

A Systematic Approach to Runtime Verification

Seyed Morteza Babamor, Mohammad Amin Alipour
Department of Computer Engineering, Faculty of Engineering
University of Kashan, Kashan, Iran
{babamir,alipour}@kashanu.ac.ir

ABSTRACT

In this paper we propose a three-step aspect-oriented method to contribute the automatic runtime analysis of requirements. In the first step the property of interest is encapsulated by some aspects and for each aspect a verifier rule as ECA rule is presented. Then, in second step, for each aspect, the corresponding crosscut points of program is considered which correspond to property of interest. At third stage, the program is monitored to observe concrete runtime events and to verify them by rule verifiers whether there exist a match of ones. If the rule matches, totally or partially, program's behavior is analyzed. ECA rules bridge the gap between abstract properties and runtime events.

1. INTRODUCTION

Software verification methods could be divided into two broad categories: static verification and dynamic verification. Static verification tries to verify required properties in all of the possible program behaviors by theorem proving or model checking. But difficulties of specification and model construction of huge and complex software on the one hand, and leaving some properties undecided in theorem proving or state explosion in model checking in the other hand, are the biggest barriers to using formal methods.

Dynamic testing, testing programs with test data, is capable of examining finite set of program's behaviors (analyzing all of the program behaviors is not possible) and as Dijkstra's stated in [13], "Program testing can be used to show the presence of bugs, but never to show their absence".

Along with static and dynamic verification, in safety critical systems, the actual behavior of the system in runtime should be monitored to ensure the equality of system behaviors and requirement properties. Runtime verification has been proposed as lightweight formal verification methods [15] aiming to investigate systems against their requirements while execution. This technique results in validity requirements properties and steering of program in runtime and decides about current execution of program not about all the executions.

Each of the requirement properties is an interest of a stakeholder that implementations of them are spread among several modules, producing tangled representations that are difficult to understand, maintain, and evolve. Aspect-oriented concept provides approaches for the identification, separation, representation, and composition of crosscutting concerns in modules of implementation. Traditional approaches such as, to software development are unable to handle such concerns adequately, being incapable of modularizing concerns but Aspect-Oriented Programming (AOP) [18] is an emerging programming paradigm providing explicit support for separation of concerns. A crosscut relates different program points or execution points.

In a dynamic software environment, reactive program exhibits *active behavior*, i.e., functionality that is executed whenever certain requirements on the program are met. Defining active behavior is facilitated by *Event-Condition-Action* (ECA) rules.

ECA rules are the correct behaviors patterns for aspects (i.e. requirement properties) and therefore they will be the analyzers for program behaviors [24]. These rules specify for each action (process) it's triggering event and its guarding condition. The action is executed when the triggering event occurs, if and only if the guarding condition is fulfilled at that time.

In addition to use of ECA rules for representation of correct behavior patterns, they could be used as means for event generation logic into the functional units of the program. We apply notion of event-based runtime environment that are able to generate events at those points in the functional units running within them that are specified as join points during aspect deployment. Such functional units are able to emit events at certain points in their executions. Afterwards, the ECA rules monitor the events.

The remainder of this paper is organized as follows. In section 2, we explain necessity of runtime verification and the state of the art. In the section 3, we express our framework for automate runtime verification. In section 4 we show how to use aspectation for relating requirement properties into crosscut points in program. Since we consider runtime environment as an active system, in section 5 we propose monitoring and verifying the properties by ECA rules. In section 6, at first, we state some known proposed approaches of monitoring and then we explain our automated monitoring source code instrumentation by aspectation and integration of monitor program and monitored program. Finally, in this section, we use UML Activity diagrams and State charts to visualize workflow of monitor and then we generate ECA rule, i.e. execution monitor from them. In section 7 we propose an abstract data type, stack, as a case study and verify required properties. For this purpose, we apply steps of our approach to it. Section 8 presents further works and conclusion.

2. RUNTIME VERIFICATION

Analysis techniques are generally partitioned into verification and validation techniques. Formal languages like algebraic specifications or CSP apply verification techniques such as model checking and theorem proving. Despite of the relative maturity of formal verification within software engineering research, practical applications are limited to safety-critical and embedded systems, i.e., systems with a high penalty of failures. Reasons for this include the complexity of formal specification techniques and the lack of training of software engineers in applying them. Furthermore, there are also well-known limitations of formal verification such as the state-explosion problem within model checking.

