Components Monitoring through Formal Specifications

Paolo Arcaini
Dip. di Tecnologie dell'Informazione
Università degli Studi di Milano, Italy
paolo.arcaini@unimi.it

Angelo Gargantini
Dip. di Ing. dell'Informazione e Metodi Matematici
Università di Bergamo, Italy
angelo.gargantini@unibg.it

Elvinia Riccobene
Dip. di Tecnologie dell'Informazione
Università degli Studi di Milano, Italy
elvinia.riccobene@unimi.it

ABSTRACT

The paper presents a specification-based approach for runtime monitoring of components in the field of component-based software engineering. The conformance of a component is checked with respect to a formal specification given in terms of Abstract State Machines. The validity of the approach is proved showing how the technique can be used for the monitoring of web services developed using Axis2. The theoretical approach is implemented in a technical framework where Java annotations are used to link the web service with its formal specification, and AspectJ is used to check the conformance runtime.

Categories and Subject Descriptors: D.2.4 [Software/Program Verification]: Formal methods

General Terms: Verification

Keywords: Abstract State Machines, Component-based software engineering, Runtime monitoring

1. INTRODUCTION

The aim of *runtime monitoring* (or *runtime verification*) is to check that a software (or even an hardware component) behaves correctly during its execution. While other formal verification techniques, such as model checking and theorem proving, aim to ensure universal correctness, the intention of runtime monitoring techniques is to determine whether the observed executions behave as expected. Thus, monitoring is a lightweight verification technique that can be used when exhaustive verification can not be executed, because too expensive or because the environment in which the software is executed is not reproducible at verification time.

In almost all the techniques proposed in literature for runtime monitoring of software, the expected behavior of the system is formalized by means of correctness properties [9] which are then translated into *monitors*. The monitors are used to check if the properties are violated during the execution of the software. The correctness properties are formalized using *declarative* languages as extended regular expressions, variants of Linear Temporal Logic (LTL), Java Modeling Language (JML), etc.

Another approach for describing the expected behavior would be to use *operational* specifications that provide a model implementation (or model program) of the system, that in general is executable. Examples of operational specifications are abstract automata and state machines. The use of operational specifications for runtime monitoring has not been investigated deeply: In [18] a user-guided runtime monitoring approach based on the use of Z specifications is presented, and in [1] we introduced an approach for automatic runtime monitoring of Java programs. Since some people find that operational specifications are easier to write and understand than declarative ones, the aim of our work was to show that also operational specifications can be used for runtime monitoring.

Component-based software engineering (CBSE) is a reuse-based approach to software systems development [20]. The aim of CBSE is to produce *independent components* that are completely specified by their interfaces. The implementation of a component should be separated from its interface so allowing to substitute a component without affecting the overall system. A problem that arises in CBSE is how to assure that components behave as expected. As stated in [20], *a viable solution is to certify that components conform to a formal specification.* Usually, classical approaches used to guarantee the component correctness are model based testing, formal verification, etc., that are executed before the component deployment.

In this paper, instead, we propose a general framework in which the expected behavior of a component is given in an *operational* way in terms of Abstract State Machines (ASMs), and the assurance that the implementation behaves as expected is done through runtime monitoring after the component deployment. Our approach permits to check, not only the correctness of the implementation, but also that the component is accessed in the right way (e.g., interface contracts on the calling order of the interface services are respected). This work extends the one in [1] to deal with a wider range of systems.

Section 2 introduces the necessary background on ASMs. Section 3 presents the theoretical framework, in which we explain the syntactical links between a component and its formal specification, and the semantic relation which represents the conformance. Section 4 shows how the theoretical framework of the proposed monitoring approach can be implemented for the web services scenario using Java annotations and AspectJ, and Section 5 presents a small case

WCOP'12, June 25–28, 2012, Bertinoro, Italy.

study example. Section 6 presents related work, and Section 7 concludes the paper indicating some future directions of this work.

