International Journal of the Physical Sciences Vol. 6(34), pp. 7771 - 7779, 16 December, 2011
Available online at http://www.academicjournals.org/IJPS
DOI: 10.5897/IJPS11.787
ISSN 1992 - 1950 ©2011 Academic Journals

Full Length Research Paper

Dynamic checking reactive applications: An event driven framework

Seyed Morteza Babamir

Department of Computer Engineering, University of Kashan, Kashan, Iran. E-mail: babamir@kashanu.ac.ir.

Accepted 14 November, 2011

Reactive applications are embedded software of reactive systems which continuously interact with their environments. In this paper, we aim to propose a method to dynamically checking reactive applications using event based rules. The rules formed in event-condition-action constitute the checker as an active program. In order to enjoy activeness, the checker actively reacts to environment events when they occur. The checker as an active system, in fact, catches runtime events and reacts to them. The effectiveness of the proposed method is shown by checking some properties in case study.

Key words: Dynamic checking, event based rule, active database, aspect-oriented.

INTRODUCTION

Reactive applications performing according to occurrence of events have continuous interaction with their environment. The term reactive system introduced by David Harel and Amir Pnueli (Harel and Politi, 1998) denotes systems that continuously interact with their environment. They react to their environment at the speed of the environment. E-commerce applications such as stock market and sale alerts, system management applications, such as command and control applications, traffic-light controller and process control in industry are the quintessential ones. The significance of such applications is rising because the majority of systems somehow have interaction with their environment. In such systems, the system reactions to events should be verified.

Reactive systems have commonly *deterministic* behavior. Although the execution of a reactive system can be an infinite series of input/output sequences, the output values are completely determined by the past and present inputs at each step.

In this paper, we use event based rules to demonstrate behavior of reactive system. Event based rule was first used in active databases (Morgenstern, 1983) in form of Event-Condition-Action. In Paton and Diaz (1999) and Widom and Ceri (1996), a couple of mechanisms have been applied to event based rules. In Event-Condition-Action rule base rule, the event triggers the rule and the condition in an expression should be held for firing the rule. Upon triggering the rule, the action is executed. In other words, upon occurrence of the event and holding the condition(s), the rule fires and the action is taken. The action may be taken instantly or with a delay. Also, rules may be cascade meaning that firing a rule is casually dependent on firing some other rule/rules.

In this paper, we aim to propose an active environment to support checking reactive software. To this end, we equip source code of the target software with observer code and then, we construct a checker in form of an active program using event based rules. Events represent execution points of the target software, such as method calls and returns. The activeness of the checker program enables it to react to events when they happen.

We continue this paper as follows: The related work is discussed in related work. Dynamic checking and event-based rule are discussed in dynamic checking and event based rules, respectively. The proposed method is considered and applied to a case study in the proposed model and case study. Finally in conclusion, we deal with conclusions and major advantages of the proposed method.

RELATED WORK

Barringer et al. (2004) presented a rule-based framework for monitoring specifications stated in temporal logics.

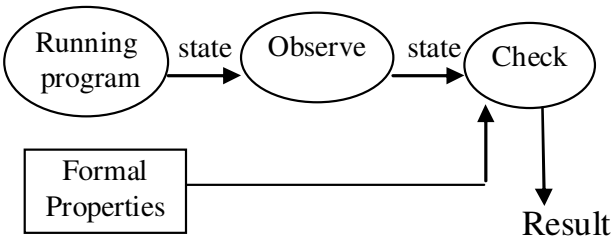

Figure 1. Dynamic checking (Delgado et al., 2004).

They implemented their method as a Java library called EAGLE. EAGLE was introduced as a general purpose rule-based temporal logic for specifying run-time monitors. Associating actions with the stated formulas and incorporating monitoring code into the target programs automatically is their future ideas.

Chavarría-Báez and Li (2006) proposed an active rule based verification to verify knowledge base. Their method is based on conditional colored petri nets. They stated that they consider incorporating the analysis of ECA rules with temporal composite events.

Song et al. (2011) proposed a method to observe potential violations of *rules* derived from protocol specification standards such as those specified by Internet Engineering Task Force (IETF) and Request For Comments (RFC) (Carpenter, 2011).

They exploited words such as “MUST” and “SHOULD”, used to express requirements in standard documents to extract the rules.

Reger (2010) considered the advantage and way of rule-based runtime verification in a multicore system. The method is evaluated using a number of micro benchmarks from the DaCapo benchmark suite. Reger used RuleR, which is a rule-based runtime verification tool and consists of a specification language. Properties are defined in terms of parameterized conditional rules in form of *rulename: antecedent→ consequent*.