In contrast to formal verification, where a property may be assured with mathematical rigor [15], validation techniques may detect errors or improve our confidence in the implementation, but they cannot prove any property in a definite way. The classic technique for validation of properties in software engineering is testing. Validation techniques such as testing is usually applied

to the *actual system*, checks whether an implementation conforms to some design (i.e., informally, has the same behaviors). Furthermore, the whole system may be very large while we are interested only in specific aspects of it. We want to check that the implementation of a particular feature meets certain correctness properties. Testing relies on the construction of test strategies for a property including subsequent execution of parts or all of the system according to these strategies. As the testing takes place on a lower level of abstraction, the range of properties that can be validated is much greater than using formal verification.

Runtime verification and monitoring have been proposed as lightweight formal verification methods with the explicit goal of checking systems against their formal requirements while they execute. This technique (1) bridges the gap between above-mentioned techniques, i.e., formal verification specifications and testing of the implementation, resulting in validity requirements properties, and steering of program in runtime, (2) decides about current execution of program not about all the executions. In [2], figure 1, we have sketched functions of runtime verification that

not fulfilled by formal verification and implementation testing. As the figure shows, analyzing of runtime decides about properties those (1) were left undecided in verification specifications, (2) not detected in testing of implementation and (3) are closely related to physical environment in that thoroughly conditions in not known in advance.

3. FRAMEWORK

We provide a framework for automated support of requirement properties and then explore approaches within it. Figure 2 shows the framework. In this framework,

1. Requirements properties are categorized into the functional and non-functional (quality) ones, and then provide functional and quality aspects, respectively,
2. Aspects are used for purpose of:
 - 2.1. Producing ECA rules,
 - 2.2. Mapping them to corresponding points of program,
3. Execute, observe, and verify program behaviors.

