Vol. 4.Special Issue ICCSII ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences

http://www.cisjournal.org

Beyond Functional Verification of Web Services Compositions

[1]Naseem Ibrahim, [2]Ismail Al Ani

[1]Department of Math and Computer Science, Albany State University, USA

[2]Faculty of Engineering and Computer Science, Ittihad University, UAE.

E-mail: [1]naseem.ibrahim@asurams.edu, [2]iialani@ittihad.ac.ae

ABSTRACT

A service composition is an aggregation of multiple interacting Web Services. The goal of the service composition is to provide a new complex functionality that meets a specific set of requirements. It is essential to verify that the functional behavior of the service composition meets the published functionality of the service. It is also necessary to verify that the nonfunctional and trustworthiness properties are satisfied by the service composition. Instead of defining a new verification tool to verify the service composition we follow a transformation approach. In this approach, a service composition can be automatically transformed into a model understood by an available verification tool that can then be used to perform the formal verification. The goal in our research is to use different verification tools in order to verify a wide range of properties and target different kinds of systems. This is because different verification tools differ in their requirements and abilities. In this paper, we define the transformation rules to generate a model that can be verified using UPPAAL [1] model checking tool.

Keywords: *Web Services, Nonfunctional properties, formal verification.*

1. INTRODUCTION

Service-oriented applications are created by composing services together to create new services that provide more complex functionalities. Service composition may be attempted either at service publication time or at service execution time. The former is called static service composition and the latter is called dynamic service composition. In this paper we will focus on static service composition. Web services is considered the de facto standard for providing service-oriented applications. Web services provides a set of standardized XML-based languages that support the different stages in service provision. One of these languages that is concerned with Web services composition is the Business Process Execution Languages for Web Service (BPEL). BPEL [2] is considered the de facto standard used to describe an executable Web service process. BPEL provide the constructs that can be used to specify complex service compositions representing complex business collaborations. Complex business collaborations results in complex business specification. Specifying complex process is error prone, due to many reasons, including [3]: 1) concurrency in the execution of activities, 2) the possibility of communication errors, and 3) faults in remote systems.

BPEL and similar approaches composes servces with respect to their functionality. In the last few years a number of approaches have extended BPEL to include nonfunctional and trustworthiness properties.

To deal with the complexity of specifying complex service commotions we propose formal verification. Formal verification is an important approach to formally verify the interaction in complex business specification. This verification process should consider both service functionality, and the nonfunctional and trustworthiness properties.

There are many Formal verification approaches. The main two approaches are theorem proving and model checking.

There are many available model checking tools in the literature. An important example is the UPPAAL [1] tool. In this paper we discuss the transformation of a service composition defined using BPEL into a model of timed automata understood by the UPPAAL tool, which then can be used to perform the formal verification.

The rest of this paper is organized as follows. Section 2 briefly describes the relationship between service composition and nonfunctional properties. Section 3 provides a brief introduction to formal verification. Section 4 introduces the model checking tool UPPAAL and its model. Section 5 gives a brief introduction to the BPEL language. Section 6 discusses the verification process of service functionality and the transformation rules from BPEL to UPPAAL model. Section 7 discuses verifying nonfunctional properties in UPPAAL. Section 8 presents an example. Finally, some concluding remarks are represented.

2. SERVICE COMPOSITION and NONFUNCTIONAL PROPERTIES

Service composition is understood in SOC to be a process that aggregates services to create new services that provide complex functionality.

Traditional service composition approaches focuses on aggregating the functionality of the service. They are concerned with generating new complex Web service that provide complex functionalities.

In the last few years the attention to nonfunctional properties and especially trustworthiness properties has been increased. Trustworthiness is defined in the literature as the system property that denotes the degree of user confidence that the system will behave as expected. It includes properties such as safety, security, availability and reliability.

Safety is the quality of the operational behavior of the system in which no system action (service) that may lead to catastrophic consequences will be triggered. Safety includes timeliness properties that describe time constrained service execution behavior.

