Active Program Analysis Using Rule-Based Modification and Aspectation

Seyed Morteza Babamir
Department of Computer Engineering, University of Kashan,
Kashan, Iran,
babamir@kashanu.ac.ir

Abstract Active programs behave based on occurrence of events and therefore they facilitate capturing events and states. The active system is a rule-based system and we use event-condition-action rules to show active rules. Thus, defining active behavior is facilitated by event-condition-action rules. An active system, forming a runtime environment, sets a trap to catch runtime events and then check them by the rules. The rules appear in form of *event-condition-action*. Exploiting active programs as event based environments and using event-condition-action rules are main contributions of our approach. In this paper, we propose a new approach based on a bipartite framework exploiting capabilities of active systems. We apply our approach to a classical Abstract Data Type (ADT), stack, and express how one can use an active environment for observance tracking the stack.

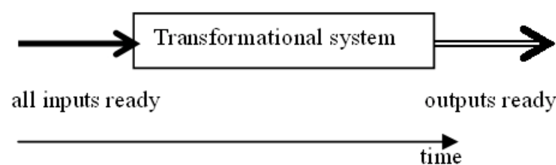
Keywords: Active system, event-condition-action rule, program modification, aspect-oriented

1. Introduction

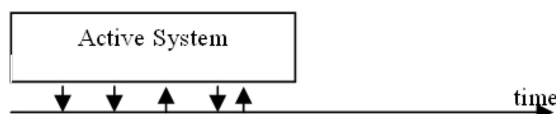
We could classify systems into two categories according to their behavior. First, *transformational systems* (e.g. compilers, data acquisition systems and etc.) that take input values, compute output values, and then are stopped. All inputs in transformational systems are ready when invoked and the outputs are produced after a certain computation period. These systems are generally *passive,* in the sense that user explicitly invokes them. Applications send requests for operations to be performed by the system and wait for the system to confirm and return any possible answers. Second, *active systems* whose action is based on occurrence of events, interacts with their environment continuously. Figures 1a and 1b show a transformational system and an active system.

Typical examples appear when the environment is a physical process (e.g., process control in industry, power plants; embedded systems in trains, aircrafts, traffic-light controller, etc). Almost most systems have some active components, because a system is not usually isolated from its environment. In addition, systems typically collaborate or interact with their environment. Therefore, a reactive behavior carries out such collaboration.

In contrast with transformational systems that are passive, active systems act based on occurrence of events. Accordingly, they are event driven and for verifying them one must monitor their reaction to the environment whenever an event occurs. The importance of active applications has increased in the recent years with the emergence of e-commerce applications such as stock market, business opportunities, and sale alerts, as well as system



a. A transformational system (all inputs ready)



b. An Active System continuously interacts with its environment

Figure 1. Transformational and active systems

management applications, such as command and control applications. There are two types of active system, interactive and reactive. *Interactive systems* are those that permanently communicate with their environment, but at their own speed (e.g. operating systems, web server).

Reactive systems are those which react to their environment, but at speed of environment. The term reactive system was introduced by David Harel and Amir Pnueli [1] and is now commonly accepted to designate systems that permanently interact with their environment, and to distinguish them from *transformational systems*. In contrast with most interactive systems, reactive systems are generally *deterministic:* from an abstract point of view, the execution of a reactive system can be viewed as an infinite sequence of input/output vectors, where, at each step, the output values are completely determined by the past and present inputs.

The idea of *active system* is close to the active databases that were coined in [2] as meaning "a paradigm that combines aspects of both database and artificial intelligence technologies". In [2] a mechanism for constraint maintenance was presented as a declarative representation for a set of related production rules in form of Condition-Action. In HiPAC [3, 4] a thorough specification was made of what different mechanisms are desirable in an *Active Database Management System* (ADBMS). Active rules are defined as *Event-Condition-Action* rules, where the Event specifies when a rule should be triggered, the Condition is a query that is evaluated when the Event occurs, and the Action is executed when the Event occurs and the Condition is satisfied. Events can be seen as signals that inform that a change to data in the database has occurred, e.g. an update of a table. There are three rules: (1) *immediate* meaning the rule conditions are evaluated and the actions are executed immediately when the event occurred. A distinction was also made between whether the rule process takes place before or after some change in the database. (2) *Deferred* rule meaning the rule process is delayed until committing the transaction. (3) *Casually Dependent Decoupled* rule meaning any triggered action is executed in a separate sub-transaction that waits until the main transaction is committed. *Decoupled* rule process means that the sub-transaction is completely decoupled from the main transaction and commits regardless of the outcome of the main transaction.

