RUNTIME VERIFICATION OF PROGRAMS USING RULE-BASED ACTIVE SYSTEMS

SEYED MORTEZA BABAMIR
DEPARTMENT OF COMPUTER ENGINEERING
UNIVERSITY OF KASHAN, KASHAN, IRAN
E-MAIL: BABAMIR@KASHANU.AC.IR

ABSTRACT. Runtime verification, monitoring and checking runtime behaviors, plays an increasingly important role to realize tasks that have become inefficient with static verification and implementation testing. In this paper, we propose a new approach based on a framework to dynamic verification programs. The framework exploits capabilities of active systems for verifying programs. Active systems are ones those act based on occurrence of events and therefore it facilitates trapping events and monitoring program's states. The active system is a rule-based system and we use ECA rules to show active rules in the active system. Thus, defining active behavior is facilitated by ECA rules. ECA rules are verifier rules in form of event-condition-action. These rules, forming runtime monitor, verify behavior of system whenever an event occur during program execution. Exploiting active systems as event based environments and using ECA rules in context of them, are main contributions of our approach to runtime verification. An active system, forming runtime environment, sets a trap to catch runtime events and then check them by the ECA rules. We apply our approach to a classical Abstract Data Type (ADT), stack, and express how one can use an active environment to verify safety properties of a stack.

AMS Classification: 68N17, 68N99.
Keywords: Run-time Verification; ECA Rule; Active Database; Aspect-Oriented

---

JOURNAL OF MAHANI MATHEMATICAL RESEARCH CENTER
VOL. 1, NUMBER 1 (2012) 77-95.
©*MAHANI MATHEMATICAL RESEARCH CENTER*

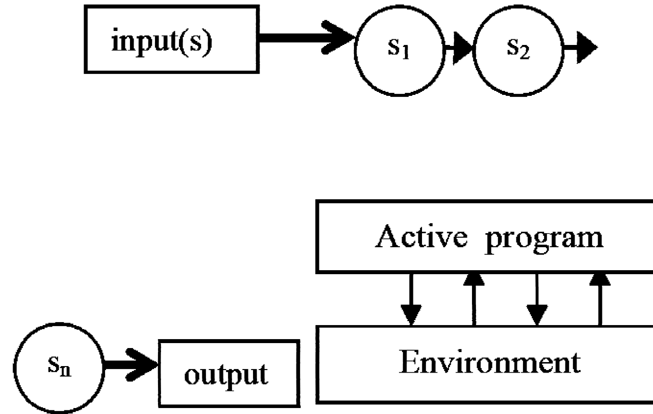


Figure 1. Passive (algorithmic) and active programs.

1. Introduction

Programs acts passively or actively. In passive programs such as compilers, an algorithm takes some input value(s), computes output value(s) during finite steps and then stops. Active programs usually monitor their environments for events and give feedback to the environment when they observe some event. Accordingly, unlike passive programs, they never terminate. Fig. 1 shows passive and active programs where s_1, s_2 and s_n indicates steps of the program algorithm. Typical active programs are process control, power plants; embedded systems in trains, aircrafts and traffic-light controller. Active programs typically interact with its environment to give feedback for some event.

To verify an active program behavior, we should verify the program reaction to an environment event. Emergence of active applications such as control applications shows the importance of dealing with them. Active programs act interactively and reactively where interactive programs such as operating systems and web server interact with their environment at their own speed. In contrast, Reactive ones

interact with their environment at environment speed [1]. Although interactive programs are usually non-deterministic, reactive ones are generally deterministic. In other words, although all executions of a reactive program may be infinite sequence of states, an execution includes finite states. This means that the output values of a reactive program are determined by specific inputs.

The idea of activeness is taken from active databases [2] where activeness is stated as a set of related production rules in form of Condition-Action. Different mechanisms in Active Database Management Systems (ADBMS) were stated [3] where active rules are defined as Event-Condition-Action (ECA) rules. Event such as change to data indicates the time of firing an event, Condition indicates evaluation of some predicate(s) on firing the event and the Action is executed when the event fires and the condition is satisfied. An action in an ECA rule may be fired instantly, with a delay such as commitment of a transaction or casually. In causally firing, an action is executed in a separate sub-transaction that waits until the main transaction is committed.