Security is a composite property that includes confidentiality and integrity. Confidentiality ensures that system services and information (data) are not disclosed to unauthorized users. Integrity ensures that there is no improper

alteration to the system state or the published service information.

Reliability is the quality of continuing to provide correct services despite any failure. It is possible to have an accepted frequency of failures. Reliability is defined as the guaranteed maximum number of failures in a unit of time.

Availability means readiness to provide correct service in a user specified context. It is the quality of operation in which there is no unforeseen or unannounced disruption of service. A temporary outage of service may not cause big problems for a noncritical system. The required services can be requested at a later point of time when the system becomes available. However, any service outage for a safety-critical system may lead to catastrophic consequences. When a system fails, availability specifies the maximum accepted time of repair until the service returns back to operate correctly.

With the increased attention to trustworthiness properties a number of approaches have extended the traditional Web services definitions to include trustworthiness properties. This paper focuses on the properties safety and security. It presents an approach that uses model checking tools to verify the functional behavior and the trustworthiness properties of Web services compositions.

3. MODEL CHECKING

Formal verification is using mathematical techniques to ensure that a design conforms to some precisely expressed notion of functional correctness. The two important types of formal verification are theorem proving and model checking. Theorem proving is one of the earliest approaches to formal verification. It uses axioms and rules to prove system correctness. The main advantages of this approach is that it can be mechanically checked where the main disadvantage is that deriving a formal proof is overwhelming tedious. There are no guarantees that it will terminate and it is time consuming. Model checking has a behavioral view of the verification process while theorem proving approach has a structural view. Figure 1, illustrates the main idea behind model checking.

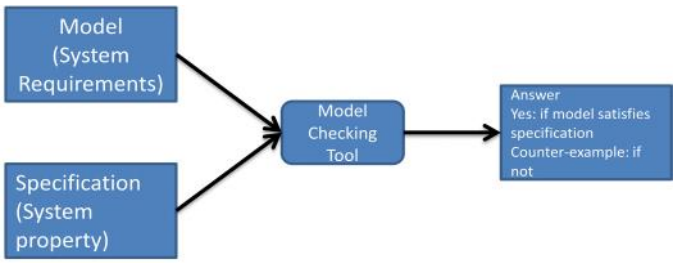

Figure 1. Model Checker

In model checking [4] , the model checking tool takes as an input the requirements or design (called models) and a property (called the specification) that the system should satisfy. The output of the tool is either yes if it satisfies the specification and no otherwise. The main strength of model checking is that it is automated. The main drawback is that it is difficult to judge whether the formulas that has been checked completely characterizes the desired behavior of the system. Verification is not a simple task. The verification complexity comes from the complexity of the systems being verified. Complex systems are difficult to manage. In some cases, more time and effort are devoted to verification than to design.

To solve the verification problem we need automated verification methods, and an integration of the verification process in the design process. And this is what will be introduced in this paper.

In the next section we will introduce UPPAAL, a notable example of model checking tools for real-time systems.

4. UPPAAL

UPPAAL [1] (shown in Figure 2) is a mature tool for the modeling, simulation and verification of real-time systems. It is designed to verify systems which can be modeled as networks of timed automata (TA) extended with integer variables, structured data types, and channel synchronization.