Active systems are event driven ones where operations such as changes to data generate events that can be analyzed by *active rules.* An active system can be invoked, not only by synchronous events that have been generated by users or application programs, but also by external asynchronous events such as changes of sensor values or time. For example, *push technology*, i.e., the ability of sending relevant information to clients in reaction to new events, is a fundamental aspect of modern information systems [5, 6]. One important aspect of Internet-based information systems is the ability of pushing information to clients, by matching new event occurrences with predefined user's interests. Such ability is embedded within many WEB development products and applications, which support one-to-one information delivery in response to users' current and past interactions. Active rules are an important ingredient for supporting this reactive technology.

The remainder of the paper is organized as follows. Section 2 explains active rule and their roles in an active system. Section 3 explains necessity of runtime analysis. Because a controlled program should be well aware of events, Section 4 deals with program awareness by program modification and aspectation facility. Section 5 explains how the control and controlled programs interrelate. Section 6 states some known analysis approaches. Section 7 proposes: (1) a bipartite model for analysis of environment, i.e. the program that should be analyzed and (2) transformation of passive environment to active one. Section 8 proposes case study "*stack*" as an *abstract data type* and verify its required properties and applies the steps of the proposed approach. Finally, Section 9 concludes major advantages of our approach.

2. Active Rules

Historically, production rules were the first mechanism used to provide automatic reaction functionality. Production rules are Condition-Action rules that do not break out the triggering event explicitly. Instead, they implement a polling style evaluation of all rule conditions. When monitoring events in a passive system, a *polling technique* or *operation filtering* 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 analysis 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 that we be able to specify the conditions to be analyzed. 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 [7, 3], workflow management, network management, personalization and publish/subscribe technology [8] and specifying and implementing business processes [9]. Triggering and activation relations between rules have been used to determine whether a set of event-condition-action rules is terminating. A rule *r*i *may trigger* a rule *r*j if the action of *r*i may generate an event which triggers *r*j. The *triggering graph* [10] 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. A cyclicity of the triggering graph implies definite termination of rule execution. An *activation graph* [11] also represents rules as vertices. In this case an arc between two distinct vertices *r*i and *r*j indicates that *r*j*'s* condition may be changed from False to True after the execution 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. Acyclicity of this graph also implies definite termination of rule execution. In [12], trigger and activation graphs were combined by *rule reduction* method giving more precise results than either of the previous methods. By this method, any vertex not consisting of both an incoming triggering and activation arc can be removed from the graph, along with its outgoing arcs. This removal of vertices is repeated until there are no such vertices. If the procedure results in all the vertices should be removed, then the rule set is definitely terminating.

In rule-based systems, the active rules can be used for purposes of *analysis, 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 [13] 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*.

An active system supports event analysis and storing events in an *event history* as *«event type>, <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.

2.1 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 rule processing 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 performed at specific times.

The rule execution model in AMOS is based on the *Event Condition Action* 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.

3. Active Program Analysis

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. Hamani et al for example, used a formal specification and verification method to production systems [14]. 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 analysis, where a property may be assured with mathematical rigor, 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 analysis.

Runtime analyses have been proposed as lightweight formal analysis 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 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. 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.

3.1 Active rules and program analysis

Functions that can use data monitoring includes, for example, verifying constraint and controlling authorization. In verification of constraint, rules can analyze and detect inconsistencies and abort any queries that violate the constraints. In controlling authorization, rules can be used to check that the user or application has permission to do specific actions in the system. Applications that depend on data analysis activities such as, Telecommunications Network Management and Financial Decision Support Systems can greatly benefit from active system concept.

4. Modification

For analysis of a program, the program should be equipped with some analyzer code to capture events that the application program emits. In fact, it is used to obtain information about the program behavior. The equipment process is carried out by modifying the program *invasively* or *non-invasively* [15]. In an invasive approach, logic of emitting events and reaction to the emitted events are embedded in application program. This logic can be embedded in source, byte, or executive code of application. Figure 2a shows pure invasive approach for modifying an application program. Invasive approach can be impure, too. In this approach shown in Figure 2b, reaction code is separated from the application program and managed by a separate entity. In non-invasive, not added any code to application program and both emitting and reaction code is separated from it. Figure 2c shows non-invasive approach.