Operations such as a database update leading to change to data generate events that may be monitored by ECA rules. Events are synchronous generated by users, applications, or asynchronous such as changes of sensor values or time. Publish/Subscribe technology is a typical event-driven method where a publisher sends relevant information to subscribers in reaction to a subscription [4, 5]. This technology is significant because pushing information to clients is one important aspect of Internet-based information systems. Accordingly, publish/subscribe is embedded within many Web applications for supporting deliverance of information in response to users' interactions. Among others, active rules play the role of supporting this reactiveness.

The paper is organized as follows. In Section 2, we explain ECA rules and their roles in an active system. In Section 3, we deal with discussing runtime verification. In Section 4 we propose our model and show how aspect-oriented method can be used for equipping the target program should be verified and in Section 5, we explain implementation of the proposed model. To show effectiveness of our method, in Section 6 we state a case study and apply our model to it. Finally, in Section 9 we states conclusions and related work.

2. ECA Rules

The first mechanism used to provide automatic reaction was production rules. The rules are stated in form of Condition-Action. When monitoring events in a passive system, a polling technique can be used to determine changes to data. With the polling method, the application program periodically polls the system by placing a query about the monitored data. The problem of this approach is that the polling should be fine-tuned so as not to flood the system with too frequent queries that mostly return the same answers, or in the case of too infrequent polling, the application might miss important changes of data. Operation filtering is based on the fact that all change operations sent to the system are filtered by an application layer that performs the situation monitoring before sending the operations to the system. The problem with this approach is that it greatly limits the way rule condition evaluation can be optimized. It is desirable to be able to specify the conditions to monitor. By checking the conditions outside the system, the complete queries representing the conditions will have to be sent to the system.

Event-Condition-Action rules have been used to provide reactive functionality in many settings, including active databases [3], workflow management, network management, personalization and publish/subscribe technology [6] and specifying and implementing business processes [7].

In rule-based systems, the active rules can be used for purposes of monitoring, control, and reasoning. In active database systems, the rules are primarily used for monitoring changes to the data stored in the database. In reactive systems the rules are used for reacting to changes of some external environment and performing actions on (controlling) the environment in response to the changes. In knowledge-based systems, the rules are usually used for reasoning using stored facts and by deducing new facts by using the rules. Active rules can serve as a complement to traditional coding techniques where all the functionality of the system is specified in algorithms written in modules and functions. Active rules provide a more dynamic way of handling new situations and are often better alternatives to modifying old functions to cope with new situations. A common technique that is used is to use rules for specifying parts of the system during the design phases and to use these rules as guideline for the actual coding phases or to compile the rules into corresponding functions to simplify the coding. This last technique is sometimes

found directly supported in some programming languages such as Eiffel [8] where pre-conditions and post-conditions on data can be specified. If the conditions are violated an error is generated. The rules can signal to the user or some application that a condition has been violated. Rules can also specify actions to be taken, such as removing inconsistencies by changing illegal values of data. In most programming languages fault handlers can be defined that catches error signals. Rules in an active system can be seen as having similar behavior, but catches events.

3. Runtime Verification

To check the correctness of software, verification and validation method are used. Formal verification methods use formal languages like the Z language for specification and use verification techniques such as model checking to check the specification. However, they are not become scalable and well-suited for some applications like safety-critical systems. This is because (1) formal specification and verification is complex to use and (2) there suffer limitations such as the state-explosion problem when one aim to use model checking. These issues and other ones justify using runtime verification (Fig. 2) The figure shows the issues that can not be resolved by conventional methods, i.e. formal verification and software testing.

By formal verification, we can check satisfaction of a property with mathematical rigor and by validation methods such as software testing, we may detect errors or improve our confidence in the implementation. However, The validation techniques are usually applied to the actual system, checks whether an implementation conforms to some design. Furthermore, the whole system may be very large while we are interested only in its specific aspects. We want to check that if a software implementation satisfy some 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 method is a lightweight formal verification method with the goal of checking software against their formal requirements at runtime. This method has advantages: (1) bridges the gap between formal verification and software testing methods, resulting in validity requirements properties and

