Section 2.1.
3) Checking the obtained model against the given properties. To do so, Spin computes the composition of all the component automata in the derived model and builds a graph containing the global states of the application. The graph is then intersected against the language of a property for containment. The details of the verification process using the Spin model checker can be checked in [21, 22].

In the following, we discuss the proposed approach to formally modeling web services based applications.

IV. Formal Modeling of Web Services based Applications

The purpose of building a formal model for a web service under test (WSUT) is to verify whether the service composition exhibits certain predefined properties using model checking techniques. It is assumed in this paper that the properties specified in a temporal logic of a chosen model checker are composed of atomic propositions and for each SOAP/HTTP service request, the value of each proposition is uniquely determined by the content of the service response. These propositions refer to attributes that are user defined and have to be checked (and of course reflected in a model). Attributes can be of various types, for instance: a numerical type to count the occurrences of a certain element, a string type to denote the domain name of a response. To build a formal model of a web service composition whose source code is accessible, one may use abstraction techniques developed in software reverse engineering following a the static, white box approach [17, 24] as described in the previous section. However, the source code is not always available, or access to the code could breach copyrights or trade secrets (especially when verification is performed by a third party). Moreover, a web service composition can be written using different languages and even different paradigms which make static analysis difficult to perform.

When the code is not available for modeling, one can build a formal model following a dynamic, black-box based approach, by executing the application and using only the observations of an external behavior of the service composition over a certain period of time. Verification of such models (resulting from finite trace of an application) is called run-time verification [15, 26]. In case of web services that rely on the SOAP or HTTP protocol considered in this work, an observable behavior consists of requests and responses, assuming that the flow of requests and responses between a client side and a server in the WSUT is observable. One possible way of achieving this is to use a proxy server [16]. A proxy server monitors the traffic between the client and the server and records it in proxy logs. The proxy logs, i.e., traces, contain the requests for composing services and the responses to these requests.

In the next section, we present our approach to derive automata based models from traces of web services.

A. *Workflow of the Approach*

Fig. 6 shows the workflow of the proposed approach. The main components are:

- Monitoring module. It intercepts SOAP/HTTP requests and responses during the navigation of the WSUT performed by the user/crawler.
- Analysis module. It takes the intercepted traces as input and generates an automata model in XML/Promela.
- Model checker, Spin. It verifies user defined properties against the generated model and produces a counterexample for each violated property.

With this approach, a behavior of a WSUT, called an execution session, aka Request/Response Sequence (RRS) [16], is interpreted as a possible sequence of web services responses intermittent with the corresponding requests. Usually, many of these requests are triggered by the user's actions (clicking links, submitting forms), while others can be triggered by the service itself.

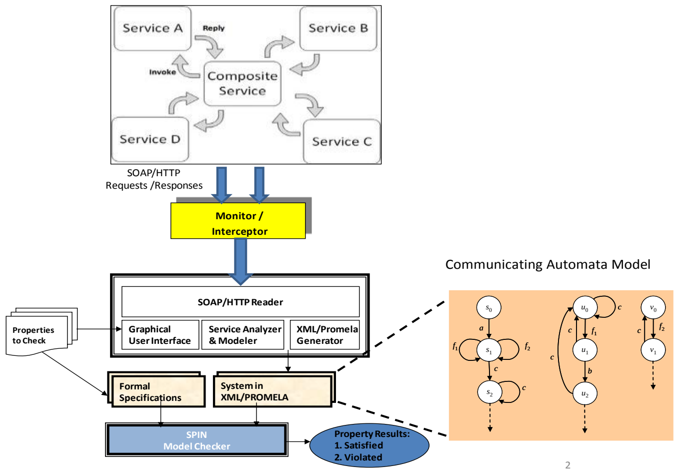


Fig. 6. Workflow of the approach

B. *The Parameterized Finite Automata Model*

Following [14], we adopt the definition of a parameterized finite automaton which extends the definition of a Finite Automaton by augmenting states with parameters and transitions with guards.

Definition 1 [13]. A PFA A over L, P, and D is a tuple $<\Sigma, Q, q_0, \rightarrow_A>$; where Σ is a finite set of actions; $Q \subseteq (L \times 2^P)$ a finite set of states, where each (parameterized) state $q = (l_q, P_q)$, $l_q \in L$ and $P_q \subseteq P$; $q_0 \in Q$ an initial state; and $\rightarrow_A$ a finite set of transitions in which a transition is a tuple $(q, (a, g_q), q')$ denoted $q \text{-}(a, g_q)\rightarrow_A q'$, where q and q' are states and (a, g_q) is a guarded action such that $a \in \Sigma$ and g_q is a predicate on P_q.