2. ABSTRACT STATE MACHINES

Abstract State Machines (ASMs), whose complete presentation can be found in [5], are an extension of Finite State Machines (FSMs), where unstructured control states are replaced by states with arbitrary complex data. The *states* of an ASM are multi-sorted first-order structures, i.e., domains of objects with functions and predicates defined on them. ASM states are modified by *transition relations* specified by "rules" describing the modification of the function interpretations from one state to the next one. There is a limited but powerful set of *rule constructors* that allow to express guarded actions (`if-then`), simultaneous parallel actions (`par`) or sequential actions (`seq`). Appropriate rule constructors also allow non-determinism (existential quantification `choose`) and unrestricted synchronous parallelism (universal quantification `forall`).

An ASM state is a set of *locations*, namely pairs (*function-name*, *list-of-parameter-values*). Locations represent the abstract ASM concept of basic object containers (memory units). Location *updates* represent the basic units of state change and they are given as assignments, each of the form $loc := v$, where *loc* is a location and v its new value.

Functions may be *static* (never change during any run of the machine) or *dynamic* (may change as a consequence of agent actions or *updates*). Dynamic functions are distinguished between *monitored* (only read by the machine and modified by the environment), and *controlled* (read and written by the machine).

A *computation* of an ASM is a finite or infinite sequence $s_0, s_1, \ldots, s_n, \ldots$ of states of the machine, where s_0 is an initial state and each s_{n+1} is obtained from s_n by executing its (unique) *main rule*. An ASM can have more than one *initial state*. It is possible to specify state *invariants*. Because of the non-determinism of the choose rule and of the moves of the environment, an ASM can have several different runs starting in the same initial state.

Code 1 reports the ASM specification of a basic e-commerce web service: the service permits to create a cart and add an item to the cart. In the ASM model, the operation chosen by the user is modeled through the monitored function *operationCalled* that can take value *CC* if the user wants to create a cart and *ATC* if he/she wants to add one item to the cart. The controlled part of the ASM state is composed of the boolean *cartCreated* function, that records if a cart has been created, and the integer function *elementsInCart* that records how many items have been added to the cart. An invariant checks that items are added to the cart only if the cart has already been created.

The ASMETA toolset is a set of tools around the ASMs [2]. They can assist the user in developing specifications and proving model correctness by checking state invariants during simulation and temporal logic properties through model checking. Among the ASMETA tools, those involved in our runtime monitoring are: the textual notation *AsmetaL*, used to encode ASM models, and the simulator *AsmetaS* [10], used to execute ASM models.

```
asm eshop
import StandardLibrary
signature:
    enum domain OperationCalledDomain = {CC | ATC}
    monitored operationCalled: OperationCalledDomain
    controlled cartCreated: Boolean
    controlled elementsInCart: Integer
definitions:
    rule r_createCart =
        if (operationCalled = CC) then
            cartCreated := true
        endif

    rule r_addToCart =
        if (operationCalled = ATC and elementsInCart < 5) then
            elementsInCart := elementsInCart + 1
        endif

    invariant inv_calledOperationsOrder over operationCalled:
                        operationCalled = ATC implies cartCreated

    main rule r_Main =
        par
            r_createCart[]
            r_addToCart[]
        endpar

default init s0:
    function cartCreated = false
    function elementsInCart = 0
```

Code 1: ASM model of an e-commerce web service.

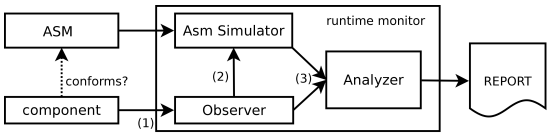


Figure 1: The runtime monitor.

3. ASM-BASED RUNTIME MONITORING

A *runtime software-fault monitor* (or simply a *monitor*) is a system that observes and analyzes the states of an executing software system. The monitor checks the correctness of the system behavior by comparing an *observed* state of the system with an *expected* state. The expected behavior is usually given in terms of a formal specification.