A rule indicates that the consequent should be held on the next step if the antecedent is true for the current step. The RuleR algorithm takes a RuleR specification and an observation trace to decide.

Barringer et al. (2010) and Barringer et al. (2009) stated that their previous work called EAGLE is complex and difficult to efficient implementation. Accordingly, they introduce RULER, a primitive conditional rule-based system, for effective run-time checking. Then, they introduced the parameterized RULER where rule names may have rule expression or data parameters. They proposed a trace-checking algorithm to check a finite trace of ground observations for conformance against the rules of the RULER specifications.

Pankowski (1995) contended with monitoring temporal behavior, integrity constraints and controlling activities in database systems using ECA rules. The rule conditions are stated in temporal expressions and may be expressed as a query in an algebra or calculus formula. Events may have attributes holding data about the current state of database or about the activity generates the event.

Koschel and Astrova (2008) proposed a method to monitor events in Web services using distributed ECA rules. They stated that their method addresses: (1) Description and detection of arbitrary event types from heterogeneous distributed sources, (2) support of parameters in ECA rules such as event occurrence notification time such as after, before, instead and event granularity such as an event instance or a set of events.

d’Amonrim and Havelund (2005) introduced a temporal logic called HAWK and its supporting tool to monitor Java programs at runtime. HAWK is an extension of the rule-based EAGLE logic (Barringer et al., 2004) with constructs for capturing parameterized program events such as method calls and returns.

In Sahoo et al. (2008) and Elliott (2000), event paradigm was extended using algebra of events to construct new type of events. Each new event may be used to construct other events; accordingly an arbitrarily sophisticated of events may be created. The authors stated that the event oriented programming can aid in dividing programs into understandable and reusable pieces. Declarative event-oriented programming proposed by Sahoo et al. (2008) and Elliott (2000) is algebra based method of event combinators embedded in a functional host language.

In Babamir and Jalili (2005), we used UML State Machines (that is, Activity and Statechart Diagrams) for specification of object-oriented programs. Then, we automatically produced ECA rules from State Machines and dynamically analyze the runtime behaviors of programs.

DYNAMIC CHECKING

Dynamic checking has already been proposed as a method with the goal of checking system runtime behavior against formal requirements (Figure 1) (Delgado et al., 2004).

This technique: (1) Bridges the gap between formal verification of software specification and testing of the software implementation. This leads to validation of requirements (properties) and steering of the software program at runtime; (2) Decides about current execution of program not about all the executions. Dynamic checking, in fact, considers properties that (1) were left undecided in verification of software specification, (2) were not discovered by testing of the software implementation and (3) are closely related to physical environment where the software executes.

As Figure 1 shows, states of running program should be

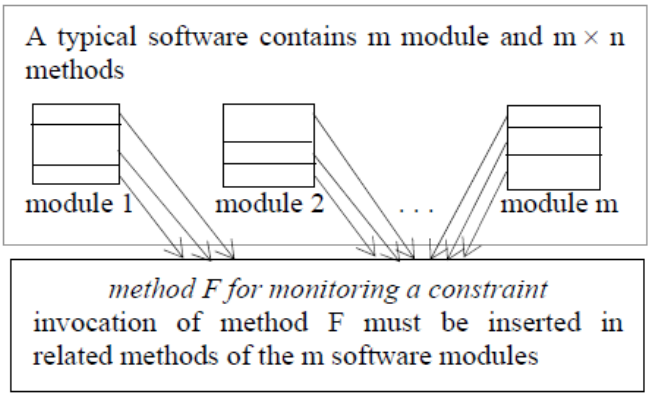

Figure 2. Weaving into modules of software (Babamir and Jalili, 2010).

observed to analysis. Observing may be carried out in one-step, two-step or multithreaded method. The one-step method, the observer code is designed as a library of procedures linked to the target software or integrated into the run-time system. This method enjoys good performance but it is an intrusive method because the target software and observer are able to affect each other.

The two-step model, the observation process is carried out separately. Accordingly, the target software and the observer cannot affect behavior of each other. However to use this method, we contend with complexity of construction of the observer. In addition, it reduces performance. The multithreaded method, the observer is constructed as a separate thread and executed in parallel with the target software.

Weaving

As stated in dynamic checking, to apply the one step method, one should *weave* the observer code into the target software. Figure 2 (Babamir and Jalili, 2010) shows weaving an invocation method into software modules where the method undertakes observing the software behavior against some constraint (property).