An *execution* of a PFA A is a finite sequence $q_0(a, g_0)q_1\ldots q_n$, where q_0 is the initial state, q_i is a state for $i = 1, \ldots n$ in Q, and each tuple $(q_i, (a_j, g_i), q_{i+1})$ is a transition in $\rightarrow_A$, where g_i is True. We denote the set of all the executions of A by $Ex(A)$. For each transition $q_i \text{-}(a_j, g_i)\rightarrow_A q_{i+1}$ of A, there can be more than one combination of parameter values that satisfy the guard g_i. Therefore, we consider two types of PFA executions:

- Symbolic, where the predicates in the transition guards are literals of the form: $v \leq p$, $p \leq v$ or $p \in X$, where $v \in Dp$, and $X \subseteq Dp$.
- Concrete, where the predicates in the transition guards are literals of the form: $p = v$, where $v \in Dp$.

This distinction is extended to PFAs as follows [14]:
A *symbolic* PFA is a PFA such that each guard on a transition is a DNF formula in which each conjunct is a literal g_p expressed as $v \leq p$, $p \leq v$, or $p \in X$, where $p \in P$; $v \in D_p$, and $X \subseteq D_p$.

A *concrete* PFA is a PFA such that each guard on a transition is a DNF formula in which each conjunct is a literal g_p expressed as $p = v$, where $p \in P$ and $v \in D_p$.

A concrete PFA can have only concrete executions while a symbolic PFA can have both symbolic and concrete executions.

Given two parameterized finite automata $A_1 = <\Sigma_1, Q_1, q_{01}, \rightarrow_{A1}>$ and $A_2 = <\Sigma_2, Q_2, q_{02}, \rightarrow_{A2},>$, A_1 and A_2 are said to be *compatible* if and only if $q_{01} = q_{02}$. The *merge* of two compatible PFAs A_1 and A_2, denoted $A_1 \sqcup A_2$, is defined as the parameterized finite automaton $<\Sigma, Q, q_0, \rightarrow>$, where $\Sigma = \Sigma_1 \cup \Sigma_2$, $Q = Q_1 \cup Q_2$, $q_0 = q_{01} = q_{02}$, and $\rightarrow$ is defined as follows:

- if q -(a, gq)$\rightarrow$A1 q′ and a $\notin$ □□□then q -(a, gq)$\rightarrow$ q′.
- if q -(a, gq)$\rightarrow$A2 q′ and a $\notin$ □□□then q -(a, gq)$\rightarrow$ q′.
- if q -(a, gq)$\rightarrow$A1 q′, q -(a, g′q)$\rightarrow$A2 q′ then q -(a, gq $\vee$ g′q)$\rightarrow$ q′.
- if q -(a, gq)$\rightarrow$A1 q′, q -(a, g′q)$\rightarrow$A2 q″, and q′ $\neq$ q″ then q -(a, gq)$\rightarrow$ q′ and q -(a, g′q)$\rightarrow$ q″.

The *intersection* of two compatible PFAs A_1 and A_2, is also defined and denoted as $A_1 \sqcap A_2$. It is a PFA $<\Sigma, Q, q_0, \rightarrow>$, where $\Sigma = \Sigma_1 \cap \Sigma_2$, $Q \subseteq Q_1 \cap Q_2$, $q_0 = q_{01} = q_{02}$, and $\rightarrow$ is defined as follows: if $q \text{-}(a, g_q)\rightarrow_{A1} q'$, $q \text{-}(a, g'_q)\rightarrow_{A2} q'$, such that $g_q \wedge g'_q \neq 0$ then $q \text{-}(a, g_q \wedge g'_q)\rightarrow q'$.

The merge and intersection operations are associative; they can be applied to finitely many PFAs. The merge (intersection) of n PFAs $A_1 \ldots A_n$ is a PFA $A = A_1 \sqcup \ldots \sqcup A_n$ ($A_1 \sqcap \ldots \sqcap A_n$) over the set of actions $\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_n$ ($\Sigma = \Sigma_1 \cap \ldots \cap \Sigma_n$). Merge and intersection apply to both symbolic and concrete PFAs, so the merge of two symbolic (concrete) PFAs is a symbolic (concrete) PFA.

Finally, the *implementation* relation between two PFAs A_1 and A_2 is defined as an execution inclusion relation: A_1 *implements* A_2, denoted $A_1 \sqsubseteq A_2$, iff for every execution E_1 in $Ex(A_1)$ there exists an execution E_2 in $Ex(A_2)$ such that for every transition q_1 -(a, g_{q1})$\rightarrow_{A1}$ q_1' in E_1 there exists q_2 -(a, g_{q2})$\rightarrow_{A2}$ q_2' in E_2 such that $g_{q1} \Rightarrow g_{q2}$. The implementation relation can be used to relate symbolic PFAs, concrete PFAs, and a concrete PFA to a symbolic PFA.

C. Modeling WS Execution Sessions Using PFA

We discuss how to infer a PFA model from traces collected by monitoring a WS. The proposed approach consists of the following steps:

1) Represent each session as a concrete PFA A_{Tr}
2) Merge the concrete PFAs of all sessions to form one concrete PFA A_{TC}.
3) Infer a symbolic PFA A_{TS} such that $A_{TC} \sqsubseteq A_{TS}$.