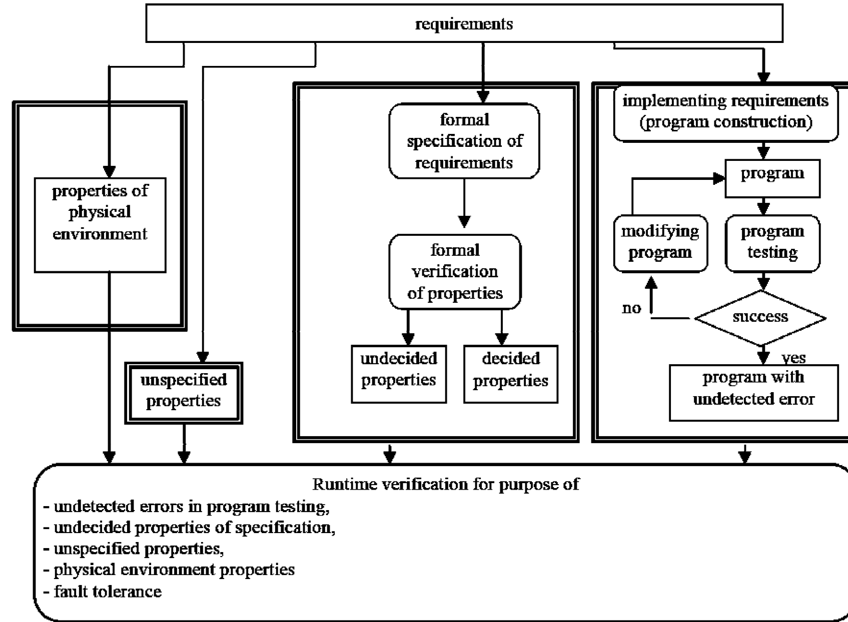


FIGURE 2. Runtime verification tasks.

steering of programs at runtime, (2) decides about current execution of programs not about all their possible executions. According to Fig. 2, runtime verification aims to determine properties those(1) could not decided by verification, (2) could not detected by testing and (3) include conditions that closely related to physical environment in which the program would run.

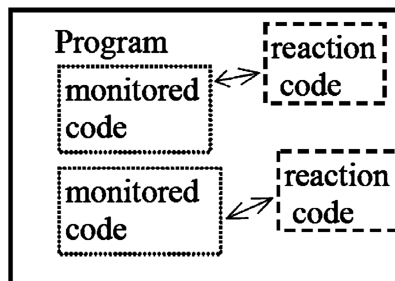
Among others, constraint verification and authorization control are functions that can use data monitoring. By constraint verification, rules can monitor, detect and abort any query that violate some constraint. By authorization control, rules can be used to check that if the user or application has permission to perform specific actions in the system. Telecommunications Network Management and Financial

Decision Support System Applications are typical systems that depend on data monitoring activities.

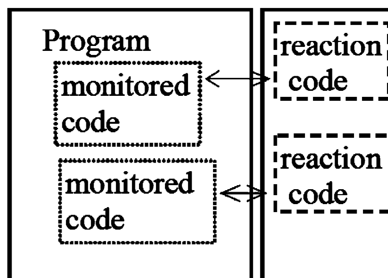
A monitor can be connected to the target program in one to two stages. In the one-stage approach an execution monitor is a library of procedures linked to the target program or it is integrated into the runtime system. This approach has good performance; however, it is invasive, i.e. by this method, the monitor code is weaved into the target program and thus it affects the target code. In addition,because the monitor is activated through callbacks, the control flow logic within the monitor is somewhat inverted. In the two-stage model, the monitor is a separate process from the target program, reducing the problem of intrusion at the expense of complicating monitor access and reducing performance. There is another two-stage approach performed by multithreading method. In such an approach, the monitor program is a separate thread in a shared address space occupied by the program program, providing a reasonable compromise between the characteristics of the previous models.

The target program should be weaved by monitor code. Weaving, in fact, is process of making aware the monitor program when an event occurs. The monitor, obtaining data about program's behavior, can be considered as invasive or non-invasive [9]. In the invasive approach, logic of obtaining events and reaction to are weaved in source, byte, or execution code of the target program. Fig. 3a shows one-stage approach to weave the monitor code into the target program. The two-stage approach has been shown in Fig. 3b, in which the reaction code is separated from the target program and managed by a separate entity. The thread model has been shown in Fig. 3c.