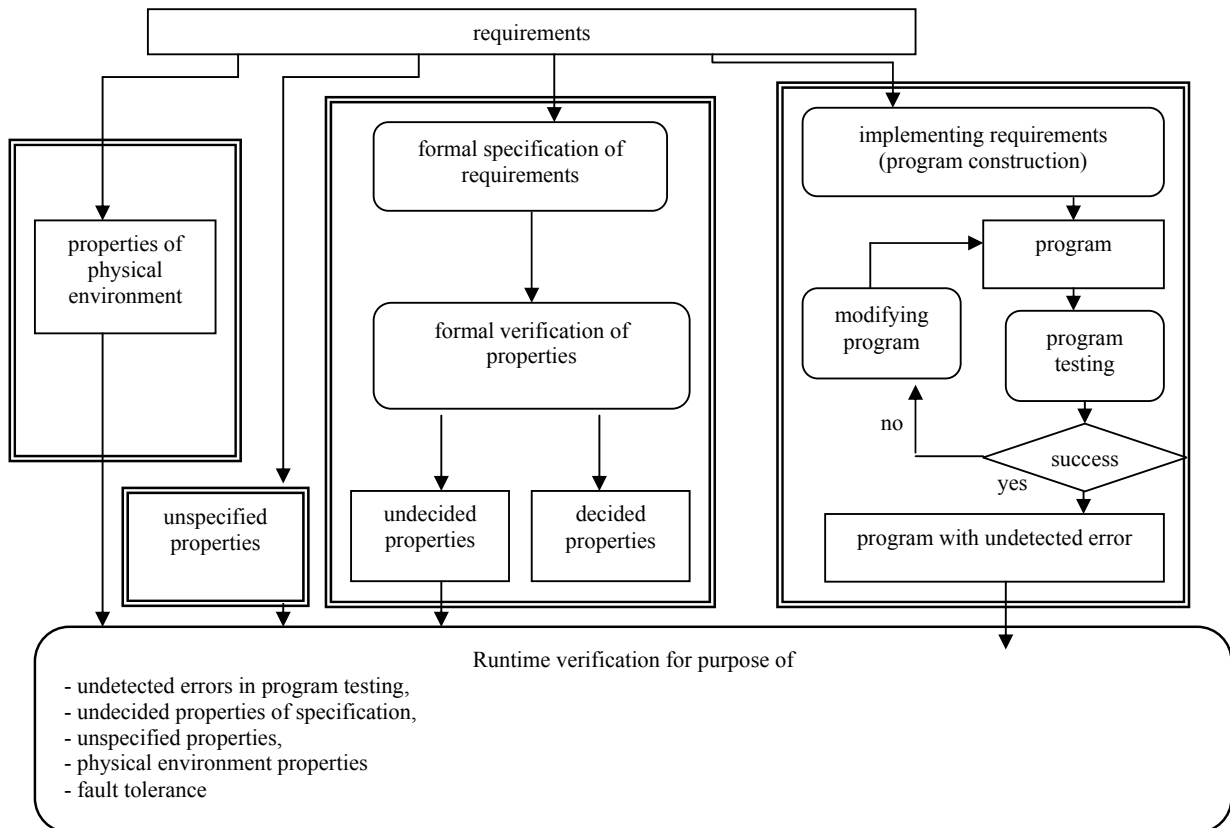


Figure 1. Functions of runtime verification that not fulfilled by formal verification and implementation testing

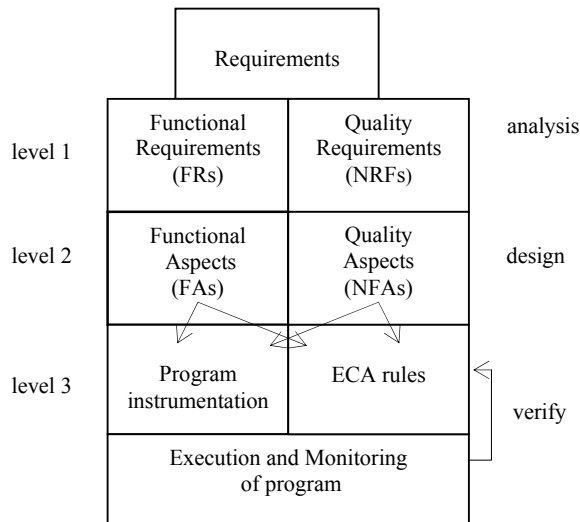


Figure 2. Framework for automated supporting requirements

4. ASPECTATION

During the requirements elicitation, while identifying the user requirements, the system is described in terms of functional and non-functional requirements. For example, a user may need that a bank account should support deposits and withdrawals and that the system should react to the account's owner requests in a short period. That is, on the one hand, the required functionalities are deposits and withdrawals, and on the other hand, we are explicitly concerned with quality of this function, i.e. response time.

Aspectation holds promise for the creation of aspects, which are modules that centralize distributed functionality. So, one of the original motivations of aspectation is to narrow the gap between analysis/design and programming.

For each requirement, we know that, there exist indispensable software properties. We call these properties as aspects. For example, when we are to examine the performance, we may consider from an aspect related to data access or from an aspect related to communications. Likewise, in examining a quality attribute from an aspect, we have typical factors we have to consider. For example, when we focus on the aspect relates to data access, we may examine the quality attributes based on the number of data, data size, access pattern and etc.

We have two intentions for aspectation. Level 3 in figure 2 shows these intentions. First, we determine aspects of the property to instrument the program by them automatically. Figure 3 shows instrumentation of program by aspectation of requirement properties. Instrumentation pre-programs event generation logic into the functional units of the program. Such functional units are able to emit events at certain points (is specified as join points) in their executions. In corresponding with AspectJ [19], the first tool provides linguistic support for the modularization of crosscutting code, we call this execution points "join points" those can be used as triggers. Join points could be method calls, exception handlers, or other points in the execution of a unit. A language construct that identify and select join points within the units, is called "pointcut" and the code to be executed when a join point is reached (after or before) in the unit, is called "advice". Therefore, an aspect encapsulates join points, pointcuts, and advices.

Afterward, we use aspects as correct behaviors pattern. So, we generate ECA rules for aspects. These rules have verifier role for program behavior based on emitted events.