First, we describe how to represent a session of a single window non-framed HTTP application by a PFA. Apart from the default page displayed in the browser, the trace T of an application is a sequence <Rq_1, Rp_1>… <Rq_n, Rp_n>, where each <Rq_i, Rp_i> is a request response pair. From this sequence, we assume that the following sets can be determined: $L_T = \{URL_1, \ldots, URL_n\}$ the set of state labels, where each label URL_i is the URL of the page identified in the response Rp_i in the trace; $P_T = \{p_1, \ldots, p_m\}$ the set of parameters, where each parameter represents a page attribute appearing in a response Rp_i; and $D_T = \{D_1, \ldots, D_p\}$ the set of parameter domains, where each domain D_i includes the values recorded for a parameter p_i in the trace. Consequently, knowing L_T, P_T, and D_T, we use the definitions of PFA and concrete PFA to represent T by a concrete trace PFA $A_T = \langle \Sigma_T, Q_T, q_{0T}, \rightarrow_T \rangle$, where

- $\Sigma_T = \{Rq_1, \ldots, Rq_n\}$ is the set of actions. Each Rq_i is either a link clicked or a form filled on the previous page represented by Rp_{i-1} except for Rq_1 which is equal to the URL of the home page of the application.
- $Q_T = \{q_1, \ldots, q_n\}$ is the set of states. For $i > 0$, each parameterized state $q_i = (l_i, P_i)$, where $l_i \in L_T$, and $P_i \subseteq P_T$ the collection of parameters of the page returned in the corresponding Rp_i.
- $q_{T0} = q_1$ is the initial state.
- $\rightarrow_T$ is the transition relation, a set of tuples (q_i, (Rq_{i+1}, g_{qi}), q_{i+1}), where q_i and q_{i+1} are states in Q_T and (Rq_{i+1}, g_{qi}) is the guarded action such that g_{qi} is a DNF consisting of one conjunction on the elements of P_i, and each conjunct in g_{qi} is a literal g_p expressed as $p_j = v_j$, where $p_j \in P_T$ and $v_j \in D_j$.

By construction, the set $Ex(A_T)$ contains one execution (the trace itself). Therefore, it could be argued that T be mapped into a concrete execution of a PFA rather than a concrete PFA. We prefer mapping traces into automata rather than executions since it makes operations such as merge and intersection directly applicable without any need to adapt them to executions.

Next, we infer a PFA model of a WS from a collection of execution sessions. We do so by merging the concrete trace PFAs representing the collected sessions. The resulting PFA has a set of states that represent all the pages of the application visited in all the traces, a set of transitions that contains all the transitions of the individual PFAs, and, in particular, a set of executions that includes all their executions. Formally, this can be stated as follows: Given a collection of m concrete PFAs $A_1 = \langle \Sigma_1, Q_1, q_{01}, \rightarrow_1 \rangle \ldots A_m = \langle \Sigma_m, Q_m, q_{0m}, \rightarrow_m \rangle$ that represent m traces collected from the same WS and are compatible since all sessions are recorded by browsing the application starting from its home page, the merge of the PFAs is a PFA denoted $A_C = \langle \Sigma_C, Q_C, q_{0C}, \rightarrow_C \rangle$ such that

- $\Sigma_C = \Sigma_1 \cup \ldots \cup \Sigma_m =$, where $\Sigma_i = \{Rq_{1i}, \ldots, Rq_{ni}\}$ is the set of actions representing all the links clicked and forms filled in the trace T_i;
- $Q_C = Q_1 \cup \ldots \cup Q_m$ the set of states representing all the pages visited in all traces, where each state $q_i = (l_i, P_i = P_{i1} \cup \ldots \cup P_{im})$;
- $q_{0C} = q_{01} = q_{0m} = (\mathrm{URL}_1, P_1)$, where URL$_1$ and P_1 are the URL of the home page of the application and its set of parameters, respectively;
- $\rightarrow_C = \rightarrow_1 \cup \ldots \cup \rightarrow_m$ is the transition relation, where each transition is a tuple (q_i, (Rq_{i+1}, g_{qi}), q_{i+1}): q_i and q_{i+1} are in Q_C, and (Rq_{i+1}, g_{qi}) is the guarded action, where g_{qi} is a DNF in which a conjunct is a literal $g_{\{p\}}$ such that $p = v$, where $p \in P_i$ and $v \in D_p$.

Notice that the PFA A_C represents more behavior than the collection of the individual concrete session PFAs. In order to avoid redundant application of merge for repeated sessions, a check can be made whether a new concrete session PFA implements the existing PFA of a collection of sessions, i.e., the behavior recorded in the new session is already modeled. Then, merge is applied only to session PFAs that violate the implementation relation.