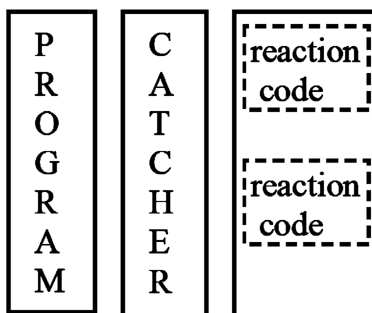
The weaving mechanism for invasive approach can be performed in several ways. The most common ones, despite its limitations, is manually insert the monitor code into the target program. In this method, the program user reads the source code of the target program and manually weaves probe codes into the target program, which is time consuming and may be incomplete. For exhaustive monitoring, weaving should be complete in the sense that it should capture all the concerned codes of the target program; otherwise, incomplete weaving leads to missing some data capturing.This could lead to false or missed detection of faults.



(A) one-stage method



(B) two-stage method



(C) thread model

Figure 3. Weaving

Another mechanism is Runtime weaving referring to the modification of the target program code immediately prior to or during execution. The probe code is inserted at the executable code (e.g., byte code in Java) level. Compared to the source-code level weaving, the executable-code level weaving is complicated since the source-level weaving is useful for understanding of the program. In addition, modifying a system at executable-code level requires keeping the format of the executable code consistent.

The interpreter weaving method, which inserts the monitoring code into the language interpreter is another non-invasive weaving mechanism. Such weaving can provide data about the behavior of any program executed by the interpreter. Similarly, weaving at compiler level includes preprocessors and code generators that add the weaving code to the code they generate. Such code usually is much larger than the non-weaved code. Weaving at the Interpreter and compiler level has two problems: (1)Either we should have source code of interpreter or compiler for weaving or that it should have already been weaved and (2) Interpreter or compiler weaving is a difficult problem and the weaver needs to be involved in the understanding of source code of interpreter or compiler. On the other hand, a weaved interpreter or compiler may be not adequately matched our probe needs.

4. The Proposed Model

An active system transforms a passive process into an active environment by using alerts and triggers. In this section, we use Fig. 4 for runtime verification of the target program. The model has two parts, monitoring and verifier. As the figure shows, the monitoring part has two tasks, capturing events that occur in the program environment and taking a decision made by the verifier part. The verifier part is responsible for taking policies and valid decisions.

A course of events, environments conditions, and actions, embedding in ECA rules, can form triggering graph, as discussed in Section 2. The triggering graph represents each rule as a vertex, and there is a directed arc from a vertex r_i to a vertex r_j if r_i may trigger r_j. An action of rule r_i may generate an event which triggers rule r_j. Activation of a rule may change an environment condition. An activation graph representing rules as vertices, has an arc between vertices r_i and r_j indicates that r_j's condition may be changed from False to True after the execution

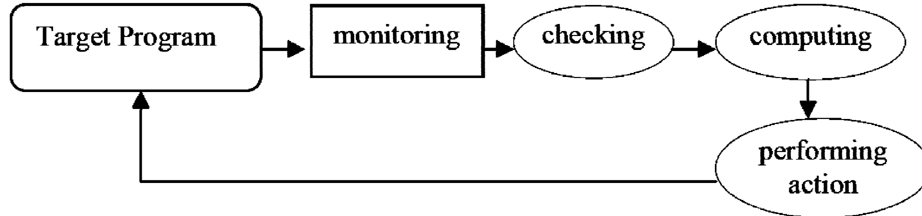


FIGURE 4. The proposed model.

of r_i's action, while an arc from a vertex r_i to itself indicates that r_i's condition may be True after the execution of r_i's action.

4.1. **Program Activation.** Our approach is different from past-related work. Our main idea is to provide an active environment to support runtime monitoring and verification. For this purpose, first we transform the program must be monitored from passive program into active one. Passive program is activated if it was well aware from occurrence of events. Awareness can be done by invasive or non-invasive approach, discussed in Section 4. Our approach uses invasive one (Fig. 3b); therefore, we weave source-code program by aspects automatically, not manually. This type of weaving has no disadvantage over the manual source-code level weaving and has no difficulty over the executable-code level one. Source-code weaving can be achieved automatically using aspect-oriented approach. By aspect, we define each property that to be verified; therefore, by aspect definition, related probe codes (i.e., crosscutting points of property) are inserted into units of the program. Aspect-oriented holds promise for the creation of aspects, which are modules that centralize distributed functionality. Fig. 5 shows using aspects for weaving the capture code into source code of the target program. Afterwards, the modified program will emit a signal when a related event occurs and the monitor will capture the event and will send it to the checking part.