The weaved code will send software states as events to the analyzer when it executes (Figure 1). The observer code used to acquire program behavior may be embedded or disjointed (Goldsby et al., 2008). The former indicates that observer code is embedded in the target software while the latter indicates that the observer code is executed in parallel with the target software.

EVENT BASED RULE

Event based rule was first used in active database systems (Morgenstern, 1983; Paton and Diaz, 1999; Widom and Ceri, 1996). An active database management system (Figure 3) is a system reacting to the events actively without intervention of user.

The activeness is stated by event based rules in form of Event-Condition-Action where: (1) Event stands for a concerned change in the system environment; (2) Condition is a predicate allowing/disallowing firing the rule and (3) Action indicating the system reaction when the event occur and the predicate holds. An event based rule is shows as Relation (1).

On *event* When *condition* Then *action* (1)

By exploiting event based rules we can enjoy advantages of rapid detection of abnormal events, notification of users and the centralized control of services should be served by the system.

Active rule is a dominant method used in reactive behavior. In Chavarría-Báez and Xiaoou (2010), a software tool called ECAPNVer was specified for verifying an active rule base. According to Chavarría-Báez and Xiaoou (2010), the tool can detect and correct structural errors as well as potential errors such as redundancy and partial redundancy, inconsistency and partial inconsistency, incompleteness and circularity. The active rules may be defined statically or dynamically. In the static method, the rules are defined based on properties have been specified in advance while in the dynamic one, the rules are created at run time (Chakravarthy and Varkala, 2006).

Active rules have been used to monitoring (such as active data bases), control (such as reactive systems) and reasoning (such as knowledge based systems) using stored facts and deducing new facts. In addition, in the most programming languages, *exception handlers* can be defined for catching program exceptions (events). In these handlers, active rules are used to catch *events*.

Active rules may be used to data observing in order to verify constraints and control authorization. In verifying constraints, rules observe and detect inconsistencies and abort queries that violate the constraints. In controlling authorization, rules check user/ application permission to perform actions. Management of telecommunications network and decision support systems are applications that depend on data observing activities.

We use event rules because reactive systems are event driven. Event based method helps us in representing a reactive system so that it may be invoked, not only by synchronous/ asynchronous events generated by users, application programs or changes of sensor values or time. *Publish/subscribe*, for instance, is an event based reactive system where providers publish notifications and consumers subscribe to notifications by issuing subscriptions, which are stateless event filters (Cheung and Jacobsen, 2010; Parzyjegla et al., 2010). In