Modification mechanism for invasive approach can be done in several ways. The most common ones, despite its limitations, is manually insert modification code into the program to be analyzed.

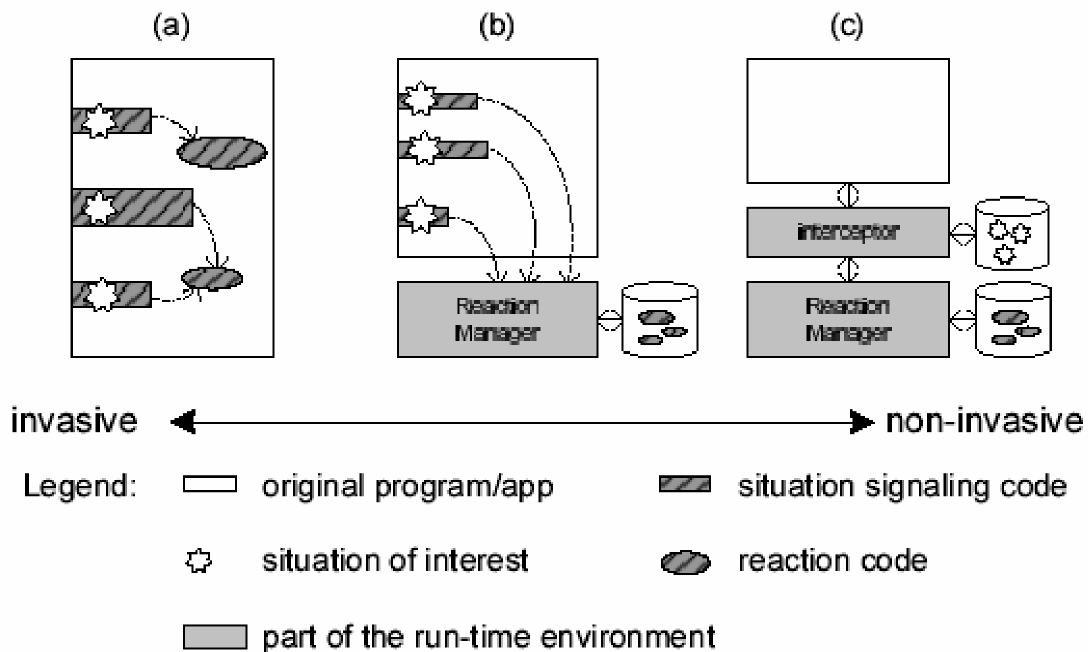


Figure 2. Application program modification as invasive or non-invasive [15]

A user reads a source code of the program, and then inserts probe codes into program. This is time consuming and may be incomplete. For assurance, modification should be complete in the sense that it should capture all interesting information. Missing information could lead to false or missed detection of faults. Another mechanism is Runtime modification that refers to the modification of the analyzed 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 modification, the executable-code level modification is complicated since the source-level modification useful for understanding the program. In addition, modifying a system at executable-code level requires modification not directly related to analysis, but necessary to keep the format of executable code consistent.

The modification mechanism for non-invasive approach is interpreter modification that inserts analysis code in the language interpreter itself. Such modification can provide information about the behavior of any program executed by the interpreter; compiler modification also includes preprocessors and code generators that add modification to the code they generate. Such code usually is much larger than the non-modified code. To modify an interpreter or a compiler we should have their source code or it should have already been modified. Interpreter or compiler modification is a difficult problem because modifier needs to be involved in understanding source code of the interpreter or compiler. On the other hand, the modified interpreter or compiler may be not to match our probe needs adequately.