Finally, we want to find a symbolic PFA that represents the same behavior modeled in the PFA obtained by the merge operation. This problem can be stated as follows: Given a concrete PFA A_C, obtained by merging a collection of concrete session PFAs, infer a symbolic PFA A_S such that $A_C \sqsubseteq A_S$.

The implementation of this step follows the approach detailed in [14], where the objective becomes to transform a set of equalities on a parameter (in the concrete PFA) into a single inequality, an interval or an enumeration on the same parameter (the case of the symbolic PFA). The works in [14] adopt the C4.5 data mining algorithm to infer the desired symbolic PFA through deducing classification rules in the form of decision trees using the concept of Information Gain [14]. However, in this work, we introduce the following improvement on the method adopted in [13] to cope with large models that can result from exploring the web services based applications by many users. We consider an approach to reengineer customized behavioral models of WS applications based on the actual executions of an application while being accessed by real users or by testers. The reengineered models are customized to depict the intentions of the users of a WS application. By default, each user accesses a WS application to fulfill a specific task: purchase a ticket, book a reservation, buy food, execute a banking transaction, etc. The main idea in the proposed approach is to reengineer models that depict the different intentions (goals) of users who interact with the WS application over a period of time. The behavior of the WS application in response to user stimuli is still collected in execution sessions observable through monitoring. Then, for each predefined goal a model is reengineered that includes the behavior recorded only in the traces that satisfy the goal.

D. Goals of WS based Applications

In general, a user interacts with a WS application with a purpose in mind. It basically depends on the type of the application and the functionality it offers. For example, we consider an example WS application which consists of a small flight reservation system [13, 14]. The customers can use the application to buy tickets and make reservations with different preferences. Hence, a user accessing the flight reservation application would, most probably, want to buy an airplane ticket to travel from one place to another. In this case, the purpose is to "buy a ticket". This can be identified as the goal of the user in his access to the WS application. By default, achieving the mentioned goal involves completing smaller tasks before actually "buying the ticket" through a confirmation issued by the WBA in the form of receipt, SMS message to a mobile phone number or through an email message. These smaller tasks might involve entering personal information of the passenger for whom the ticket is being bought, source destination information, financial and credit card information, and finally consent for purchase and payment. This means the bigger goal of "buying the ticket" is broken down into smaller goals that are not necessarily an expression of the functionality of the WS application.

This reasoning about goal definition and classification is similar to the work in [13] where the intentions of the user of a web application are classified into two types:

1) Action intentions, which are perceived on a low level. Each action can be a mouse click, keyboard typing, or any other basic action performed on a computer.

2) Semantic intentions, which correspond to what the user wants to achieve at high level. A semantic intention may involve several basic actions on a computer to accomplish it.

Our reasoning is also similar to the reasoning made in the field of automated planning, where hierarchical decomposition of goals is considered to devise and implement proper plans especially in the presence of contingencies.

We consider the following classification of goals in a WS application:

1) Non functional goals: They relate to completing low level tasks in the WS application. The completed tasks do not need to satisfy a functional requirement of the application. They include tasks like filling personal information on a page, entering login information, navigating from one page to another using various controls (buttons, links, form submissions, etc).
2) Functional goals: They relate directly to satisfying a functional requirement of the WS application. Examples include buying a ticket, reserving a hotel room, buying a book, etc. Each functional goal is achieved through the completion of at least one non functional goal. In other words, each functional goal can be broken down into a sequence of one or more non functional goals that should be achieved in a certain order (usually defined by the developers of the application).

In this paper, we focus on functional goals and describe how to use them in reengineering customized models of the WS application with respect to the various goals that can be achieved when using the application. We describe in the following section the model checking based approach where execution sessions from a WS application that satisfy a specific pre-defined goal can be used to infer a behavioral model of the application.

E. Goal based Modeling of WS Applications

In this section, we describe the goal based modeling of WS applications. The proposed approach is an extension of the inference approach presented in [14]. Fig. 7 shows the goal-based approach, i.e., modeling is based on knowing the goals of the users of the application under test. A new step in the workflow is added to filter out sessions that do not satisfy the desired goal.

Each recorded trace of the WS application is checked against the representation of a goal for satisfaction. The check is performed in the model checker Spin [21], where the trace is modeled by a PROMELA process and the desired functional goal is specified using the Linear Temporal Logic (LTL) formalism. Notice that since the trace of a WS application is a sequence of pages interleaved with HTTP/SOAP requests, the check for goal satisfaction can be performed more easily through a simple search to match the goal. However, we choose to use model checking in order to keep the approach more generic and capable to treat more sophisticated traces where a total order between components of a trace is not always present. In some cases, such as in [15], the traces collected from a system under test can be partially ordered sets of events and the simple search to match the goal becomes insufficient.

F. Goal Specification

LTL is the main language for property specification in the model checker Spin [21]. Other forms of specification like automata (never claims) are possible but not considered in this work. Following the discussion in Section 4.4, we consider that each functional goal is a sequence of smaller non-functional ones.