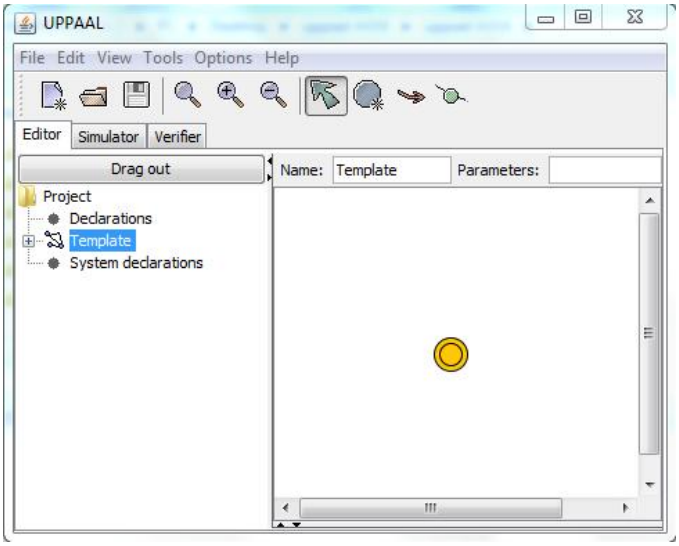

Figure 2. UPPAAL

A TA is a finite-state machine extended with clock variables. It can be formally defined as a tuple $< L, L_0, K, A, E>$, where L is a set of locations denoting the states, L_0 is the initial state, K is a set of clocks, A is a set of actions that cause transitions between locations, E is a set of edges, E in (L X A X B(K) X 2^k X L), where B(K) is the set of data and time constraints that restrict the transitions and 2^k is the set of clock initializations to set clocks whenever required, I is a set of invariants, where $I : L \rightarrow B(K)$ is a function that assigns time constraints to clocks. UPPAAL extends the definition of TA with additional features. Below are some of these features that are relevant to our goal.

Templates: TAs are defined as templates with optional parameters. Parameters are local variables that are initialized during template instantiation in system declaration. Global variables: Global variables and user defined functions can be introduced in a global declaration section. Those variables and functions are shared and can be accessed by all templates.

Binary synchronization: Two TA can have a synchronized transition, caused by an event, when both move

to new state at the same time when the event occurs. An event that causes synchronous transition is defined as a channel, a UPPAAL data type. A channel can have two directions: input (labeled with ?) and output (labeled with!).

Committed Location: Time is not allowed to pass when the system is in a committed location. If the system state includes a committed location, the next transition must involve an outgoing edge from the committed location.

Expressions: There are three main types of expressions

(1) Guard expressions, which are evaluated to Boolean and used to restrict transitions, they may include clocks and state variables,
(2) Assignment expressions, which are used to set values of clocks and variables, and
(3) Invariant expressions, which are defined for locations and used to specify conditions that should be always true in a location.

Edges: Edges denote transitions between locations. An edge specification consists of four expressions

(1) Select, which assigns a value from a given range
(2) Guard, an edge is enabled for a location if and only if the guard is evaluated to true,
(3) Synchronization, which specifies the synchronization channel and its direction for an edge, and
(4) Update, an assignment statements that reset variables and clocks to required values.

UPPAAL can be used by users to specify a checking formula that contains a set of properties. The checking formula can be a combination of the following [5]:

(1) A[] φ, which means φ will invariantly happen,
(2) E<> φ, which means φ will possibly happen,
(3) A<> φ, which means φ will always happen eventually,
(4) E[]φ, which means φ will potentially always happen, and
(5) φ → ψ , which means φ will always lead to ψ.

Where φ and ψ are Boolean expressions defined on locations, integer variables, and clocks constraints. The properties that can be checked using UPPAAL are [1]:

- ***Reachability***: UPPAAL can check whether or not it is possible to reach a certain location. It also checks whether or not there is a deadlock in the system.
- ***Safety***: UPPAAL can check whether or not anything bad will ever happen, declared in UPPAAL as something good is always true.
- ***Liveness***: UPPAAL can check whether or not something will happen eventually.

5. BUSINESS PROCESS EXECUTION LANGUAGE (BPEL)

Business Process Execution Language (BPEL) [6] is an XML based language that was designed to enable the coordination and composition of services. It has emerged as the standard to define and manage composition for Web services. BPEL is based on the Web services Description Language WSDL [7]. BPEL provides several features to facilitate the modeling and execution of service compositions, such as [8]:

- It models business process collaboration.
- It models the execution control of business processes.
- It separates abstract definition from concrete binding.
- It represents the participants' roles and role relationships.
- It supports compensation.
- It supports service composability.
- It facilitates context support.
- It supports event handling.

BPEL use a workflow-based approach to provide behavioral extension to WSDL. BPEL defines relationships by invocations using control and data flow links. Process [9] is the main construct to model the composition of services. It is a concurrent description that connects activities that send and receive messages. External Web services providers are defined as port of a particular port type. A port type has a WSDL description. Partner links are used to specify which activity is linked to which port provider.