We here propose a monitoring approach for component-based systems where the formal model of a component is given in terms of ASMs. In our technique the monitor observes the behavior of the component and determines its correctness w.r.t. the ASM specification working as an oracle of the expected behavior: while the software system is executing, the monitor checks conformance between the observed state and the expected state. Fig. 1 shows the structure of the proposed framework:

- a linking between a *component* and an *ASM* must be provided in order to describe the conformance relation;
- the *Observer* evaluates when the component observed state is changed (1), and leads the corresponding ASM to perform a machine step (2);
- the *Analyzer* evaluates the conformance between the component execution and the ASM behavior (3). If a conformance violation is detected, a trace in form of counterexample can be recorded for debugging.

In the following, we introduce the theoretical foundation of our monitoring approach. We define what is an observed component state, how to describe the linking between the component and its formal specification and, finally, how to

describe the conformance relation between the component and the ASM.

3.1 Observable components

In order to mathematically represent a component and and formally describe its relation with its formal specification, we introduce the following definitions. We adapt the definitions introduced in [1] for Java programs.

DEFINITION 1. ***Component*** *According to the definition given in [20], a* component *C is a couple $\langle R, P\rangle$ where R denotes a possible empty set of services required by C, and P the non empty set of services provided by C. If C has an internal state that can change as a result of the invocation of one of its services, the component is said* stateful*, otherwise it is said* stateless*.*

DEFINITION 2. ***Pure service*** *Pure services $P_{pure} \subseteq P$ are services provided by the component that are* side effect free *with respect to the component state, that is they provide a result but they do not modify the internal state of the component.*

Pure services are similar to pure methods [8] in object-oriented programs. Pure methods are very useful in verification since they can be invoked runtime, without affecting the program state. Some languages that support the use of pure methods in pre and post-conditions are Eiffel [19] and JML for Java [16].

If $P = P_{pure}$, that is all the services provided by the component are pure, the component is stateless, otherwise is stateful.

Our approach can handle both stateless and stateful components.

DEFINITION 3. ***Observed State*** *Let $Ris(P_{pure})$ be the set of pure services invocation results. We define* external state*, $ES(C) = Ris(P_{pure})$, as the description of the component state that results from the invocation of its pure services. We define* observed state*, $OS(C) \subseteq ES(C)$, as the subset of the external state consisting of all the pure services of the component C that the user wants to observe.*

So, $OS(C)$ describes the portion of the state that the user wants to monitor at runtime.

The values returned by the services in $OS(C)$ can change, as time goes by, as the result of the execution of any not pure service (in $P_{\neg pure} = P - P_{pure}$).

DEFINITION 4. ***Changing service*** *We define* changing service*s, $CS(C) \subseteq P_{\neg pure}$, all services provided by C whose execution can change the interpretation of $OS(C)$ and the user wants to monitor.*

DEFINITION 5. ***Observed input*** *We define* observed input*s, $OI(C) \subseteq R$, all the services required by C that the user wants to monitor.*

Linking a component with an ASM.

Our approach requires that a component C, in order to be runtime monitored, must have a corresponding ASM model ASM_C that abstractly specifies the expected behavior of the component.

The observable state $OS(C)$ must be linked to the controlled functions $ContrFuncs(ASM_C)$ of the model ASM_C. The function

$$linkOS : OS(C) \rightarrow ContrFuncs(ASM_C)$$

establishes the link between the observable pure services belonging to $OS(C)$ and some ASM controlled functions. The function *linkOS* is not surjective because there are controlled functions that are not used in the conformance analysis, and neither injective because different pure services can be linked to the same ASM function.

The observed inputs $OI(C)$ must be linked to the monitored functions $MonFuncs(ASM_C)$ of the ASM model. The function

$$linkOI : OI(C) \rightarrow MonFuncs(ASM_C)$$

establishes the link. Monitored functions are suitable to represent the required services of the component since, in an ASM, they represent the part of the dynamic state that is determined by the external environment and not by the machine.

Execution step in the component and in the ASM.

We borrow from the Unifying Theories of Programming (UTP) [14] the definitions of *machine step* and *last state* of an execution sequence.

A *state* of a component C is the set of the actual values of its internal *fields*.

DEFINITION 6. ***Change Step*** *Let p be a service provided by a component. A* component step *is defined as the relation (s, p, s') where s is the starting state of the execution of p and s' the last state of this execution. A* change step *is defined as a component step for $p \in CS(C)$.*

ASM *state* and ASM computation *step* have been defined in Section 2.