5. Analysis

There are three models to connect the analyzer and the target program: (1) the *one-process model* (for pure invasive approach) and (2) the *two-process model*, and the *thread model* (for impure and non- invasive approaches). In the one-process model, an execution analyzer is a library of procedures linked to the target program or it is integrated into the run-time system. The one-process model has good performance and access characteristics, but it does not prevent the target program and analysis code from affecting each other in critical ways. In addition, the control flow logic within the analyzer is somewhat inverted, since the analyzer is activated through callbacks. In the two-process model, the analyzer is a separate process from the program being analyzed, reducing the problem of intrusion at the expense of complicating access and reducing performance (impure invasive or non-invasive approach). In the thread model, the analyzer is a separate thread in a shared address space occupied by the program and possibly other analyzers, providing a reasonable compromise between the characteristics of the one-process and two process models for many analysis applications. Interaction facilities vary both in terms of the kind of execution controls provided to the user, and the techniques used in presenting the user with execution information. Execution controls range 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. Related Work

Meyer proposed *Design by Contract* approach for the object-oriented language Eiffel [13] 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 analysis of the dynamic behavior of an object, the ordering of method invocation and calls in time [16]. 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.

Auguston and Trakhtenbrot automatically created monitoring UML Statecharts by the formulas that specify the system's behavioral properties in a proposed assertion language [17]. 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 [18] 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 [19], 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 [20] 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.

Considering time constraints and with assembling tasks, M'halla et al exploited combination of chronicle and fault tree approaches to analyze a milk manufacturing workshop [21]. By proposing their method, they aim to help workshop operators for identifying failures in order to stay away from some damage of an accident with humans.

7. Program Analysis Using Active Systems

An active system transforms a passive process into an active environment by using alerts and triggers. In this section, we use a bipartite model, inspired by [18], for runtime verification of environment, i.e. the program that must be analyzed. The model has two parts, mechanisms, and verifier. Figure 3 shows the bipartition, which makes data handling on active system easier. The mechanisms part has two tasks, profiling those events that occur in environment and doing those decisions made by the verifier part. The verifier part is responsible for making policies and valid decisions. In the runtime, needed action is made after data profiling and monitoring. Taking action by actuators, state of environment can be changed that may drive another event.

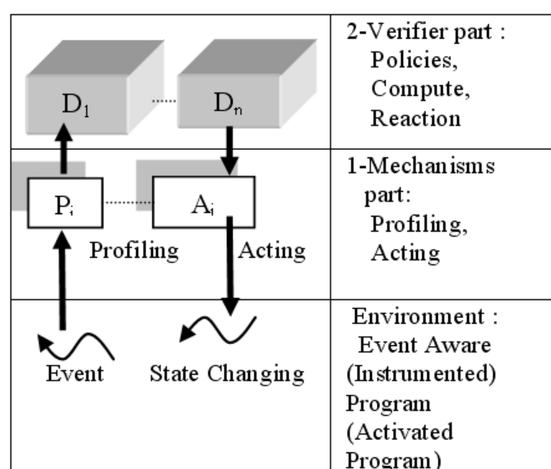


Figure 3. Bipartite model for data handling on active system

A course of events, environments conditions, and actions, embedding in event-condition-action 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 ri to a vertex rj if ri may trigger rj. An action of rule ri may generate an event which triggers rule rj. Activation of a rule may change an environment condition. An activation graph representing rules as vertices, has an arc between vertices ri and rj indicates that rj's condition may be changed from False to True after the execution of ri's action, while an arc from a vertex ri to itself indicates that ri's condition may be True after the execution of ri's action.

7.1 Program activation

Our approach is different from past-related work. Our main idea is to provide an active environment to support runtime analysis and verification. To this end, first we equip the program with an analyzer code (see Section 4).

We call the equipment process program activation because the program is made well aware of occurrence of events. Awareness can be done by invasive or non-invasive approach, discussed in Section 4. Our approach use invasive modification (Figure 2b). Therefore, we automatically modify source-code program by aspects. This type of program modification has no disadvantage of mentioned manually source-code level modification and has no difficulties of executable-code level. Modifying source-code can be achieved automatically by using aspect-oriented approach. By aspect, we define each property that is to be verified. Therefore, by aspect definition, related probe codes (i.e., crosscutting points of property) are inserted into units of the program.

Aspectation holds promise for the creation of aspects, which are modules that centralize distributed functionality. Figure 4 shows that how one uses aspects for automatic modification of source-code. The modified program emits a signal when a related event occurs. Afterwards, profiler in the mechanisms part, profile the event and sends it to decision support part.