The basic element in the BPEL process is called an activity, it can either be a primitive or a structured activity. Primitive activities contain:

(1) ***invoke***: which invokes an operation of some Web service,
(2) ***receive***: which waits for a message from an external source,
(3) ***reply***: which reply to an external source,
(4) ***wait***: which wait for some time,
(5) ***assign***: which copy data from one place to another,
(6) ***throw***: which indicate errors in the execution,
(7) ***terminate***: which terminate the entire service instance, and
(8) ***empty***: which does nothing.

On the other hand, structured activities contain:

(1) ***sequence***: which defines an execution order,
(2) ***switch***: which is used for conditional routing,
(3) ***while***: which is used for looping,
(4) ***pick***: which is used for race conditions based on timing or external triggers,
(5) ***flow***: which is used for parallel routing, and
(6) ***scope***: which is used for grouping activities.

6. FUNCTIONAL VERIFICATION

To formally verify the correctness of the service composition we propose a transformation approach. In this approach, a service composition defined using BPEL is transformed into timed automata that can be formally verified using the model checking tool UPPAAL. The input to this transformation process is the service composition defined in BPEL. The output to this transformation process is a system of timed automata's that can be verified using UPPAAL.

Each BPEL model can be translated into one UPPAAL model in a one-to-one relationship. The resulted UPPAAL model will consist of multiple TA's. The transformation rules can be divided into two sets. The first set

is template level rules and the second set in automata level rules.

The template level rules define the rules for creating the timed automata's. It defines the mapping between the BPEL constructs and the TA's. These rules are stated below:

- Each BPEL sequence in translated to one TA that represents the main automata. This automata will trigger other automata's using channels.
- Each BPEL Flow is translated into one TA.
- Each BPEL Link is translated into one TA.

The automata level rules define the transformation rules for each automata. It defines how the automata states and edges are defined, and the mapping between the BPEL definitions.

- Synchronization is achieved by using UPPAAL channels. They indicate the synchronizes trigger of events.
- A BPEL flow is defined by a state with multiple edges starting from it.
- A BPEL switch is represented in the same manner as a flow but with the conditions added as guard statements on the edges exiting from the switch state.
- Internal events need not to be represented in the automata as channels.
- Every Activity or link is defined by two states, Start state and End state.

7. NONFUNCTIONAL VERIFICATION

The TA's generated from the basic transformation has been extended to include the nonfunctional properties safety and security. These properties have been verified using the model checking tool UPPAAL.

6.1 Safety

The service composition has been extended to include safety properties. UPPAAL is used to make sure that those properties are satisfied. Those properties are added as global level variables in the UPPAAL system, and are added as guard statements on the TA edges. This will ensure that those edges and hence the TA will only be triggered if this property is satisfied.

This has enabled the testing of the behavior of the system in case those properties are satisfied or not. For example, a data safety property that states "Web service B can only be invoked if X > 5", can be added and tested in UPPAAL. If this is not satisfied, B should not be invoked and another path should be followed. This behavior should be tested as it might cause deadlocks to the system.

6.2 Security

The service composition has been extended to include security mechanism. This can be achieved following a similar approach to safety properties. A security function is defined in the declaration section of the UPPAAL model. A call to this function is added on the TA edges in the guard part. Those functions return a Boolean value. If they return true then the user have access to this edge and can invoke this Web service. If they return false, then the user does not have the access rights and the Web service should not be invoked. The function works with a list of users and access values. It defines who can and who cannot activate an edge. The functions can accept a user name and return the access right (true or false).

8. EXAMPLE

This section represents a simple example that illustrates the transformation approach presented in this paper. Figure 3 illustrates the service composition that we need to verify. It consists of six activities starting with A which is a receive activity. Then a concurrent scope flow follows A, that contains four invoke activities B, C, D, E. The links modify the execution order. C should be executed after B, and E after D and have to wait for B to finish. The activity B is reply activity.