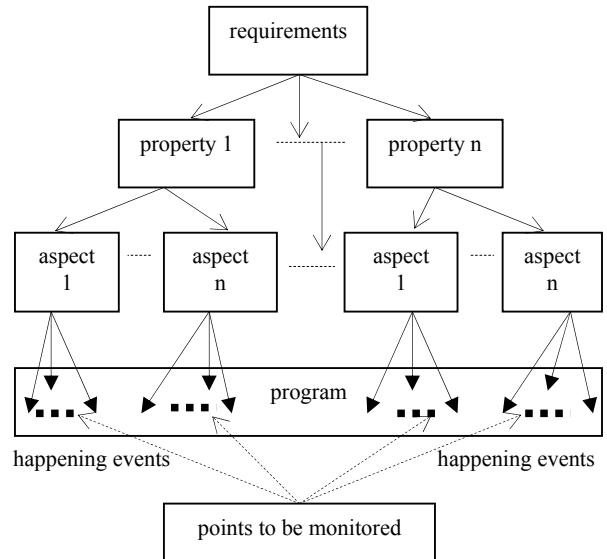


Figure 3. Instrumentation of program by aspectation of properties

5. ECA RULE

Events are objects, which represent execution points of program, such as method invocation and return events. An aspect is the code takes an event as parameter and monitors it. Complex systems especially in reactive ones continuously respond to external and internal stimuli (events), so the designer should determine the system's behavior regarding these events, as well as its flow of control. A common way for expressing control flows is via Event-Condition-Action (ECA) rules. The ECA model introduced in HiPAC [8] is widely used to support rules in active database systems. An active database is a database that has active behavior. Active behavior, capable of reaction to events, is functionality that is executed whenever certain requirements on database are met. This behavior is defined by ECA rules. These rules specify for each action (process) its triggering event and its guarding condition. The action is executed when the triggering event occurs, if and only if the guarding condition is fulfilled at that time. A general definition of the ECA model can be found in [23], where *Event (E)* is a primitive (basic) or composite event, *Condition (C)* is a Boolean expression and *Action (A)* is an action that must be executed. The need for modeling ECA rules with logical events was addressed in [7]. Complex events can be formed from simple events. For example composed event (E1|E2) means that one of the two events E1 or E2 must occur, and composed event (E1:E2) means that the two events must occur in the given order. In this paper, we specify how produce ECA rules from design model and based on the rules we analyze runtime behavior of object-oriented program.

Based on our framework, a system that is built on top of this framework consists of several instrumented functional units and a set of ECA rules. Functional units in such a system are able to emit events at certain points in their executions. ECA rules define how and under which conditions to monitor and verify behavior of the functional units at these points.

6. MONITORING AND CHECKING

Dynamic analysis is the study of program execution behavior. This analysis widely used in debugging and performance tuning. Some researchers proposed dynamic analysis of program behavior so far. Meyer proposed *Design by Contract* approach

for the object-oriented language Eiffel [21] that is a lightweight formal technique and allows for dynamic runtime checks of specification violation. The specification of the contract is directly written into the program in the form of assertions. Ideally, the syntax of assertions is close to the programming language itself and thus easy to use for all programmers and those are checked during program execution. Design by Contract extensions have been proposed for a number of languages besides Eiffel, such as Ada and C++. While trace assertions in the Design by Contract are the counterpart of specification like process algebras or temporal logic, they are used to monitor the dynamic behavior of an object, the ordering of method invocation and calls in time [7]. There are automated trace analyzers that is connected to an event-oriented tracer. The traced program runs in parallel with a trace analysis session in which the user enters high-level queries about the traced execution. Then, the trace is processed automatically according to the query.

The monitoring, checking and steering (MaC) framework [20] is another technique which has been designed to ensure that the execution of a real-time system is consistent with its requirements at run-time. It provides a language, called MEDL, to specify safety properties based on linear temporal logic. The safety properties include both computational and timing requirements. These properties are defined in terms of events, conditions, auxiliary variables, and auxiliary functions. Finally, some other focused on detecting likely program invariants [12], rather than verifying properties directly, which can then be used to reason about programs or in error detection.

6.1. Automated Monitoring

In our approach, inspired by the active database that exhibits active behavior, i.e., functionality that is executed whenever certain requirements on the database are met, we consider runtime environment as an *active system* that waits for events to happen. Since *active* property causes an event caught automatically by corresponding ECA rule, it helps to automate runtime monitoring and checking. Set of ECA rules form an execution monitor. The execution monitors have a global view on the program execution behavior and are used for tasks that require an understanding of program behavior, such as debugging and profiling. These monitors are highly appropriate for natural abstraction in terms of events, and enable the explicit definition of complex crosscuts by means of event patterns, and do actions.