As an example, consider the function goal "purchase a ticket for a minor" that is adopted with minor modifications from [14]. Such goal involves several steps including:

1) Providing the name of the passenger
2) Providing the age
3) Providing name of the guardian traveling with the child
4) Providing the source and destination information
5) Providing seating preferences
6) Providing payment information
7) Confirming purchase

However, formulation of the goal does not require an LTL formula that involves all the non-functional steps. Instead, one can choose key steps to use as indicators of the goal. For example, "***purchasing a ticket for a minor***" must always involve the two non-functional goals: "*providing guardian information*" and "*confirming the purchase*". For simplicity, we denote the functional goal *A* and the two non-functional goals *b* (for providing guardian information) and *c* (for confirming the purchase). In addition to the decomposition of *A* into *b* and *c*, we know that *b* must always come before *c*. In LTL, this specification of *A* can be expressed in several forms. For illustration purposes, we show one of the simplest forms:

!c U b, which reads: NOT c UNTIL b.

This means that *c* does not happen until after *b*.

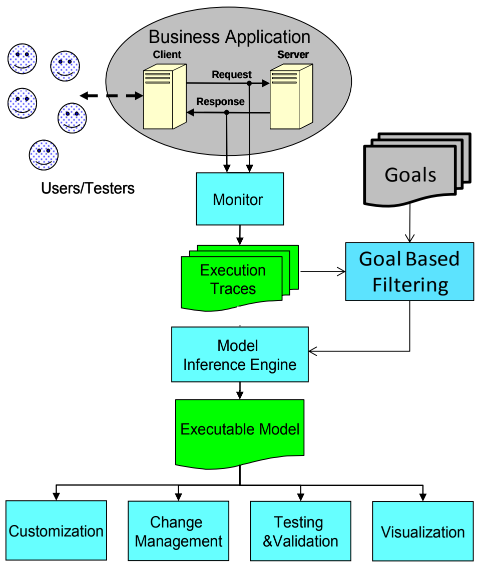


Fig. 7. Modified model inference approach.

On the other hand, a single session that features purchasing a ticket for a child is visualized in Fig. 8. The session is represented as an automaton (concrete PFA), where states represent the pages visited in the WS application to fulfill the goal and the transitions are the transitions between the pages. After selecting the session to be added to the model, the existing framework for model inference [14] is used to deduce a model based on the filtered trace and any traces chosen previously.

Fig. 9 shows the model of the WS application corresponding to the goal *A* generated from 200 traces. The presented model includes, in addition to states and transition, the conditions on the data submitted to the WS application in order to reach the goal. Notice that even though some customers were interested in buying a ticket for a child, they entered wrong information that led them into rejection. This is due to the formulation of the goal itself, which states the confirmation has to come after setting a guardian while rejection is reached directly from reservation. This example shows that proper definition of the goals is the key to obtain the optimal model for the WS application under test. On the other hand, the presented model shows how the behavior of the application from all the processed sessions (in this case 200 traces) is aggregated in a single state diagram. While some traces would contribute new states and transitions to the model, other traces might contribute only new conditions on the transitions between states.

Run time verification of webbed, and web service-based, applications have gained a lot of attention in many research activities both in academia and in the industry given the role such applications have in the shaping of today's economy based on e-commerce and e-services. This work can be evaluated in the context of enhancing existing solutions to address the problem of applying formal analysis to web services based applications.

V. Related Work

This research is closely related to the work published in [14, 16, 17], where we have designed and implemented various approaches to address the problem of modeling and analyzing web applications. From this viewpoint, the current work can be seen as extension of the previous works in [13, 14, 15, 16] to cover the web services application domain. The proposed extension is intended in the form of a formal framework that integrates static analysis techniques and dynamic analysis techniques along with a library of property patterns, relevant to web services. The proposed library of patterns builds upon the previous work in [12] and [18] with the intention to extend the existing catalogs with properties related to the correctness, style, and performance of web services based applications.

The proposed framework features a set of static analysis techniques inspired by the work in [14], where multithreaded Java applications are analyzed. The intended work on static analysis involves adaptation of techniques like linear scanning (code inspection), abstraction extraction and model inference to the specifics of web services based applications.

The main contribution in this paper is the proposal to build an automated approach to dynamic analysis of web services based applications relying on modeling actual behavior of such applications, formulating relevant properties to be verified against the derived models, and using an existing model checker to verify the application against the property specifications. In recent years, a large body of research has been produced with a focus on formal modeling of web services based applications in order to induce automation in the analysis of the developed applications against some predefined properties specified from the description and requirements texts.

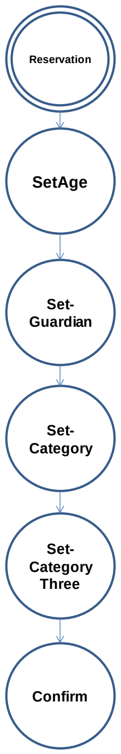


Fig. 8. Sample execution session for ticket purchase.