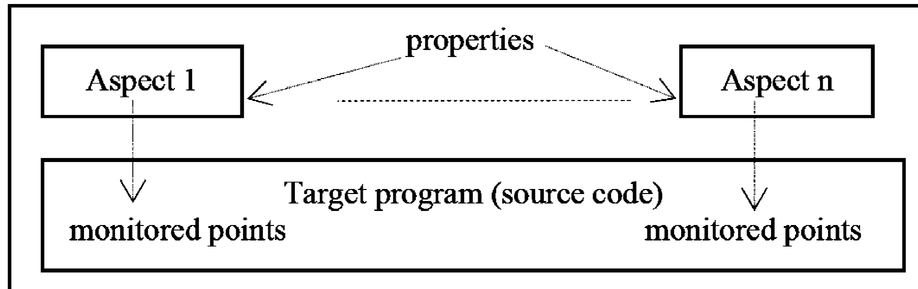


FIGURE 5. Weaving source code by Aspects.

Second, we build verifier as a rule based system. The verifier obtains events emitted by the modified program and monitors them. Events are objects, which represent execution points of program, such as method call and return events. Therefore, in an active system, we continuously respond those events that determine the system's behavior, as well as its flow of control. One common way for expressing control flows is via Event-Condition-Action (ECA) rules. The verifier, a set of ECA rules, has active behavior as well as activated program. This behavior is defined by ECA rules and is capable of reaction to events. Active behaviors are functionality that is executed whenever certain requirements on environment are met. 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. As discussed in Sections 1, 2, in an ECA rule, Event is a primitive (basic) or composite event, Condition is a Boolean expression and Action is an action that should be executed. 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.

Therefore, activated system consists of two activated components: (1) The application program that to be monitored and (2) The control program (verifier) that monitors the application program. The application program is activated automatically by source-code level weaving using aspects. The control program is activated using a set of ECA rule. Therefore, the activated system consists of the event-aware functional units that emits events at certain points in their executions and consists of verifier functional ones that define how and under which conditions to monitor and verify behavior of the event-aware functional units at these points.

5. Implementation

An active system supports monitoring and storing events in an event history as {*event, type*} and {*time*} where the {*event, type*} represents any primitive event and the {*time*} is the time when the event occurred. In addition, an active system has clearly defined rule semantics such as event consumption policy (i.e. when events are discarded, when events are detected and signaled to the rule manager). Some possible event consumption policies are recent, chronicle, and cumulative. In the recent policy, the latest instance of a primitive event that is part of a complex event is consumed if the complex event occurs. In the chronicle policy, the events are consumed in time order. In the cumulative policy, all instances of a primitive event are consumed if the complex event occurs.

The expressiveness of the event part can be divided into comparing the types of events that the rules can reference and how the events can be modeled and combined into complex events. Different types of events include events such as sensor value changes, specified state changes in the applications, or time. Modeling events can include an event specification language that can combine events using logical composition, event ordering, sequential and temporal ordering, and event periodicity. The expressiveness of the condition part can be divided into whether the events can be referenced as changed data and whether old values can be referenced or not. The expressiveness of the action part can include rule activation/deactivation. Execution semantics of rules includes rule processing coupling modes. Cascading rule execution, i.e., (1) if one rule can trigger another and (2) if simultaneously triggered rules are subjected to some conflict resolution method are also part of the classification of rule semantics.

5.1. **Event Management.** Event management includes dispatching events received on an event bus to the rule processor. Event manager also supports storing events in event histories represented as time series that can be accessed through event functions. The event functions can be accessed by the rule processor. For example, the AMOS rule processor handles rule creation/deletion, activation/deactivation, monitoring, and execution. The processing of rules is divided into four phases: (1), Event Detection (2), Change monitoring (3), Conflict resolution and (4) Action execution. Event detection consists of detecting events that can affect any activated rules. Events are accumulated in event histories represented by event junctions. Change monitoring includes using the event data from the event functions to determine whether any condition of any activated rules have changed, i.e. have become true. During action execution, further events might be generated causing all the phases to be repeated until no more events are detected. The agenda is a time management module that can schedule activities to be per-formed at specific times.

The rule execution model in AMOS is based on the Event Condition Action (ECA) execution cycle. All events are sent on an event bus that queues the events until they are processed. The execution cycle is always initiated by non-rule-initiated events such as time events. All events are dispatched through table-driven execution. Events are accumulated chronologically in stored temporal event functions represented by time series.