6.2. Integration

There are three models to integrate the monitor and the monitored program together: the *one-process model*, the *two-process model*, and the *thread model*. In the one-process model, an execution monitor is a library of procedures linked to the program being monitored or integrated into the run-time system. The one-process model has good performance and access characteristics, but it does not separate the target program and monitor and they may interfere each other and cause unpredicted result. In addition, the control flow logic within the monitor is ambiguous because the monitor is activated through callbacks. In the two-process model, the monitor is a separate process to reduce the problem of interference but the monitor access is more complicated and performance is degraded. In the thread model, the monitor is a separate thread in a shared address space occupied by the program and possibly other monitors, providing a reasonable compromise between the characteristics of the one-process and two process models for many monitoring applications. Interaction facilities vary both in terms of execution

controls provided to the user and the techniques used in presenting the user with execution information. Execution controls vary from controls that can only start and stop execution to entire languages that can be used to query for execution information or modify program variables.

6.3. Visualization

Runtime issues in dynamic analysis always need to trade-off between the low-level issues of instrumenting the system under observation with the high-level issues of making the customization of analysis accessible to the user. A variety of solutions have been proposed and implemented, from special purpose systems that only allow a specific class of analyses to be performed, to special languages (e.g., event processing languages[1] and ...) that could be used to specify the desired analysis. Opium[10], Morphine[17] and Coca[11] are three automated trace analyzers, for three different programming languages: respectively Prolog, Mercury and C.

Another approach is the use of visual and interactive methods. The study of data display techniques used in execution monitors has developed into its own sub-field: *program visualization* refers to the use of graphics to depict execution-monitoring information. Algorithm animation is well-known use of program visualization. That it is easier to illustrate software issues using a visual formalism, as demonstrated by UML diagrams. With development of the Unified Modeling Language in recent years, model-based development is becoming more widely accepted within industry. In [3], by verifying reactive programs in runtime using Interval Temporal Logic, we proposed an approach for model-based analysis of properties by UML-based development process and in [17] we suggested an oracle for runtime monitoring of reactive programs.

In this paper we divide properties of a concern into safety and liveness ones. Safety properties assert that something bad never happens, while liveness properties assert that something good will eventually happen. The classification of properties into safety properties and liveness properties allows us to choose the most appropriate verification method for showing correctness with respect to these properties. For example, methods based on global invariants have been extensively used for safety properties, while methods based on well-founded induction have been employed for liveness properties. When checking safety properties, the behavior of a system can be described by a finite state automaton. Temporal logics (TL) are currently the most widely specification formalism for specifying safety and liveness properties. Some examples of practical use of TL formulas in specification are: safety property: $\Box \neg(cs_1 \wedge cs_2)$ (it always holds that two processes are not at the same time in a critical section) and (2) liveness property: $\Box(req \rightarrow \Diamond ack)$ (it is always the case that a request is eventually followed by an acknowledgement).

For verifying property in [2] that illustrated in figure 4, we create one rule-based system from UML state machines in which rules are in the form of ECA. UML state machines could be used as a stand-alone behavioral description. These machines are showed in Statechart and Activity diagrams. Those are an extension of the conventional formalism of state machines and are based on states, which specify a situation in which a system (or an object) exists, and transitions, which enable the system to move from one state to another. A transition can be expressed in the form of an ECA rule, i.e., syntactically formulated as "event [condition] | action". The need for modeling ECA rules with logical events was addressed in [8]. The rules are notified about events that have occurred and the conditions, which gives the necessary and sufficient conditions for the event to be generated (Event-Condition). Another approach is proposed in [8], which used the Event algebra in ECA rules. In this paper, with separating safety

7.2. Aspectation

The concern for access has three aspects, (1) modifying the stack, (2) visiting the stack, and (3) existence the stack. Modifying aspect crosscutted in push and pop units, visiting aspect in peek unit and existence aspect in create and destroy units. Consider figure 2, The requirement (i.e. the concern) is access to stack, property 1 is two above-mentioned *safety* properties, aspect 1 is modify aspect, and points of interest to be monitored are push and pop units. Therefore, the aspect, join points, and pointcut are defined by AspectJ language as follows:

```
Public aspect ModifyAspect {           // the Aspect
    Pointcut UpdateStack ( ) :
        call (public void stack.push(string s); // join point 1
        before() : UpdateStack ( ) {
            if stack.state == full
                request.change(reject);           //overflow is prevented
                                                    //and state of request object
                                                    //is set to rejected
        call(public string stack.pop(void); // join point 2
        before() : UpdateStack ( ) {
            if stack.state == empty           //underflow is prevented and
                request.change(reject);       //state of request object
                                                    //is set to rejected
```