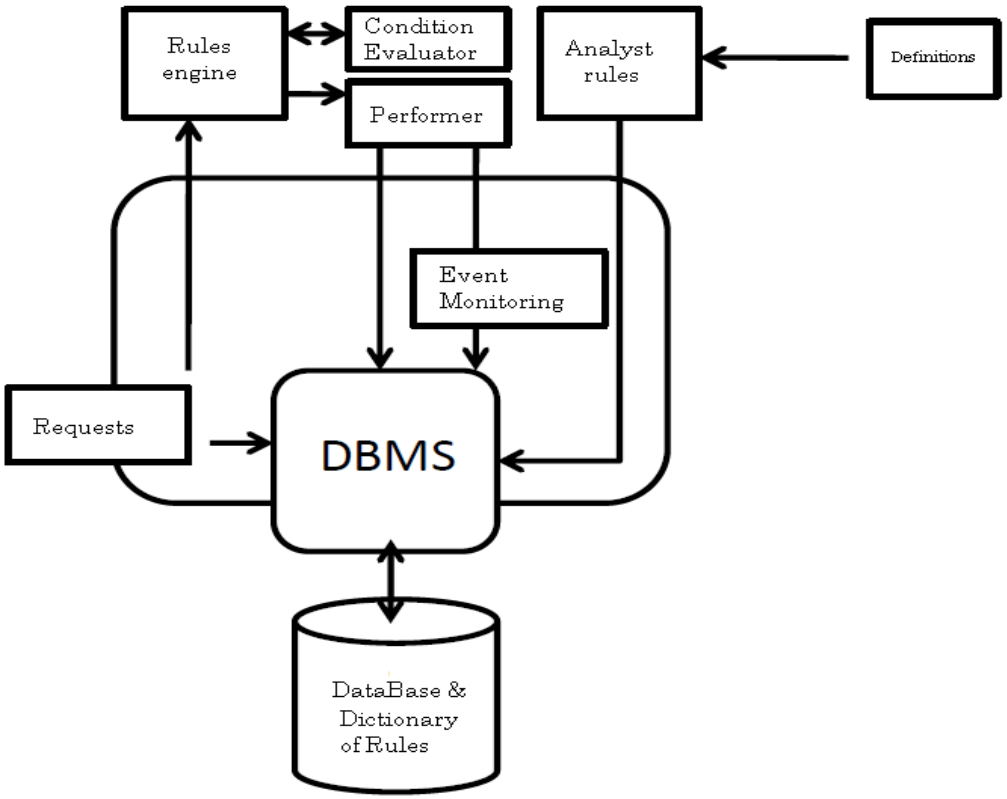

Figure 3. Event rules in active database.

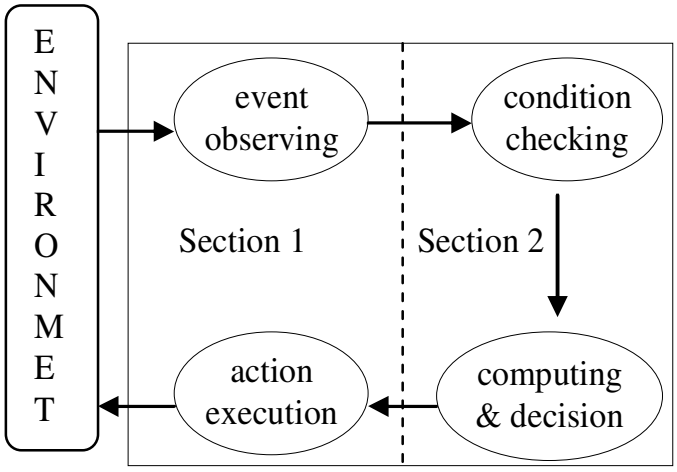

Figure 4. The proposed framework.

this model, routing is decided using distribution of event notifications in the network. Events may be different; typical examples are change of a sensor value, change of an application state or change of time. In addition, events may be combined into complex events. The combination may be carried out in form of logical composition, event ordering, sequential and temporal ordering and event periodicity. Rules in combined and complex rules fire according to *consumption* policies. The policies are new, historical, and increasing. The first policy denotes consumption of the most recent primitive event of a complex event if the complex occurs. In the second policy, events are consumed in time order. In the third policy, all primitive events of a complex event are consumed if the complex event occurs.

Event handling

Dispatching events to the rule processor, storing events in a history are tasks of event handler. The rules are processed in four steps: (1) Detection (2), Condition observing (3), Conflict resolution and (4) Action execution. In the first step, events that may influence any activated rules are detected. Then the events are stored in a history. Condition observer is responsible for monitoring condition of any activated rule when it becomes true. Execution of an action may cause to fire further events. This leads to recurrence of all the steps to be repeated until no more events are detected.

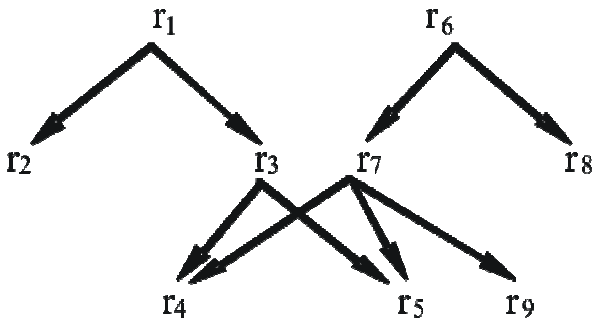

Figure 5. Trigger graph of a composite rule.

```
Aspect Aspect1 {

pointcut P1() :

call (void *.credit (float));
before: change() {
write("about to perform method" );
  }
}
```

Figure 6. A typical aspect with pointcut P_1.

THE PROPOSED MODEL

We use triggers to activate processes in order to check software at run time. To this end, we represent a framework consisting of two sections: (1) Event observation and action execution and (2) Checking and decision (Figure 4). Having observed a concerned event and environment conditions, the first section sends them to the second section. The second one checks conditions and computes some reaction and takes some action if the condition(s) are satisfied. The decision is taken according to policies ascertained in advance. After taking the action, the environment state will change to new state that it may drive another event.