Derived models are often generated from textual descriptions of applications (BPEL, BPEL4WS, and WSCI), and can be used mainly to check static properties that relate to the structure and content of the application, usually described as a composition of services. Examples of such research include the work of Foster et al. [8, 9], which models BPEL descriptions as Finite State Process models, which can be verified against properties that are mainly derived from design specifications written in UML notations like the Message Sequence Chart (MSC) or activity diagrams.

Properties sought for verification include mostly semantic failures and difficulties in providing necessary compensation handling sequences that are tough to detect directly in common workflow languages like BPEL. Other attempts have been described in the literature as well including the work of Breugel and Koshkina [26, 31] who introduce the BPE-calculus to capture control flow in BPEL descriptions and programs.

The service descriptions in the proposed language allow for checking against properties like dead path elimination and control cycles. The verification, mainly formal model checking, is performed in the toolset Concurrency Workbench (CWB). However, as discussed in Section 3, proposed verification approaches based mainly on the static analysis of an existing source code, where different types of models like EFA, Promela, and communicating FSMs [9, 16, 26] are used, have their limitations and impracticalities. Consequently, more efforts are being spent on performing run-time verification of web service applications based on monitoring and model extraction.

Also, [24] address the run-time monitoring of functional characteristics of composed Web services, as well as for individual services [27].

Meanwhile, inferring behavioral models of software applications has been the focus of many research efforts over decades, e.g., [1, 4, 7, 10, 21] where models are either inferred mainly from system requirements [18, 15, 16], depicted as scenarios, or extracted from execution traces [7, 9, 18, 19] collected by monitoring. The approach presented in this paper can be compared to the work in [9], [11], and [32]. In [13], a method is proposed to learn HTTP request models for intrusion detection, where the signatures of known attacks are used in enhancing the learning. On the other hand, in [15], a trace recorded during a browsing session is used to infer a model of a web application.

The obtained model in [16] consists of communicating automata representing windows and frames of the application, thus resulting in a hierarchical model that describes the control flow of the application, but does not address the data variations that are revealed by traces collected in different browsing sessions. In [32], the focus is mainly on predicting simple low level intentions of users of applications based on the features extracted from the user interaction such as user’s typed sentences and viewed content. The work does not consider high level goals and structured intentions that relate more to the functionality of the application.

This work builds on the results obtained in [13] and [14] in the sense that we reuse the formal framework for model inference, which is capable of inferring models that depict both the data and control flows of a WBA. The implementation of the framework was completed using data mining algorithms applied to random sets of traces generated by actual use of the WBA. Here, we follow [13] and filter execution sessions of WS applications before using them based on satisfying a pre-defined goal specified as a property tested on the recorded execution session. If the session satisfies the goal, then it is added to the model. Otherwise, it is ignored. The objective is to customize the model and reduce its size to make it more useful in automation of specific tasks like property testing, test derivation and test case generation.