3.2 State, Step and Runtime Conformance

In Section 3.1 we have provided all the elements necessary to link a component with its formal specification given in terms of ASMs.

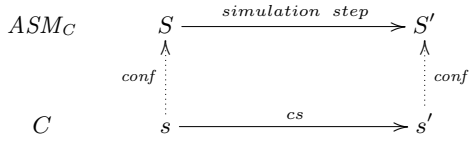
We now report some definitions useful to define the conformance relation. Let C be a component and ASM_C its corresponding ASM abstract model.

The function $val_C(e, s)$ yields the value returned by a service $e \in ES(C)$ of C in a given state s of C. The value of an ASM function f in a state S is given by $val_A(f, S)$. We assume that a conformance relation $\stackrel{conf}{=}$ exists between the values returned by any service and the ASM function values.

DEFINITION 7. ***State Conformance*** *We say that* a state *s of C* conforms to a state *S of ASM_C if all observed elements of C have values conforming to the values of the controlled functions in ASM_C linked to them, i.e.,*

$$conf(s, S) \equiv \forall p \in OS(C) : val_C(p, s) \stackrel{conf}{=} val_A(linkOS(p), S)$$

DEFINITION 8. ***Step Conformance*** *We say that a* change step *(s, cs, s') of C, with $cs \in CS(C)$,* conforms with a step *(S, S') of ASM_C if $conf(s, S) \wedge conf(s', S')$.*



DEFINITION 9. ***Runtime Conformance*** *Given an* observed computation *of a component C, we say that C is* runtime conforming *to its specification ASM_C if the following conditions hold:*

- *the initial state s_0 of the computation of C* conforms *to the initial state S_0 of the computation of ASM_C, i.e., it yields conf(s_0, S_0);*
- *every observed change step (s, cs, s') with s the current state of C,* conforms *with the step (S, S') of ASM_C with S the current state of ASM_C;*
- *no specification invariant of ASM_C is ever violated.*

4. MONITORING WEB SERVICES

The theoretical framework presented in Section 3 is general enough to handle any kind of component-based system. In order to implement the proposed approach for a particular concrete scenario, an ad-hoc technical framework must be selected. As a proof of concept, we here describe how we implemented our monitoring approach in the web services scenario. In our experiments the web services have been developed using Apache Axis2[1], a framework for the development of web applications in Java.

4.1 Providing the linking by Java Annotations

In order to link a web service with its formal specification we use *Java annotations*, that are metadata tags that can be used to add some information to code elements as class declarations, field declarations, etc. Annotations can be defined by the user similarly as classes. It is possible to define annotations that can be accessed at runtime using reflection.

For our purposes, we have defined a set of annotations:

- `@Asm` is used to link a web service with its corresponding ASM model; it has a string attribute that contains the path of the ASM. It must annotate the Java class that implements the web service. Code 2 reports the Java class `Eshop` linked to the ASM *eshop.asm* (see Code 1).
- `@PureService` is used to establish the mapping defined by the function *linkOS*. It annotates each observed pure service $p \in OS(C)$; the annotation has a string attribute yielding the name of the corresponding controlled ASM function. In the example, the observed state is composed just by the pure method *getNumberOfElements*.
- `@Operation` annotates the services in $CS(C)$[2]. The annotation has two string attributes, *func* that permits to specify the name of a monitored function of the ASM model, and *value* that specifies the value to whom the monitored function must be set. This information is used by the runtime monitor that, when the operation is called, sets the value of the monitored function in the simulation of the ASM. In the example, there are two changing services, *createCart* and *addItem*, that are the two operations provided by the web service; in the ASM the monitored function *operationCalled* records which service has been invoked.