A *trigger graph* is constituted when we have a sequence of events, conditions and actions. In the graph, a vertex denotes a rules and an arc from a vertex to other vertex(es) denotes triggering target nodes by the source one. Figure 5 shows a trigger graph where rules r_1 and r_6 trigger rules r_2 and r_3 and rules r_7 and r_8 respectively. Rules r_4 and r_5 are triggered by both rules r_3 and r_7.

Activation of a rule may change status of the system environment. In an *activation graph,* a rule is thought as a vertex and an arc denotes realization of rule condition of the source vertex after the execution of the rule action. In fact, in the trigger graph we have a trigger when a rule event occurs, while in the activation graph we have activation when a rule condition becomes true. An activation graph is complementary to a trigger graph.

Weaving method

Here, we aim to address our weaving method (Weaving) by which the observer code is weaved into the target software. The observer code undertakes the task of sending software states to the checker. In fact, the software will become activate when it is equipped with the observer code. This means that the embedded observer code will notify the checker when some event captured by the target software. As Figure 2 shows, the observer code should be weaved in proper places of the target software. To weave the observer code automatically into source code of the target software, we use *aspects* (Katz and Mezini, 2011). The automatic weaving avoids the weakness of manual one.

In a manual weaving, source code of the target software is read by user and the observer code is inserted in. Due to manual method is time consuming and suffers from probable incompleteness, the automatic method is preferable. If inserting the observer code in some places of the target software is missed, it leads to the missing information that the checker needs to check behavior of the target software. Accordingly, false or missed detection of faults may occur.

If we consider that the target software would have a function to process each distinct event, the observation code should be weaved before/after functions. By an automatic method, places of functions invocations are automatically identified and then the relevant observer code is weaved into. To this end, the aspect weaver is used. By using the *weaver* we are able to regularly crosscut the target software, locate the relevant functions and weave the observer code into.

Representing locations and the weaving method (that is, before or after function invocation) are called *joint points*. Each joint point is stated by modifier *pointcut*. Figure 6 shows aspect $Aspect_1$ with pointcut P_1 consisting of *call* instruction and *before* modifier.

The *call* instruction indicates that the *concerns* are functions specified by “void *.credit(float)” and the *before* modifier indicates that the insertion code (The *write* instruction) would be executed before the functions. The insertion code, that is, the *write* instruction is called *advice*. Figure 7 shows how an aspect is weaved into software by the aspect weaver.

In our method, we use concept of aspect oriented in requirements specification level (Garcia-Duque et al., 2006; Baniassad and Clark, 2004; Rashid et al., 2002; Busyairah and Zarinah, 2011) put forward the idea of

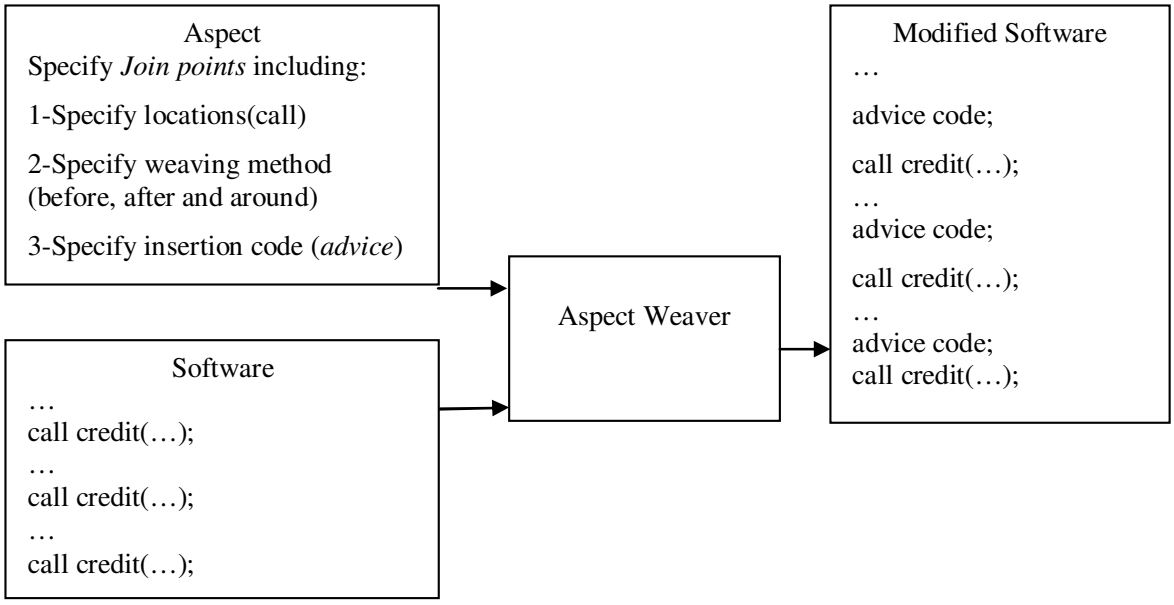

Figure 7. Weaving aspect into software.

aspects for requirements. Aspect is new idea for allocating requirements to components. These components appear in form of classes, packages and services and there are some *constraints* which cut across requirements, that is, components. Such constraints called *crosscutting concerns* in aspect-speak form aspects. Accordingly, an aspect enables us to keep concerns separate in design and implementation phases of system.