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 rules. The verifier, a set of the rules, has active behavior as well as an activated program. This behavior is defined by the rules and is capable of reaction to events. Therefore, we call such behavior active behavior and in fact it is a functionality executed whenever certain environment requirements are met. These rules specify triggering event and guarding condition for each action (process). The action is executed on triggering event, if and only if the guard condition is fulfilled at that time. As discussed in Sections 1, 2, in an event-condition-action 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.

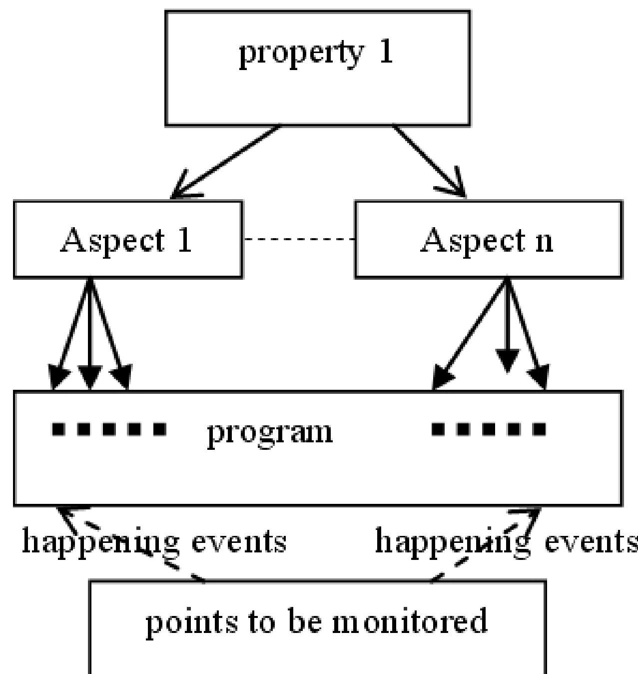


Figure 4. Automatic instrumentation of source code by aspects

Second, we build verifier as a rule based system. The verifier gets events that are emitted by modified program and analyzes them. Events are objects, which represent execution points of program, such as method call and

Therefore, activated system consists of two activated components: (1) The target program to be analyzed and (2) The control program (verifier) that analyzes the target program.

The target program is activated automatically by modifying source-code level using aspects. The control program is activated using a set of event-condition-action rules. 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 analyze and verify behavior of the event-aware functional units at these points.

8. 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. By 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. Figure 5 shows request and stack classes.

8.1 Requirement properties

The stack access concern, 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 be able to push the item onto the full stack and (2) Stack underflow never happens; therefore, request should not be able to pop up from empty stack.

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

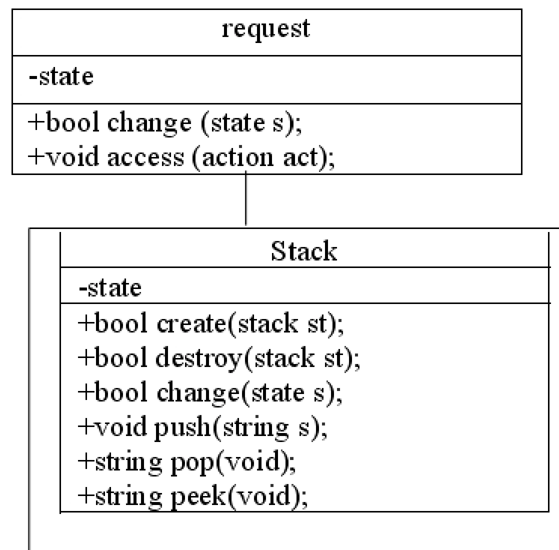


Figure 5. Request and stack classes

8.2 Aspectation

The access concern has three aspects: (1) Modifying the stack (the aspect is crosscut in push and pop units), (2) Visiting the stack (the aspect is crosscut in peek unit) and (3) Creating and removing the stack (the existence aspect in the “create” and “destroy” units).

As stated above “access to stack” is the concerned requirement and absence of stack overflow and underflow are safety properties. For the concerned requirement and the safety properties, we consider *ModifyAspect* in which points of interest to be analyzed are push and pop units.