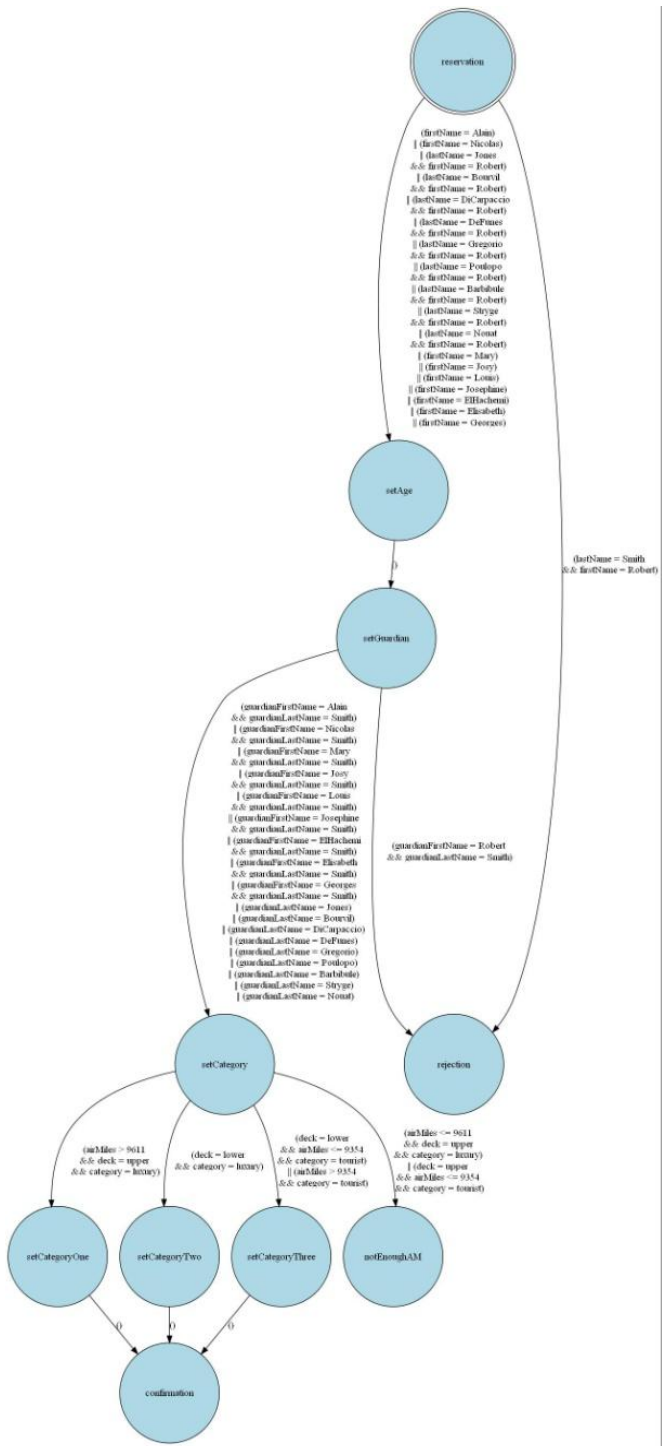


Fig. 9. Model of the WBA for the goal "purchase a ticket for a child" (200 traces).

VI. Conclusion

In this paper, we proposed an integrated formal framework for the analysis and verification of Web services composition. The framework features both static and dynamic analysis techniques, which complement each other. We also discussed the development of a library of patterns and antipatterns of interesting specifications of web services. These specifications will be automatically translated into formal specification languages, currently LTL is being considered. We also presented the component of the framework responsible for the inference of behavioral models of WS applications and for the run-time verification of such applications against the desired properties.

Needless to mention that the inferred models, which depict the data and control flows of an application, can be useful in various development activities like testing and validation. The adoption of goal-based reengineering of WS application models allows us to customize the models to reduce the complexity of handling them especially in automated environments.

Based on our previous experience and the initial results obtained in the use of our formal approach for run-time verification, we believe that results of this proposed work are promising.

Major improvements of the proposed approach include extending the modeling technique to cover all types of WS applications by adopting a decentralized monitoring approach, where each service involved in the application can be represented by its reported behavior. The challenge in this case becomes the consolidation of different sessions produced by different monitors and defining a global order on the recorded messages or events. The future plans to advance this research is to apply the proposed set of tools to specific application domains where the use of web services based application is witnessing significant growth as is the case in education. Many universities are integrating multiple enterprise solutions including legacy ones through the use of web services to avoid rewriting large modules of existing applications. In this context, verifying and testing the integrated solutions imposes the use of dynamic analysis through inference of behavioral models due to the lack of proper access to source code of legacy systems.

Finally, the proposed approach can be extended through combining multiple models to infer a global model of an application, which offers more coverage of the behavior of the application. In addition, multiple goals that are satisfied in collected traces can themselves be used to reengineer partial specifications of the original application.

References

[1] G. O. Young, "Synthetic structure of industrial plastics (Book style with paper title and editor)," in *Plastics*, 2nd ed. vol. 3, J. Peters, Ed. New York: McGraw-Hill, 1964, pp. 15–64.

[2] Andrews, J. Offutt J, R. Alexander, "Testing Web Applications by Modeling with FSMs", Software Systems and Modeling, 4(3):326-345, July 2005.

[3] D. Angluin, "Learning regular sets from queries and counterexample"s, Information and Computation, v.75 n.2, 1987, pp.87-106.

[4] A. Arkin, S. Askary, S. Fordin, et al. "Web Service Choreography Interface (WSCI) 1.0". Retrieved on April 10, 2005 from www.w3.org/TR/wsci.

[5] S. Boroday, A. Petrenko, J. Sing, and H. Hallal, "Dynamic Analysis of Java Applications for MultiThreaded Antipatterns", In Proceedings of the Third International Workshops on Dynamics Analysis (WODA 2005). St-Louis, MI, USA, 2005.

[6] E. M. Clarke, O. Grumberg, D. A. Peled, "Model Checkin"g. The MIT Press, 2000.

[7] M. Dwyer, G. Avrunin, J. Corbett, "Patterns in Property Specifications for Finite-state Verification", In Proceedings of the 21st Int. Conference on Software Engineering, May, 1999.

[8] H. Foster, S. Uchitel, J. Magee, & J. Kramer, "Model-based Verification of Web Service Compositions". In Proceedings of 18th IEEE International Conference on Automated Software Engineering, 2003, pp. 152-161. Montreal, Canada.

[9] H. Foster, "Tool Support for Safety Analysis of Service Composition and Deployment Models". In Proceedings of the 2008 IEEE International Conference on Web Services, 2008, pp. 716-723. IEEE Computer Society.

[10] X, Fu, T. Bultan, & J. Su, "Analysis of interacting BPEL Web Services". In Proceedings of the 13th International World Wide Web Conference, 2004, pp. 621-630. ACM Press.