[1] `http://axis.apache.org/axis2/java/core/`

[2] The name `@Operation` is due to the fact that in web services the provided interfaces are called *operation*s.

```
package org.ecommerce;
import org.asmeta.monitoring.Asm;
import org.asmeta.monitoring.MethodToFunction;
import org.asmeta.monitoring.RunStep;
import org.asmeta.monitoring.StartMonitoring;

@Asm(asmFile="models/eshop.asm")
public class Eshop {
    private static int counter = 0;
    private String clientID;
    private int numberOfElements;

    @StartMonitoring
    public Eshop() {
        clientID = "client_" + counter;
        counter++;
        numberOfElements = 0;
    }

    @Operation(setFunction = "operationCalled", toValue = "CC")
    public String createCart() {
        //creation of the cart (not reported here)
        return clientID + " - cart created";
    }

    @Operation(setFunction = "operationCalled", toValue = "ATC")
    public String addItem() {
        numOfEls++;
        return clientID + " - item added - " +
                "# elements = " + numberOfElements;
    }

    @PureService(func = "elementsInCart")
    public int getNumberOfElements() {
        return numberOfElements;
    }
}
```

Code 2: E-shop web service.

- `@StartMonitoring` is used to select a proper (not empty) set of constructors; when the runtime monitor detects that an annotated constructor has been called, it starts the monitoring (i.e., it starts a simulation of the corresponding ASM).
- `@Param` is used to link a parameter of a changing service with a monitored function, specified by the string attribute *func*. It is used to establish the mapping defined by the function *linkOI*. The runtime monitor, when the operation is called, sets the value of the monitored function using the value of the actual parameter.

4.2 Runtime monitoring using AspectJ

AspectJ[3] permits to integrate the principles of Aspect-Oriented Programming (AOP) in Java. We used AspectJ to implement the *runtime monitor* (see Fig. 1). Indeed AspectJ permits to easily observe the execution of Java objects: it allows programmers to define special constructs called *aspects* where to specify some code to be executed when the program under monitoring executes some given actions. *Pointcuts* are points of the program execution one wants to capture (e.g., execution of methods with a given signature); for each pointcut it is possible to specify an *advice*, that is the actions that must be executed when the pointcut is reached. The *advice* can be executed *before* or *after* the execution of the code specified by the pointcut.

In our framework we developed just one aspect that permits to runtime monitor all the components that must be monitored. Indeed the pointcuts are general enough to capture the component creation and method executions that

[3] `http://www.eclipse.org/aspectj/`

must be monitored, and the advices are able to dynamically inspect the Java and the ASM states in order to do the conformance checking.

Monitor implementation.

The *Observer* is implemented using two pointcuts that permit to detect, respectively, the creation of a web service, when a constructor annotated with `@StartMonitoring` is called, and the execution of a *changing service*, when a method annotated with `@Operation` is called (we do not consider nested calls of operations):

```
pointcut serviceCreated(): call(@StartMonitoring *.new(..));
pointcut operationCalled(): call(@Operation * *.*(..)) &&
                            !cflowbelow(call(@Operation * *.*(..)));
```

When the creation of a web service is detected (the pointcut *serviceCreated* is reached), an advice creates a simulator for the ASM and connects it with the created web service.

When the call of an operation *cs* is detected (the pointcut *operationCalled* is reached):

1. before the execution of *cs*, an advice (the *Analyzer* in Fig. 1) checks the state conformance between the Java and the ASM state ($conf(s, S)$ in Def. 8);
2. *cs* is executed and an advice performs a step of simulation of the ASM by AsmetaS (the *ASM Simulator* in Fig. 1); before the simulation step, the values of the monitored functions are set according to the values provided by the `@Operation` and `@Param` annotations;
3. after the execution of *cs*, the advice *Analyzer* checks again the state conformance ($conf(s', S')$ in Def. 8).

5. CASE STUDY

We have evaluated our technique on a small example, an e-commerce web service (shown in Code 2). The web service exposes just two operations, *createCart* to create a cart, and *addItem* to add an item to the cart.

A correct usage of the web service requires that:

PROP1: the operation *addItem* is called only if the operation *createCart* has already been called, i.e., one item can be added to the cart only if the cart exists;

PROP2: no more than 5 items are added to the cart.

The assurance of both correctness properties is not guaranteed by the web service implementation. However their violations can be discovered if the web service is runtime monitored using as formal specification the ASM shown in Code 1.