The following code shows the aspect in the AspectJ language.

```
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 state of
request
            //  object is set to rejected
            request.change(reject);
      }
}
```

8.3. Tracker

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

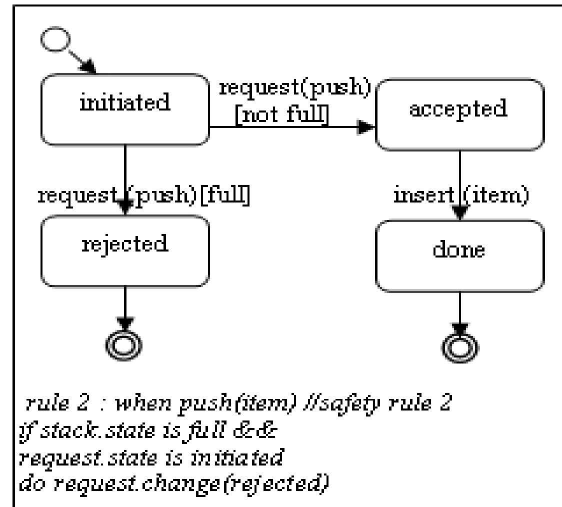


Figure 7. UML Statechart diagram for push join point of modify aspect

Control program (verifier) for push unit of modify aspect formed by the generated even-condition-action rules in Figures 6 and 7. When a request (an event) is occurred that is related to modify aspect such as push event, (1) the safe guard rule (Figure 6) of verifier rejects the request if it causes overflow, (2) the rule 2 of verifier (Figure 7) changes state of request and stack objects. Similarly, one visualize safety property 2 of modify aspect for pop unit and generate the related rule. In addition, we can visualize liveness properties.

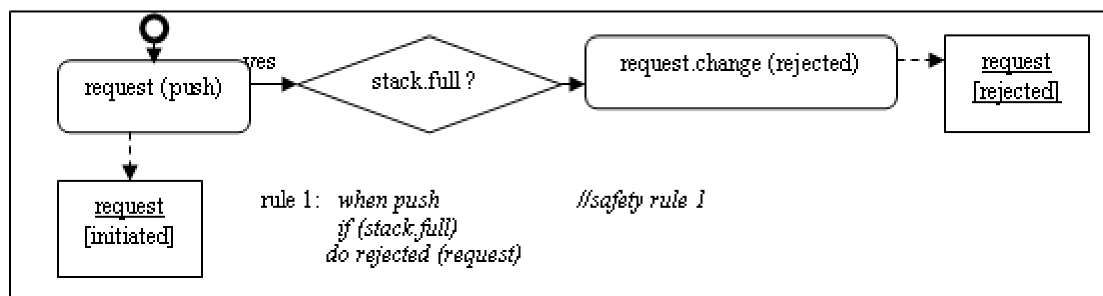


Figure 6. Activity diagram for the overflow prevention safety property of modify aspect and its rule

9. Conclusion

This paper presented a rule-based bipartite model to runtime verification of programs. The main feature of the presented approach is equipping a software system (including verifier and verified programs) with an analyzer. The equipment process called activation exploited Aspect-oriented method.

First, we equipped the verified program with an analyzer making the program well aware of events. To this end, we automatically modified the program by exploiting Aspect-oriented method that is one of the main contributions to automated support of runtime analysis. Second, by exploiting event-condition-action rules, we showed how one can develop an activated verifier program. This contributes to automatic runtime analysis of a verified program. Creation of analysis 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 is analyzing diverse behavior of a program in terms of satisfaction of different parts of the rule(s). The analysis of some typical behaviors was shown via one case study. Some future work can apply the approach to distributed systems as well as real-time ones.