6. Case Study

In this case study, we apply our approach to abstract data type, ADT, that have emerged as an effective mechanism for organizing large modern software systems. An ADT allows us build programs that use high-level abstractions. With ADT, we can separate the conceptual transformations that our programs perform on our data from any particular data structure representation and algorithm implementation. Therefore, an abstract data type is a data type (a set of values and a collection of operations on those values) that is accessed only through an interface. The ADT interface defines a contract between users and implementers that provides a precise means of communicating what each can expect of the other. This contract is specified by means of safety and liveness properties.

We refer to program that uses an ADT as a client, and a program that specifies the data type as an implementation. We consider a special case of ADT, so-called pushdown stack. A pushdown stack is an ADT that comprises operations of insert (push) a new item, delete (pop) the item that was most recently inserted, and visit (peek) the item. Insertions and deletions are made at one end called the top. The main request (i.e. concern) of client is access to stack through the interface; therefore, we consider two classes, one for stack object and other for the request object to get access to stack. The following shows request and stack classes:

```
class request {
state s;
bool change (state s);
void access (action act);
}

class stack {
state s;
bool create(stack st);
bool destroy(stack st);
bool change(state s);
void push(string s);
string pop(void);
string peek(void);
}
```

6.1. **Requirement Properties.** The concern for access to stack, contract between user and implementer, has two safety and two liveness properties. The safety properties (something bad never happens) are: (1) Stack overflow never happens; therefore, request must not able to push the item onto the full stack and (2) Stack underflow never happens; therefore, request must not able pop up from empty stack.

The liveness properties (something good will eventually happen) are: (1) The request to push the item onto not made full stack must be done and (2) The request to pop up the item from not made empty stack must be done.

6.2. **Aspects.** The concern for access has three aspects, (1) modifying the stack (modifying aspect crosscutted in push and pop units), (2) visiting the stack (visiting aspect in peek unit) and (3) existence the stack (existence aspect in create and destroy units). 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! state of request object is set to rejected
}
    call(public string stack.pop(void); // join point 2
    before() : UpdateStack( ) {
    if stack.state = = empty //underflow! state of request object is set to rejected
    request.change(reject);
}
}
```

6.3. **The Monitor.** First, we use UML diagrams for visualizing properties, and then generate ECA rules. Control program (verifier) for push unit of modify aspect formed by the generated ECA rules 1 and 2. When a request (an event) occurs that is related to modify aspect such as push event, (1) the safe guard ECA rule of verifier rejects the request if it causes overflow, (2) ECA rule 2 of verifier changes state of request and stack objects. Similarly, one visualizes safety property 2 of modify aspect for pop unit and generates the related ECA rule. In addition, we can visualize liveness properties.