7.3. The Monitor

First, we use UML diagrams for visualizing properties, and then generate ECA rules. Figure 6 shows Activity diagram for safety property 1 (i.e. overflow prevention) of modify aspect and related ECA rule, figures 7, 8 show Statechart diagram for changing states of request and stack objects.

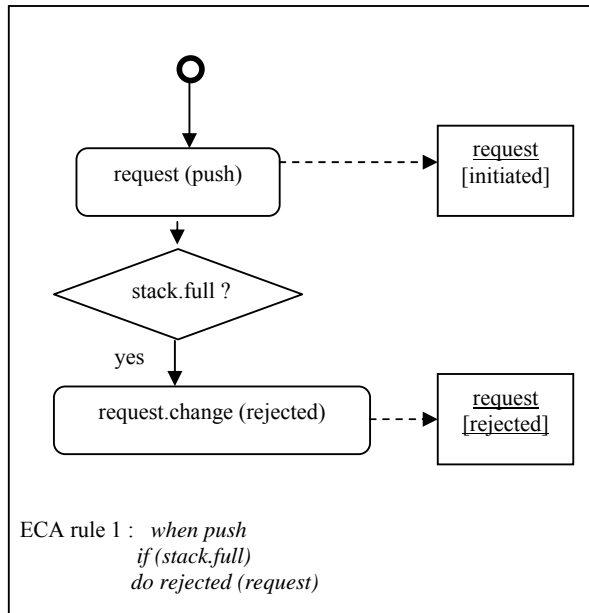


Figure 6. Activity diagram for safety property 1 of modify aspect (overflow prevention) and related ECA rule

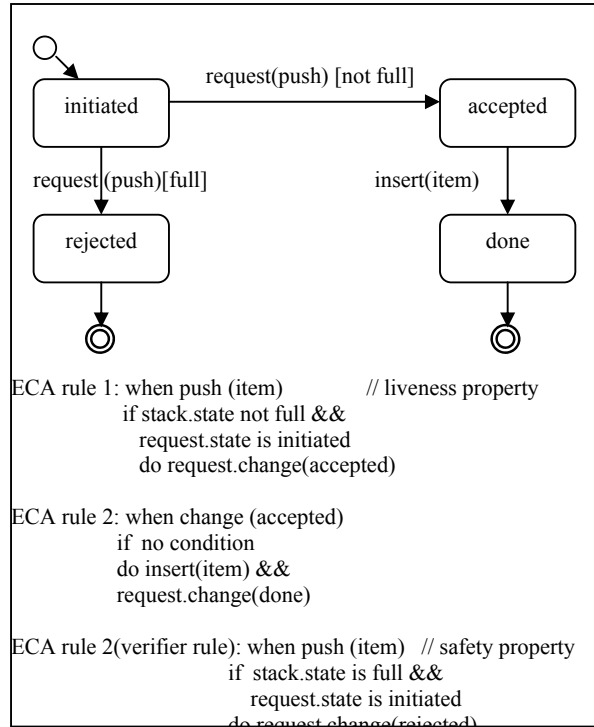


Figure 7. Statechart diagram for push join point of modify aspect related to request object

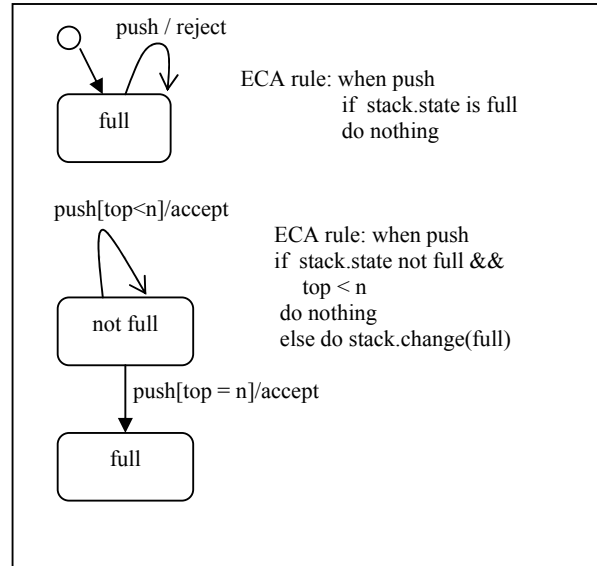


Figure 8. Statechart diagram for push join point of modify aspect related to stack object