[11] X. Fu et al, "Analysis of interacting BPEL Web Services". In 13th Int. World Wide Web Conference, 2004.

[12] H. H. Hallal, E. Alikacem, W. P. Tunney, S. Boroday, A. Petrenko, "Antipattern-Based Detection of Deficiencies in Java Multithreaded Software", In Proceedings of Fourth International Conference on Quality Software (QSIC'04), 2004, pp.258-267.

[13] H. H. Hallal, M. Haidar, "Goal Based Reengineering of Web Business Applications", In Proceedings of the 11th Conference on Software Engineering and Research Practice (SERP 12). 2012, USA.

[14] H. Hallal, A. Dury, A. Petrenko, "Web-FIM: Automated Framework for the Inference of Business Software Models". In Services2009 Competition Conference Associated with ICWS 2009 International Conference on Web Services, 2009, Los Angeles, USA.

[15] H. Hallal, S. Boroday, A. Petrenko, A. Ulrich, "A Formal Approach to Property Testing in Causally Consistent Distributed Traces", Formal Aspects of Computing, 2006, 18(1): 63-83.

[16] M. Haydar, A. Petrenko, and H. Sahraoui, H, "Formal Verification of Web Applications Modeled by Communicating Automata", In Proceedings of 24th IFIP WG 6.1 IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2004), pp. 115-132. Madrid, Spain. [LNCS, vol. 3235]

[17] M. Haydar, "A Formal Framework for Run-Time Verification of Web Applications: An Approach Supported by Scope Extended Linear Temporal Logic". VDM Verlag, Germany, ISBN: 978-3-639-18943-8, 2009.

[18] M. Haydar, H. Sahraoui, and A. Petrenko, "Specification Patterns for Formal Web Verification", In Proceedings of 8th International Conference on Web Engineering (ICWE 08), 2008, Yorktown Heights, New York, USA.

[19] M. Haydar, S. Boroday, A. Petrenko, and H. Sahraoui, "Properties and Scopes in Web Model Checking", In Proceedings of 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 05), 2005, Long Beach, California, USA.

[20] M. Haydar, S. Boroday, A. Petrenko, and H. Sahraoui, "Propositional Scopes in Lenear Temporal Logic", In Proceedings of 5th International Conference on New Technologies of Distributed Systems (NOTERE 05), 2005, Gatineau, Quebec, Canada.

[21] G. Holzmann, "The SPIN Model Checker: Primer and Reference Manual". ISBN-10: 0321228626. Addison-Wesley, September 2003.

[22] G. J. Holzmann, "The SPIN Model Checker". Addison-Wesley, 2003.

[23] S. Kallel, A. Char, T. Dinkelaker, M. Mezini, M. Jmaiel, "Specifying and Monitoring Temporal Properties in Web services Compositions". In Proceedings of the 7th IEEE European Conference on Web Services (ECOWS), 2009.

[24] N. Kavantzas, D. Burdett, G. Ritzinger, "Web Services Choreography Description Language", (WS-CDL) 1.0.

[25] R. Kazhamiakin, M. Pistore, and M. Roveri, "Formal Verification of Requirements Using Spin: A Case Study on Web Services". In SEFM'04: Proceedings of the Software Engineering and Formal Methods, 2004, pp. 406–415.

[26] M. Koshkina, F. van Breugel, "Modeling and Verifying Web Service Orchestration by Means of the Concurrency Workbench". SIGSOFT Software Engineering Notes, 2004, 29(5):1-10. ACM.

[27] S. Nakajima, "Model-Checking Behavioral Specification of BPEL Applications". In Proceedings of the International Workshop on Web Languages and Formal Methods, 2006, 2(151):89-105, ENTCS.

[28] OASIS, OASIS Web Services Business Process Execution Language. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

[29] Oracle, http://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html

[30] J. Simmonds, Y. Gan, M. Chechik, S. Nejati, B. O'Farrell, E. Litani, J. "Waterhouse, Runtime Monitoring of Web Service Conversations". IEEE Transactions on Services Computing, 2009, 99, 223-244.

[31] F. Van Breugel, M. Koshkina, "Dead-Path-Elimination in BPEL4WS", In Proceedings of the 5th International Conference on Application of Concurrency to System Design, 2005, pp. 192-201. IEEE Computer Society.

[32] W3C. (2004). "Web Services Glossary". Retrieved on Oct 18, 2008 from http://www.w3.org/TR/ws-gloss/.

[33] Zheng, L, Fan, H. Liu, Y. Liu, W. Ma, L. Wenyin, "User Intention Modeling in Web Applications Using Data Mining", World Wide Web: Internet and Web Information Systems, 5, 181–191, Kluwer Academic, 2002.

[34] Lazic, Ljubomir, and Nikos Mastorakis. "Cost effective software test metrics." WSEAS Transactions on Computers Volume 6, Number 7, pp. 599-619, 2008.