In other words, an aspect is a solution for an engineering technique to separate concerns from requirements. The crosscutting leads to *modularization* of the requirements constraints. Accordingly by inspecting the primary requirements, we try to identify aspects. Consider the *security* concern in using an ATM card, for instance; this is a concern that cuts across below requirements. So, it would be considered as an aspect.

R1: ATM needs to send the customer's card and account number to the system for activation.
R2: ATM needs to send the customer's account number to the system to get the identifier of his/her account.

Implementing proposed model

We deal with implementing the model proposed in Figure 4 using event based rules stated in form of event-condition-action. To this end, we start with requirements and constraints. Having identified constraints and requirements, the related probe codes for crosscutting points of requirements are weaved into units of the program. In fact, the created aspects denote the modules that centralize distributed functionality. In order to what events should be captured by the observer, we consider the *constraints* and equip the observer with method calls for obtaining events ("event observing" of in Figure 4). So, events are objects representing execution points of software.

Equipped with event processing mechanism, the target software becomes an active system passing events to the *checker*. The components *condition checking* and *computing and decision* in the checker program (Figure 4) is built from condition and action parts of event-based rules. According to taken decision, the response to the event is made. Below shows the steps we take to implement the proposed model. These steps are practically shown in case study using a case study.

1. Identifying functional requirements
2. Identifying requirements constraints
3. Considering the functions of the target software and event processing functions
4. Specifying event-based rules
5. Specifying aspects

Events may be primitive or composite. A composite appears in form of (E1|E2) meaning that one of the two events E1 or E2 must occur and (E1:E2) meaning that the two events must occur in the given order.

Therefore, our activated system consists of two activated components: (1) The target software whose

behavior to be checked and (2) The checker program that checks the target software. The target software is activated automatically by weaving aspect codes into its source code. The checker program is activated using a set of event-based rules. Therefore, the system becomes event-aware, which can observe and pass events at certain points of the target software.

CASE STUDY

Here, we apply the proposed method to safety critical software of aircraft traffic system. In an air traffic system, there is a region of airspace consisting of a number of aircrafts with a unique identifier for each aircraft. Some requirements implemented by the software and the constraint should be checked are as follows:

Functional requirements:

1. To add an aircraft to the airspace at a specified height,
2. To remove an aircraft from a region. This is carried out if some aircraft moves to an adjacent region.
3. To move an aircraft from one height to another,
4. To lookup an aircraft returning the current height of that aircraft in the region

Constraint:

For safety reasons, all aircraft must be separated by at least *n* meters in height. In fact, there must not be other aircraft at that height or within *n* meters of an aircraft. Also, the vertical separation of an aircraft must be at least *n* meters.

Functional requirements stated previously is implemented by the following functions

```
public Create( ) {
// To create an empty (without any aircraft) region.
  new region;
  region. state=empty;
  return region.no;
  }
public EnterFirst (region, aircraft, height){
// To add an aircraft to a region
  region.state=full;
  aircraft.region_no=region.no;
  return region.no;
  }
public Exit(region, aircraft){
// To remove an aircraft from a region. This operation is used when the aircraft moves to an adjacent region.
   region.state=empty;
   aircraft.region_no=0;
   return region.no;
  }
public EnterAgain(region,aircraft,height){
  // To move an aircraft from one height to another
    Exit(region, aircraft);
    EnterFirst(region, aircraft, height);
    return region.no
  }
public Altitude (region, aircraft) {
// To show height of an aircraft in a region
   return region.height;
   }
public Vacated(region, aircraft) {
// This is a Boolean function returns true if there is not the aircraft in the region.
   if region.no != aircraft.region_no
       return true
        else return false;
public Occupied(region, height){
// This is a Boolean function returns true if there is some aircraft within n meters of that height
  if (region.state = empty) return false;
   if ABS(region.height-hight) ≤ n)
    return true
     else return false;
   }
```

Specifying aspects

As stated previously in, the proposed model, rules are stated in form of event-base, “ON Event When condition DO action”. Therefore, we should identify, (1) *event(s)* that should be observed, (2) *condition(s)* to be considered when the concerned events occur, and (3) *action(s)* to be decided by the checker program. The stated requirements indicate the concerned *event* entry of an aircraft to a region.