Therefore, execution monitor for push unit of modify aspect formed by the generated ECA rules in figures 6, 7, 8. When a request (an event) is occurred that is related to modify aspect such as push event, (1) the safe guard ECA rule (in figure 6) of execution monitor rejects the request if it causes overflow, (2) the reset of ECA rules of execution monitor (in figures 7,8) changes state of request and stack objects. Similarly, one

visualize safety property 2 of modify aspect for pop unit and generate the related ECA rule. In addition, we can visualize liveness properties.

8. FUTURE WORK

Our approach is usable for structured and object-oriented programs. We have started to equip and provide the approach with distributed and real-time system constraints. In a distributed system, we must consider distributed aspectation. Distributed aspectation can be performed based on notification services in network-based systems or collaboration work in distributed environments such as internet [4]. Thus, join points are distributed in clients' program. Each event that happens in distributed environment is multicasted through the environment to members of group that the members subscribe to the events. Network server performs monitoring and checking. Thus, some threads on clients emit events to the server while one thread on server monitors the emitted events.

In real-time systems, we must consider time constraints of the system. Therefore, first, we must specify requirement properties in terms of timed specifications such as temporal logics. Second, using a known approach such as tableau method, we transform and visualize time specifications in automata. We have some experiments in the context. In [5,6] we expressed some properties of CBCAST protocol in FIL¹ and GIL² [16] and then monitor them by concurrent and hierarchical automata, i.e. Harel's Statecharts [14]. CBCAST protocol is a communication protocol that is applied to reliable communications. Properties of the protocol is verified by *virtual synchrony* concept [22], ensuring message ordering is done in a low cost of time and correctly. However, the work to be must done is to equip generated ECA rules with timed events and conditions.

9. CONCLUSION

The paper presented an approach to automatize runtime verification by aspectation. The main feature our approach is exploiting Aspect-oriented concept that is main contribution to automated support of runtime verification. This contribution is acquired in two ways. The first one is help to build execution monitor in the form of ECA rules. Aspect-oriented concept and UML diagrams together, helped to visualize and generate execution monitor. The approach is based on the automatic creation of monitoring rules from state machines, specifying the properties of design abstracts in a proposed Event-Condition-Action rule. The second one is bridge the gap between abstract specification of requirements and low-level implementation. The big challenge for runtime verification is wide gap between high-level requirements and low-level points of the implementation. One of the promising advantages of this approach is in its ability to bridge the gap between design abstracts and low-level events in execution of the program. Another advantage is that in terms of which part of the rule(s) is satisfied, a variety of possible behaviors of program can be analyzed. Verification of some typical behaviors has showed in one case study. Our future work applies the approach to distributed systems as well as real-time ones. In addition, we consider monitoring of quality attributes of requirements.

REFERENCES.

- [1] Auguston, M., Gates, A., and Lujan, M. Defining a program Behavior Model for Dynamic Analyzers. In Proceedings of

the 9th International Conference on Software Engineering and Knowledge Engineering, pages 257-262, IEEE Computer Society Press, June 1997.