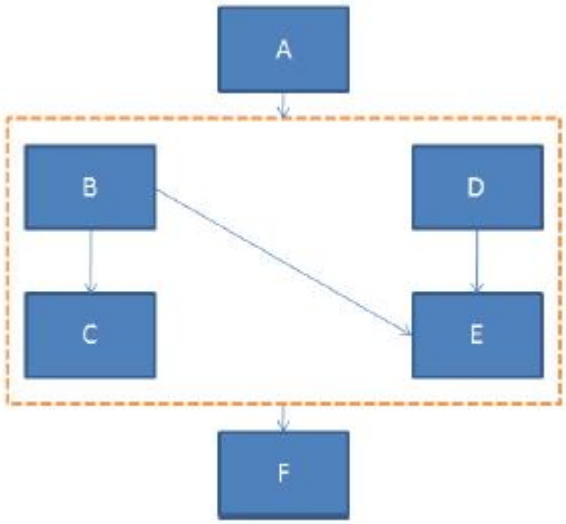

Figure 3. Business Process of the Example

Applying the transformation rules discussed above will result in the following automatas:

1) Main Automata: this automata represents the main sequence, and is presented in Figure 4.
2) Flow1 Automata: this automata represents the first flow that starts with Activity B, and is presented in Figure 5.
3) Flow2 Automata: this automata represents the second flow that starts with Activity D, and is presented in Figure 6.
4) Link1 Automata: this automata represents the link between Activity B and C and invoke of activity C, and is presented in Figure 7.
5) Link2 Automata: this automata represents the link between Activity B and E, and is presented in Figure 8.