Considering the stated constraint, we can obtain *conditions* from functions *Altitude*(), *Vacated*() and *Occupied*(). Having specified the event and conditions, now we can specify the rules:

ON *entry* WHEN *Vacated*(region, aircraft) Alert (“some aircraft in the region”) and Reject request

ON *entry* WHEN Occupied (region, height) Alert (“unsuitable height”) and Reject request

Now, we should take an aspect per rule; however, since both rules were defined on same event, just an aspect is considered. Functions which deal with the event are *EnterFirst*(), *Exit*() and *EnterAgain*(). Accordingly, the concern *entry* cut across these functions. The following code shows the *EntryAspect* in the AspectJ language.

```
Public aspect EntryAspect          // the Aspect
{
   Pointcut Entry ( ):
   call (public no Enter*(region, aircraft, height);
       // join point 1
    before( ): Entry ( ) {
```

```
        if Occupied(region, height)
            Alert ("not suitable height");
      }

   call(public no EnterFirst(region, aircraft,
     height);   // join point 2
   before( ): Entry ( ) {
        if not vacated(region, aircraft)
            Alert ("entrance not permitted");
       }

   call(public no EnterAgain(region, aircraft);
           // join point 3
      before( ): Entry( ) {
 if region = = Create()
              Alert ("no aircraft in region")
           if vacated(region, aircraft)
             Alert ("the aircraft is not in region")
      }

    call(public no Exit(region, aircraft);
        // join point 4
      before( ): Entry( ) {
            if region = = Create()
               Alert ("aircraft not in region")
      }

      call (public height Altitude(region, aircraft);
        // join point 5
      before( ): Entry( ) {
            if region = = Create()
               Alert ("aircraft not in region")

       }
}
```

The *call* functions in aspect *EntryAspect* indicate the points would match in the program. These points are target functions in the software defined previously in, case study. The call function "*Enter**" in aspect *EntryAspect*, for instance, matches functions *EnterFirst* and *EnterAgain* in the target software. Having a call function matched some target function(s), the weaver weaves the advice code of the call function into the target function. The advice codes (denoted by the "Entry" label) in the joint points 1 and 2 are weaved into function *EnterFirst*() and the joint points 1 and 3 are weaved into function *EnterAgain*(). The joint points 4 and 5 are weaved into functions *Exit*() and *Altitude*() respectively.

Conclusion

An event-based model was proposed to check reactive software. To this end, we provided source code of the target software with observer code to pass events to the checker program. Observing and checking presented in form Even-Condition-Action rules where events represented as method calls.

In contrast with the related work stated in related work, the main contribution is presenting a constructive method where check rules are mapped to aspects. This facilitated and automated mapping design of checker program into its implementation.

However, obtaining event-based rules for observing and checking from requirements and constraints was not automated. This can be carried out by bridging the gap between abstract specification of requirements and constraint and event-based rules. Exploitation of event-based rules has additional benefit because a diverse behavior of target software can be checked in terms of which part of the rule(s) is satisfied.

A case study was proposed to show practicality of the proposed model. Employing the proposed mode to distributed and real-time software can be thought as future work.

REFERENCES

Babamir SM, Jalili S (2005). Dynamic analysis of object-oriented programs using state machines and ECA rules. The 14th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE-200S). pp. 243-248.

Babamir SM, Jalili S (2010). Making Real Time Systems Fault Tolerant: a Specification Based Approach. J. Sci. Ind. Res., 69: 501-509.

Baniassad E, Clarke S (2004). Finding aspects in requirements with Theme/Doc. Workshop on Early Aspects. pp. 15-22.

Barringer H, Goldberg A, Havelund K, Sen K (2004). Rule-Based Runtime Verification, The 5th International Conference on Verification, Model Checking, and Abstract Interpretation, LNCS, 2937, Springer. pp. 44-57.

Barringer H, Havelund K, Rydeheard D, Groce A (2009). Rule Systems for Runtime Verification A Short Tutorial. The 9th International Workshop on Runtime Verification, LNCS 5779, Springer. pp. 1-24.