```
ECA rule 1 for safety property 1 of modify aspect:
when push //safety rule 1
if (stack.full)
```

do rejected (request)
ECA rule 2 for safety property 1 of modify aspect:
when push(item) //safety rule 2
if stack.state is full and
request.state is initiated
do request.change(rejected)

7. Conclusions and Related Work

This paper presented a model to runtime verification of programs using rule based active system. The main feature of our approach is that we develop an active system by transformation passive environment as set of verifier and verified programs to active system. First, we make activate verified program by well making aware it of events. For this purpose, we automatically weave the monitor code into the target program by exploiting aspect-oriented method, which is one of the main contributions for automated support of runtime verification. Second by exploiting ECA rules, we show how one can develop an activated verifier program. This contributes to automatic runtime verification of verified program. Creation of monitoring rules was not automated. One can automate this by specifying the properties of design abstracts in a proposed Event-Condition-Action rule and bridge the gap between abstract specification of requirements and low-level implementation. Another advantage that be obtained 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. Some future work can apply the approach to distributed systems as well as real-time ones.

Meyer proposed Design by Contract approach for the object-oriented language Eiffel [8] 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 Eiffle, such as Ada and C++. While trace assertions in the Design by Contract are the counterpart of behavioral-oriented 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 [10]. There are automated trace analyzers, is connected to an event-oriented tracer. The traced program is executed alongside a trace analysis session in which the user enters high-level queries about the traced execution. The trace is then automatically processed according to the query.

In [11], Auguston and Trakhtenbrot automatically created monitoring Statecharts by the formulas that specify the system's behavioral properties in a proposed assertion language. Such monitors are then translated into code together with the system model, and executed concurrently with the system code. This approach leads to a more realistic analysis of reactive systems, as monitoring is supported in the system's actual operating environment. For models that include design level attributes (division into tasks, etc.), this is crucial for performance-related checks, and helps to overcome restrictions inherent in simulation and model checking.

The Monitoring, Checking and Steering (MaC) framework [12] 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 [13], rather than verifying properties directly, which can then be used to reason about programs or in error detection.

An alternative paradigm, declarative event-oriented programming [13], provides algebra of event combinators with a simple semantic model and embedded in a functional host language. Its ideas have been implemented in Fran ("Functional reactive animation"), a library for use with the functional programming language Haskell. It main idea is the explicit focus on declarative event-oriented programming. It attempts to convey this new paradigm for programming interaction applications, illustrate its use by means of a running example, and contrast it with the dominant but ill-structured approach, which is based on imperative callback procedures.

Using two special Interval Temporal Logic, FIL and GIL, [14] specified properties of a reactive system and created Harel's Statecharts from the specifications. These hierarchical and concurrent automata constitute runtime monitor and verifier for

the specified reactive system. Using UML State Machines, [15] stated specification of Object-Oriented programs and automatically produced ECA rules from the specification. The rules used to analyze the program runtime behavior.

References

[1] Luca Aceto, Anna Inglfsdttir, Kim Guldstrand Larsen, Jiri Srba, Reactive Systems Modelling, Specification and Verification, Cambridge University Press, 2007.

[2] Jennifer Widom, Active Database Systems, Triggers and Rules, Morgan Kaufmann, 1996.

[3] Norman W. Paton Paton and Oscar Daz, Active Database Systems, ACM Computing Surveys, Vol. 31, Number 1 (1999), 63-103.

[4] Paul Pedley, Intranets and Push Technology: Creating an Information-Sharing Environment, Aslib Publisher, 1999.

[5] Junman Sun, Huajing Fang, Ganyi Wang, Zhendong He, Information Push Technology and its Application in Network Control System, In Proc. of Int. Conference on Computer Science and Software Engineering (2008), 198-201.

[6] Alex King Yeung Cheung, Hans-Arno Jacobsen, Load Balancing Content-based Publish/Subscribe Systems, ACM Transactions on Computer Systems, Vol. 28, Numbaer 4 (2010).

[7] Hamidah Ibrahim, Event-Condition-Action (ECA) Rules for Maintaining the Integrity Constraints of Mobile Databases, IGI Publication, 2009.

[8] ISO/IEC 2543. Analysis, Design and Programming Language, Information technology-Eiffel. http://www.iso.org/iso/catalogue-detail.htm?csnumber=429242006.

[9] Heather J. Goldsby, Betty H. Cheng and Ji Zhang, AMOEBA-RT: Run-time Verification of Adaptive Software, Software Engineering Models in Software Engineering: In Proc. of Workshops and Symposia at MoDELS, Lecture Notes in Computer Science, Vol. 5002, 212-224, DOI: 10.1007/978-3-540-69073-3-23, 2008.

[10] Harry Foster, Applied Assertion-Based Verification: An Industry Perspective, Foundations and Trends in Electronic Design Automation: Vol. 3, Number 1 (2009), 1-95.

[11] Mikhail Auguston, Run-time Monitoring of Reactive System Models, The 2nd Int. Workshop on Dynamic Analysis, 2004.

[12] Usa Sammapun, Insup Lee and Oleg Sokolsky, RT-MaC: Runtime Monitoring and Checking of Quantitative and Probabilistic Properties. In Proc. of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (2005), 17-19.

[13] Daniel Zingaro. Invariants: A Generative Approach to Programming, College Publications, 2008.

[14] Seyed Morteza Babamir and Saeed Jalili, Runtime Monitoring and Verifying Reactive Systems Using Interval Temporal Logic, In Proc. of the 13th Iranian Conference on Electrical Engineering, University of Zanjan, 2005.

[15] Seyed Morteza Babamir and Saeed Jalili, Dynamic Analysis of Object-Oriented Programs Using State Machines and ECA Rules. In Proc. of the 14th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE-200S), Toronto, Canada (2005), 20-22.