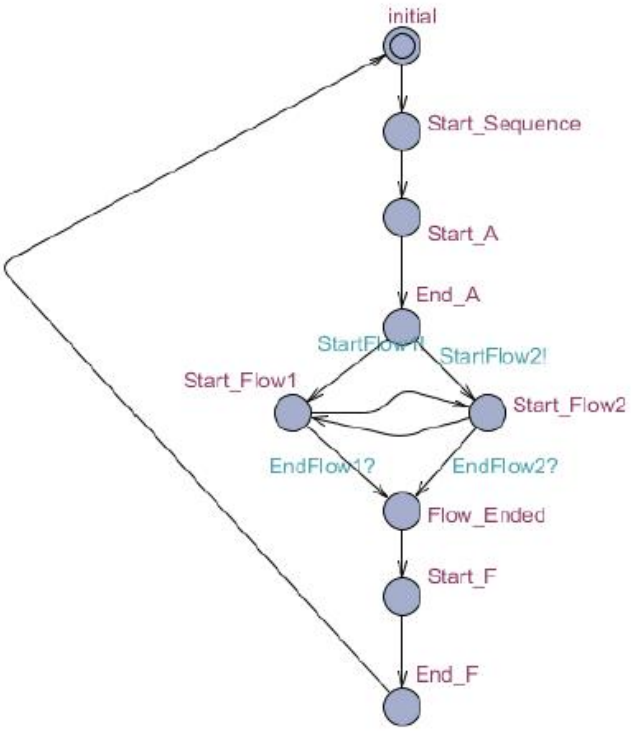

Figure 4. Main Automata

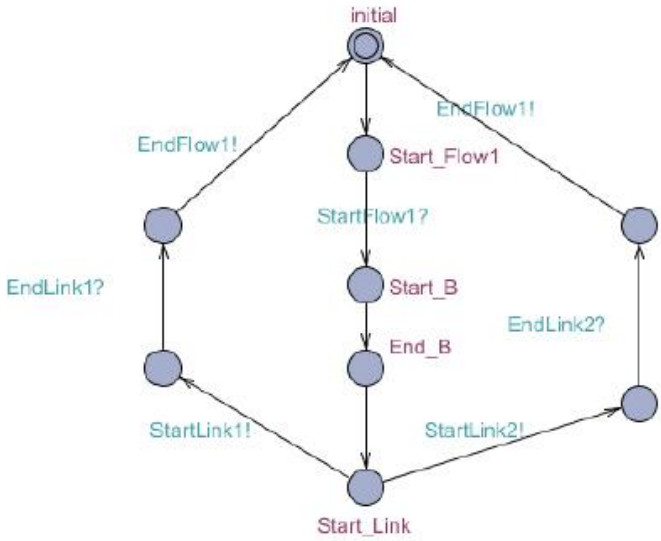

Figure 5. Flow 1 Automata

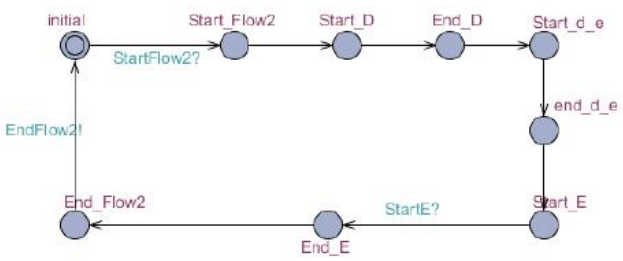

Figure6. Flow 2 Automata

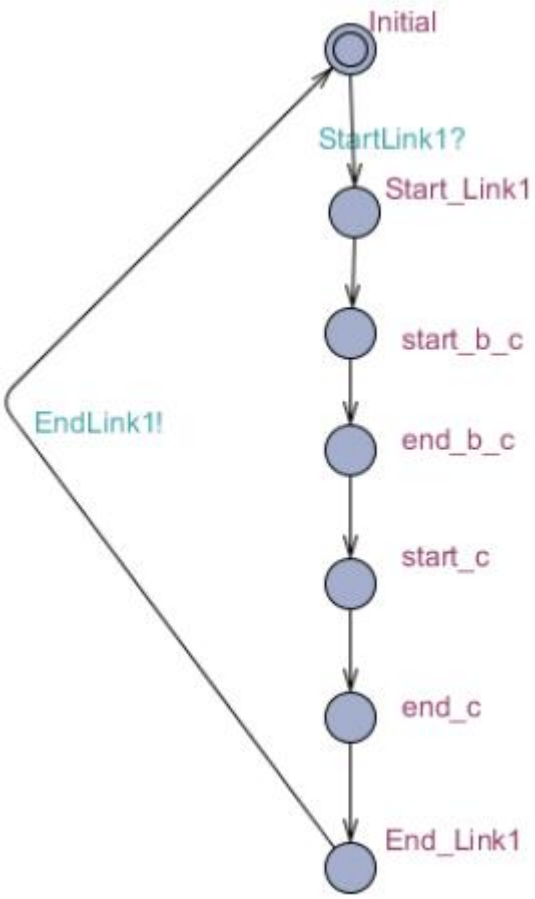

Figure7. Link 1 Automata

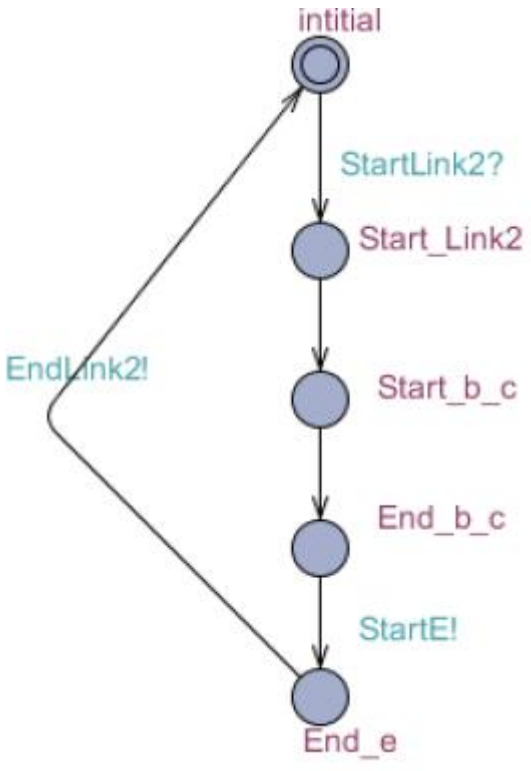

Figure 8. Link 2 Automata

The resulted automatas is then verified using the model checking tool UPPAAL. Below is a sample set of properties that was verified using UPPAAL:

- **A[] not deadlock:** which checks if the model has any deadlock, and the result was the property satisfied.
- **E<> f2.Start_E:** which checks if there is a way for Activity E to be invoked, and the result was the property satisfied.
- **E<> m. Start_A**: which checks if there is a way for Activity A to be invoked, and the result was the property satisfied.
- **E<> m. Start_F**: which checks if there is a way for Activity F to be invoked, and the result was the property satisfied.
- **E<> f1.Start_B:** which checks if there is a way for Activity B to be invoked, and the result was the property satisfied.
- **E<> l1.start_c**: which checks if there is a way for Activity C to be invoked, and the result was the property satisfied.
- **E<> f2.Start_D**: which checks if there is a way for Activity D to be invoked, and the result was the property satisfied.

9. CONCLUSION

In this paper we have discussed the transformation of a service composition defined using BPEL into an automata that can be verified using the UPPAAL model checking tool. We have also presented an extension to this approach where UPPAAL can also be used to verify the nonfunctional properties safety and security.

The transformation process presented in this paper has been automated. The automation was achieved using XSLT [10]. XSLT was used to formally define the transformation rules from BPEL to UPPAAL. Both BPEL and UPPAAL are XML based, which made XSLT the perfect transformation language. The automated transformation process was tested using the Automotive Case Study [11] [12], a widely presented case study in the literature of SOA.

REFERENCES

[1] G. Behrmann, A. David, and K. G. Larsen, “A tutorial on UPPAAL,” in Formal Methods for the Design of Real- Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004, vol. LNCS 3185. Springer– Verlag, September 2004, pp. 200–236.

[2] y. P. Yuhong Yan, Philippe Dague and M.-O. Cordier, “A model-based approach for diagnosing faults in web service processes,” the International Journal on Web Service Research, vol. 6, no. 1, pp. 87–110, January-March 2009.

[3] L. S. F. Jesus Arias Fisteus and C. D. Kloos, “formal verification of bpel4ws business collaborations,” in K. Bauknecht, M. Bichler, and B. Proll (Eds.): EC-Web 2004, vol. LNCS 3182. Springer-Verlag, 2004, p. 76.

[4] G. K. Palshikar, “An introduction to model checking,” http://www.embedded.com/columns/technicalinsights/17603352? requestid=179878, December 2004.

[5] J. Bengtsson and W. Yi, “Timed automata: Semantics, algorithms and tools,” The United Nation University, P.O.Box 305, Macau, Report 316, September 2004.

[6] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana, “The next step in web services,” Commun. ACM, vol. 46, no. 10, pp. 29–34, 2003.

[7] WSDL, “Web services description language 1.1,” W3C Note. March, 2001. http://www.w3.org/TR/wsdl.

[8] M. P. Papazoglou, WEB SERVICES: PRINCIPLES AND TECHNOLOGY, 1st ed. Prentice Hall, 2008.

[9] M. H. ter Beek, A. Bucchiarone, and S. Gnesi, “Formal methods for service composition,” Annals of Mathematics, Computing and Teleinformatics, vol. 1, no. 5, pp. 1–5, 2007.

[10] D. Tidwell, Mastering XML Transformation XSLT. O’Reilly, 2001.

[11] M. H. ter Beek, S. Gnesi, N. Koch, and F. Mazzanti, “Formal verification of an automotive scenario in service oriented computing,” in Proceedings of the 30th international conference on Software engineering, ser. ICSE ’08. New York, NY, USA: ACM, 2008, pp. 613–622.

[12] D. Berndl and N. Koch, “Sensoria automotive scenario: Illustrating servicespecification,” - FAST, No. 2, Tech. Rep., August 2007.

AUTHOR PROFILES

1. Naseem Ibrahim received his PhD in computer science from Concordia University (Canada) in 2012. Currently, he is an Assistant Professor of Computer Science at Albany State University, Georgia, USA.

2. Ismail Al Ani received his PhD in computer science from Kent University (UK) in 1987. Currently, he is the dean of the Faculty of Engineering and Computer Science at Ittihad University, RAK, UAE.