Barringer H, Rydeheard D, Havelund K (2010). Rule Systems for Run-time Monitoring: from EAGLE to RuleR. J. Logic and Comput, Oxford University Press, 20(3): 675-706.

Busyairah SA, Zarinah MDK (2011). An approach for crosscutting concern identification at requirements level using NLP. Int. J. Phys. Sci,. 6(11): 2718-2730.

Carpenter B (2011). The IETF Process: an Informal Guide. http://www.ietf.org/about/process-docs.html

Chakravarthy S, Varkala S (2006). Dynamic programming environment for active rules. The 7th International Baltic Conference on Databases and Information Systems. pp. 3-16.

Chavarría-Báez L, Li X (2006). Structural Error Verification in Active Rule-Based Systems using Petri Nets. The 5th Mexican International Conference on Artificial Intelligence, IEEE Society. pp. 12-21.

Chavarría-Báez L, Xiaoou L (2010). ECAPNVer: A Software Tool to Verify Active Rule Bases. The 22nd IEEE International Conference on Tools with Artificial Intelligence. pp. 138-141.

Cheung AKY, Jacobsen HA (2010). Load balancing content-based publish/ subscribe systems. ACM Transactions on Computer Systems (TOCS). 28(4): Article 9.

d'Amonrim M, Havelund K (2005). Event Based Runtime Verification of Java Programs. The 3rd international workshop on Dynamic Analysis (WODA '05), ACM SIGSOFT Software Engineering Notes. 30(4): 1-7.

Delgado N, Gates AQ, Roach SA (2004). Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. IEEE Transactions on Software Engineering. 30(22): 859-872.

Elliott C (2000). Declarative Event-Oriented Programming. The 2nd ACM

SIGPLAN Int. Conference on Principles and Practice of Declarative Programming. pp. 56-67.
Garcı´a-Duque J, López-Nores M, Pazos-Arias JJ, Fernández-Vilas A, Díaz-Redondo RP, Gil-Solla A, Ramos-Cabrer M, Blanco Fernández Y (2006). Guidelines for the incremental identification of aspects in requirements specifications. J. Requirements Eng., 11(4): 239-263.
Goldsby HJ, Cheng B HC, Zhang J (2008). AMOEBA-RT: Run-time verification of adaptive software, Software Engineering Models in Software Engineering. Workshops and Symposia at MoDELS, Lecture Notes in Computer Science. 5002: 212-224.
Harel D, Politi M (1998). Modeling reactive systems with Statecharts. McGraw-Hill.
Katz S, Mezini M (Eds.) (2011). Transactions on Aspect-Oriented Software Development VIII. Lecture Notes in Computer Science. Vol. 6580.
Koschel A, Astrova I (2008). Event Monitoring Web Services for Heterogeneous Information Systems. World Academy of Science, Engineering and Technology. Vol. 33.
Morgenstern M (1983). Active databases as a paradigm enhanced computing environments. The 9th VLDB International Conference. pp. 34-42.
Pankowski T (1995). Active Objects With Temporal ECA Rules in Intelligent Database Systems. Decentralized Intelligent Multi Agent Systems. pp. 347-354.
Parzyjegla H, Graff D, Schr¨oter A, Richling J, M¨uhl G (2010). Design and Implementation of the Rebecca Publish/Subscribe Middleware, in From Active Data Management to Event-Based Systems and More, Springer, in Sachs K, Petrov I, Guerrero P (Eds.). pp.124-140.
Paton NW, Díaz O (1999). Active database systems. ACM Computing Surveys. 31(1): 63-103.
Rashid A, Sawyer P, Moreira A, Arau´jo J (2002). Early Aspects: A Model for Aspect-Oriented Requirements Engineering. The 10th IEEE International Conference on Requirements Engineering. Pp.199-202.
Reger G (2010). Rule-Based Runtime Verification in a Multicore System Setting. MSc Dissertation, University of Manchester.
Sahoo SK, Man-Lap Li, Ramachandran P, Adve SV, Yuanyuan Z (2008). Using likely program invariants to detect hardware errors, IEEE International Conference on Dependable Systems and Networks with FTCS and DCC. pp. 70-79.
Song J, Ma T, Cadar C, Pietzuch P (2011). Rule-based Verification of Network Protocol Implementations Using Symbolic Execution. The 20th International Conference on Computer Communications and Networks (ICCCN). pp. 1-8.
Widom J, Ceri S (1996). Active database systems-triggers and rules for advanced database processing. Morgan Kaufmann Publishers.