REFERENCES

1. HAREL, D., M. POLITI, **Modeling Reactive Systems with Statecharts**, McGraw-Hill, 1998.
2. MORGENSTERN, M., **Active Databases as a Paradigm Enhanced Computing Environments**, Proceedings of the 9th VLDB Conference, Florence, Nov. 1983.
3. PATON, N. W., O. DÍAZ, **Active Database Systems,** ACM Computing Surveys, vol. 31(1), 1999, pp. 63-103.
4. WIDOM, J., S. CERI, **Active Database Systems-triggers and Rules for Advanced Database Processing**, Morgan Kaufmann Publishers, Inc. , 1996, ISBN-1-55860-304-2.
5. DENG, X., **Application of Information Push Technology in Residential Building Performance Assessment System**, in Proceedings of 8th World Congress on Intelligent Control and Automation (WCICA), 2010, pp. 3965-3968.
6. SUN, J., H. FANG, G. WANG, Z. HE, **Information Push Technology and Its Application in Network Control System**, in Proceedings of the International Conference of Computer Science and Software Engineering, 2008, pp. 198-201.
7. WIDOM J., S. CERI, **Active Database Systems**, Morgan-Kaufmann, San Mateo, California, 1995.
8. CHEUNG, A. K. Y., H. A. JACOBSEN, **Load Balancing Content-Based Publish / Subscribe Systems**, ACM Transactions on Computer Systems (TOCS), vol. 28(4), December 2010.
9. BRY, F., M. ECKERT, P. PATRANJAN, I. ROMANENKO, **Realizing Business Processes with ECA Rules: Benefits, Challenges, Limits,** in Proceedings of International Workshop on Principles and Practice of Semantic Web Reasoning, Lecture Notes in Computer Science, vol. 4187, 2006, pp. 48-62.
10. TERADA, T., M. TSUKAMOTO, S. NISHIO, **Dynamic Construction Mechanism of a Trigger Graph on Active Databases in Mobile Computing Environments**, in Proceedings of the 14th International Workshop on Database and Expert Systems Applications, IEEE Computer Society Washington, DC, USA, 2003, pp. 936-941.
11. SHANKAR, C., R. CAMPBELL, **A Policy-based Management Framework for Pervasive Systems using Axiomatized Rule-actions**, in Proceedings of the 4th of IEEE International Symposium on Network Computing and Applications, IEEE Computer Society Washington, DC, USA, 2005.
12. BARALIS, E., S. CERI, S. PARABOSCHI, **Improved Rule Analysis by Means of Triggering and Activation Graphs**, Second International Workshop on Rules in Database Systems (RIDS-95), Athens, LNCS 985, Springer, 1995, pp. 165-181.
13. **Analysis, Design and Programming Language**, Information technology-Eiffel: ISO/IEC 2543, 2006.

14. HAMANI, N., N. DANGOUMAU, E. CRAYE, **Specification and Verification of the Model of Component and the Model of Function**, Studies in Informatics and Control, vol. 17(1), 2008.

15. GOLDSBY, H. J., B. H. C. CHENG, J. ZHANG, **AMOEBA-RT: Run-time Verification of Adaptive Software, Software Engineering Models in Software Engineering**, in Proceedings of Workshops and Symposia at MoDELS, Lecture Notes in Computer Science, vol. 5002, 2008, pp. 212-224.

16. CILIA, M., M. HAUPT, M. MEZINI, A. BUCHMANN, **The Convergence of AOP and Active Databases: Towards Reactive Middleware**, in Proceedings of the International Conference on Generative Programming and Component Engineering (GPEC'03), Lecture Notes In Computer Science, Springer, 2830, 2003, pp. 169-188.

17. TONG, J. G., M. BOULE, Z. ZILIC, **Defining and Providing Coverage for Assertion-based Dynamic Verification, Journal of Electronic Testing: Theory and Applications**, Kluwer Academic Publishers, vol. 26(2), 2010, pp. 211-225.

18. AUGUSTON, M., M. TRAKHTENBROT, **Run-time Monitoring of Reactive System Models**, The 2nd International Workshop on Dynamic Analysis (WODA), Edinburgh, 25 May 2004.

19. KIM, M., I. LEE, U. SAMMAPUN, J. SHIN, O. SOKOLSKY, **Monitoring, Checking and Steering Real-time Systems**, in Proceedings of the 2nd International Workshop on Run-time Verification, July 2002.

20. SAHOO, S. K., L. MAN-LAP, P. RAMACHANDRAN, S. V. ADVE, Z. YUANYUAN, **Using Likely Program Invariants to Detect Hardware Errors**, in Proceedings of IEEE International Conference on Dependable Systems and Networks with FTCS and DCC, 2008, pp. 70-79.

21. M'HALLA, A., S. C. DUTILLEUL, E. CRAYE, M. BENREJEB, **Monitoring of a Milk Manufacturing Workshop Using Chronicle and Fault Tree Approaches**, Studies in Informatics and Control, vol. 19(4), 2010.