In order to test the web service, we have developed an application client for the Android platform[4]: the application is composed of two buttons that invoke the operations *addItem* and *createCart*, and a text box where the result of an operation execution is shown. As suggested in [13], the developed client is *faulty* since its usage can lead to the violation of the two correctness properties required by the web service. Indeed, we do not hide a button when it should not be called: in such way it is possible that the conversation with the web service is executed wrongly (e.g., an item is added to the cart before the cart has been created).

Fig. 2 shows the client application when a connection with the web service has been established (in the following figures we do not show the two buttons).

Figures 3 and 4 show the application when, respectively, the creation of the cart has been executed and an item has been added to the cart. In both cases the web service is invoked correctly and no correctness property is violated.

[4] `http://www.android.com/`

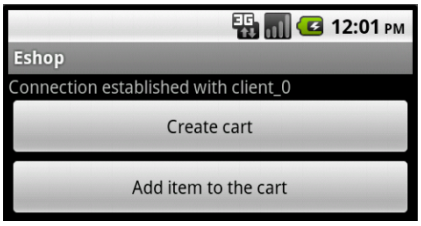


Figure 2: Client application loaded.

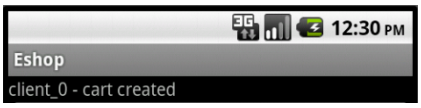


Figure 3: Correct creation of the cart.

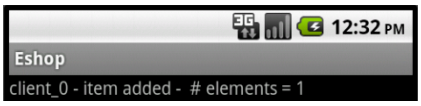


Figure 4: Correct insertion of an item to the cart.

The violation of both correctness properties can be discovered through runtime monitoring.

Violation of **PROP1.** The ASM has a boolean function *cartCreated* that records if a cart has been created, that is if the operation *createCart* has been called. The invariant contained in the machine checks that, when an item is added to the cart, the cart has already been created. Fig. 5 shows the error message that is shown when the "Add Item to the cart" button is selected firstly: the message is generated by the invariant violation during the ASM simulation.

Violation of **PROP2.** The violation of **PROP2** can be discovered when the runtime monitor checks for the state conformance after the execution of an operation. Fig. 6 shows the error message that is obtained if, after the correct creation of the cart, the "Add Item to the cart" button is selected 6 times. The web service does not bound the value of *numberOfElements*, whereas in the ASM the function *elementsInCart* is not incremented if it is greater than or equal to 5.

6. RELATED WORK

Surveys on runtime verification can be found in [7, 17, 9].

In [1] we presented an approach for runtime monitoring of Java programs. That work has been inspired by the work presented in [18], in which the authors describe a *formal specification-based software monitoring system*. In their system they check that the behavior of a concrete implementation (a Java code) complies with its formal specification (a Z model). Another approach that, like ours, uses ASMs as formal specifications for system monitoring purpose is presented in [4].

Several approaches use Aspect-oriented programming for implementing the monitor [6, 15, 12].

One of the properties that our approach can check is that the operations are called in the correct order (e.g.,

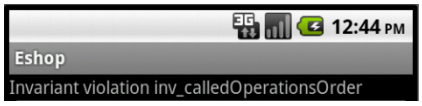


Figure 5: Invariant exception.

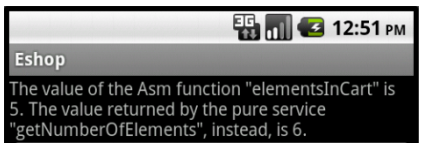


Figure 6: Conformance exception.

PROP1 in our example). Web Services Conversation Language (WSCL 1.0) [3] is a conversation definition language based on XML that permits to specify what the *conversations* (i.e., sequences of operation calls) supported by a web service are. It permits to define the documents to be exchanged and the order in which they can be exchanged. WSCL is responsible of checking just the interface of the web service, the business payload and the choreography of the messages, not its implementation. In [13] a runtime monitor for Ajax applications is presented. It permits to check contracts expressed in $LTL - FO^+$, an extension of LTL. In their case study, based on the Amazon Web Services, they show how contracts related to the order in which the operations are called can be specified and checked.