- [2] Babamir, S.M. and Jalili, S. Dynamic Analysis of Object-Oriented Programs Using State Machines and ECA Rules. To Appear in 14th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE-05), July 20-22, Toronto, Canada, July, 2005.
- [3] Babamir, S.M., and Jalili, S. Monitoring and runtime verification of reactive programs using Interval Temporal Logic, In Proceedings of 13th International Conference on Electrical Engineering, Zanjan University, Iran, May 2005.
- [4] Babamir, S.M. Design and Prototyping of a Integrated Environment for Software Development Based on Web. M.S. Thesis, Tarbiat Modares University, Computer Department, Iran, 2001.
- [5] Babamir, S.M., and Jalili, S. Monitoring and runtime verification of reactive programs using Interval Temporal Logic, In Proceedings of 13th International Conference on Electrical Engineering, Zanjan University, Iran, May 2005.
- [6] Babamir, S.M., Jalili, S. Test Evaluation of Reactive Programs using Statecharts Approach, To Appear in Proceedings of 2th of International Conference on Information and Knowledge Technology Engineering (IKT2005), Amirkabir University, Iran, June 2005.
- [7] Bartetzko, D., Fischer, C., Moller, M., and Wehrheim, H. Jass – Java With Assertions, Electronic Notes in Theoretical Computer Science 55 No. 2(2001).
- [8] Chakravarthy, S., Blaustein, B., Buchman, A., Carey, M.J., Dayal, U., Goldhirsch, D., Hsu, M., Juahari, R., Livny, M. D.McCarthy, R.McKee and A.Rosenthal. HIPAC: A research project in active, time-constrained database management, Final Technical report. Technical Report XAIT-89-02, Xerox Advanced Information Technology, August 1989.
- [9] Clarke, L.A., and Naumovich, G. Classifying properties: an alternative to the safety-liveness classification, Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering: twenty-first century applications, Pages:159-168, 2000.
- [10] Ducasse, M. Opium : An extendable trace analyzer for prolog, The Journal of Logic Programming, 39:177 223,1999.
- [11] Ducasse, M. Coca : An automated debugger for C. In Proceedings of the 21st International Conference on Software Engineering, pages 504-513. ACM Press, May 1999.
- [12] Ernst, M.D., Cockrell, J., Griswold, W.G., and Notkin, D. Dynamically discovering likely program invariants to support program evolution, In International Conference on Software Engineering, pages 213–224, 1999.
- [13] Gahl D.J., Dijkstra E.J. and Hoare C.A.R. Notes on Structured Programming. Academic Press London, pp.182, 1972.
- [14] Harel, D., and Politi, M. Modeling Reactive Systems with Statecharts, McGraw-Hill, 1998.

¹ Future Interval Logic

² Graphical Interval Logic

- [15] Havelund, K. and Rosu, G. Runtime Verification 2001. Proceedings of a Computer Aided Verification (CAV'01) satellite workshop, Electronic Notes in Theoretical Computer Science, Elsevier Science, Volume 55, 2001.
- [16] Hornos, M.J., and Capel, M.I. On-the-fly model checking from interval logic specifications, ACM SIGPLAN Notices, Volume 37, Issue 12, pp. 108 – 119, December 2002.
- [17] Jahier, E. Collecting graphical abstract views of Mercury program executions, In Proceedings of the International Workshop on Automated Debugging, Munich, August 2000.
- [18] Kiczales, G. et al. Aspect-oriented programming. 11th European Conference on Object-Oriented Programming, Volume 1241 of LNCS, pages 220-242. Springer Verlag, 1997.
- [19] Kiczales, G., Hilsdale, E., Hugunin, J. et al. An Overview of AspectJ. In proceedings of the 15th Conference on Object-Oriented Programming (ECOOP), volume 2072 of LNCS, pages 327-353. Springer-Verlag, Berlin, June 2001.
- [20] Kim, M., Lee, I., Sammapun, U., Shin, J., and Sokolsky, O. Monitoring, Checking, and Steering of Real-Time Systems. In Proceedings of the 2nd International Workshop on Runtime Verification, July 2002.
- [21] Meyer, B. Object-Oriented Software Construction. Prentice Hall, 1997, 2nd edition.
- [22] Moser, L.E., Amir, Y., Melliar-Smith, P.M., and Agarwal, D.A. Extended virtual synchrony, in Proceedings of the IEEE 14 Computing Systems, pp. 56--65, IEEE Computer Society Press, Los Alamitos, CA, June 1994.
- [23] Pissinou, N., Snodgrass, R.T., Elmasri, R., Mumick, I.S., Özsu, M.T., Pernici, B., Segev, A., Theodoulidis, B., and Dayal, U. Towards an Infrastructure for Temporal Databases: Report of an Invitational ARPA/NSF Workshop, ACM SIGMOD Record, vol. 23(1), pages 35-51, March, 1994
- [24] The ACT-NET Consortium. The Active Database Management System Manifesto: A Rule base of ADBMS Features. ACM SIGMOD Record, 25(3):40-49, 1996.