In our work we monitor the behavior of a single web service. Other research works focus their attention on checking web services orchestration, by runtime monitoring BPEL processes [11].

7. CONCLUSIONS AND FUTURE WORK

We have presented a theoretical framework for the execution of runtime monitoring of component in a component-based software scenario. We claim that our definitions are generic enough to be applied to all the settings that can be described as instantiations of CBSE. As a proof of concept we have applied the proposed approach to the web services scenario.

The advantage of using executable specifications for runtime monitoring is that they can be executed in isolation, even before their implementations exist. Our approach fosters the reuse of specifications for further purposes, thanks to its integration in the ASMETA framework [2], which supports editing, type checking, simulation, model review, formal verification, and test case generation for ASMs.

Moreover, although disputable, we claim that operational specifications are sometimes easier to write than declarative ones. A detailed analysis of the advantages/disadvantages of using operational specifications instead of declarative ones is given in [1].

On the basis of these advantages and of our experience in the context of monitoring of Java programs, we think that using operational specifications for runtime monitoring is a research area that is worth investigating. Currently we can monitor only single components, and we can not perform monitoring of a composition of components and the communication among components. As future work we plan to extend our approach to deal with these aspects of CBSE.

We also plan to improve our technical framework by finding more efficient solutions for the simulation of ASMs to speed up the monitoring process.

8. REFERENCES

[1] P. Arcaini, A. Gargantini, and E. Riccobene. CoMA: Conformance Monitoring of Java programs by Abstract State Machines. In *Proc. of RV 2011*, Berlin, Heidelberg. Springer-Verlag.

[2] P. Arcaini, A. Gargantini, E. Riccobene, and P. Scandurra. A model-driven process for engineering a toolset for a formal method. *Softw., Pract. Exper.*, 41(2):155–166, 2011.

[3] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, and Et. Web services conversation language (wscl) 1.0. Technical report, March 2002.

[4] M. Barnett and W. Schulte. Runtime verification of .NET contracts. *The Journal of Systems and Software*, 65(3):199–208, Mar. 2003.

[5] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[6] F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In J. Davies, W. Schulte, and M. Barnett, editors, *Formal Methods and Software Engineering*, volume 3308 of *LNCS*, pages 357–372. Springer Berlin / Heidelberg, 2004.

[7] S. Colin and L. Mariani. Run-time verification. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*, pages 525–555. Springer Berlin / Heidelberg, 2005.

[8] A. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In *Proc. of FASE'07*, pages 336–351, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Soft. Eng.*, 30(12):859–872, 2004.

[10] A. Gargantini, E. Riccobene, and P. Scandurra. A metamodel-based language and a simulation engine for Abstract State Machines. *Journal of Universal Computer Science (JUCS)*, 14(12):1949–1983, 2008.

[11] C. Ghezzi and S. Guinea. Run-time monitoring in service-oriented architectures. In *Test and Analysis of Web Services*, pages 237–264. 2007.

[12] Z. Haiteng, S. Zhiqing, and Z. Hong. Runtime monitoring Web services implemented in BPEL. In *URKE 2011*, volume 1, pages 228 –231, aug. 2011.

[13] S. Halle, T. Bultan, G. Hughes, M. Alkhalaf, and R. Villemaire. Runtime verification of web service interface contracts. *Computer*, 43(3):59 –66, 2010.

[14] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall International, Englewood Cliffs, NJ, 1998.

[15] K. Kähkönen, J. Lampinen, K. Heljanko, and I. Niemelä. The LIME interface specification language and runtime monitoring tool. In S. Bensalem and D. A. Peled, editors, *Runtime Verification*, volume 5779, pages 93–100, Berlin/Heidelberg, 2009. Springer.

[16] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31:1–38, May 2006.

[17] M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009.

[18] H. Liang, J. Dong, J. Sun, and W. Wong. Software monitoring through formal specification animation. *Innovations in Systems and Soft. Eng.*, 5:231–241, 2009.

[19] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[20] I. Sommerville. *Software Engineering*. Addison Wesley, 9th edition, 2010.