www.ietdl.org

Published in IET Software
Received on 27th January 2009
Revised on 24th April 2010
doi: 10.1049/iet-sen.2009.0009

ISSN 1751-8806

UML interaction model-driven runtime verification of Java programs

X. Li X. Qiu L. Wang X. Chen Z. Zhou L. Yu J. Zhao

State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, Jiangsu, People's Republic of China, Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu, People's Republic of China
E-mail: lxd@nju.edu.cn

Abstract: The authors use unified modelling language (UML) 2.0 interaction overview diagrams (IODs) and sequence diagrams to construct simple and expressive scenario-based specifications, and present an approach to runtime verification of Java programs for exceptional consistency and mandatory consistency. The exceptional consistency requires that any forbidden scenario described by a given IOD never happens during the execution of a program, and the mandatory consistency requires that if a reference scenario described by a given sequence diagram occurs during the execution of a program, it must immediately adhere to a scenario described by a given IOD. In the approach, the authors first instrument a program under verification so as to gather the program execution traces related to a given scenario-based specification; then they drive the instrumented program to execute for generating the program execution traces; finally they check if the collected program execution traces satisfy the given specification. The approach leads to a supporting tool for testing in which UML interaction models are used as automatic test oracles to detect the wrong temporal ordering of message interaction in programs.

1 Introduction

Runtime verification [1–4] is a lightweight approach to program reliability. Its basic idea is to gather information during program execution and use it to conclude properties about the program, either during testing or in operation, which increases the confidence in whether the program implementation conforms to its specifications.

Scenarios are widely used as a requirements technique because they describe concrete interactions and are therefore easy for customers and domain experts to use. Scenario-based specifications such as message sequence charts [5] and unified modelling language (UML) interaction models [6, 7] offer an intuitive and visual way of describing system requirements. For object-oriented programs, such specifications focus on the temporal ordering of message interactions among objects, which forms an important aspect of system behaviour.

The program specifications used in runtime verification are typically represented by formal languages such as temporal logic [8], regular expressions [9] or state machines [10]. Since UML became a standard in object management group (OMG) in 1997, UML interaction models have become an important class of artefacts in software development processes [11]. In this paper, we use simple UML interaction models as scenario-based specifications, and consider runtime verification of Java programs.

UML sequence diagrams form a class of important UML interaction models. Each of them describes an interaction, which is a set of messages exchanged among objects within a collaboration to effect a desired operation or result, and its focus is on the temporal ordering of the message flow [6, 7]. For example, an UML sequence diagram D is depicted in Fig. 1*a*, which describes a scenario about the well-known example of the railroad crossing system in [12, 13]. This system operates a gate at a railroad crossing, in which there are a railroad crossing monitor and a barrier controller. When the monitor detects that a train is arriving, it sends a message to the controller to move down the barrier. After the train leaves the crossing, the monitor sends a message to the controller to open the barrier.

For facilitating the use of scenario-based specifications, in this paper we just adopt a simplified version of UML sequence diagrams, which describes exactly one scenario without any alternative and loop. For describing multiple scenarios and complete system specifications, we use a simplified version of UML2.0 interaction overview diagrams (IODs) [7], which focuses on the overview of the flow of control where the nodes are sequence diagrams. For example, Fig. 1*b* depicts a simple IOD G.

In this paper, we focus on checking Java programs for message interaction consistency. In object-oriented programs, we often need to set some restrictions on the temporal ordering of message interaction along the program execution flow, which form a class of safety requirements. For describing such requirements, we introduce the following four kinds of specifications which are depicted in Fig. 2:

- 'Exceptional consistency specifications' require that any forbidden scenario described by a given IOD D never happens during the execution of a program.

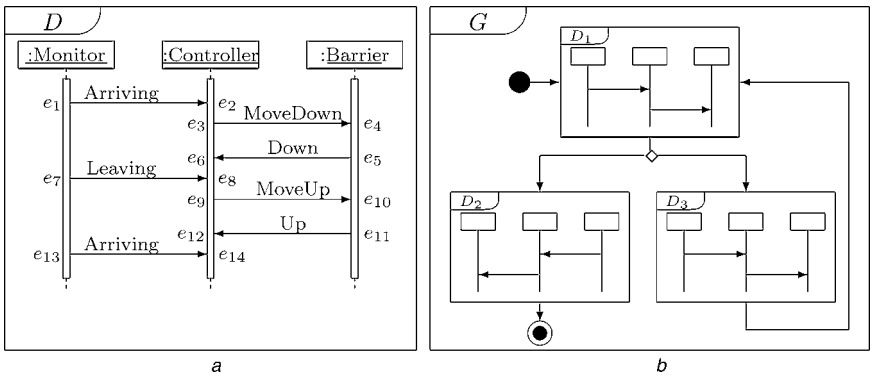


Fig. 1 *UML interaction models*

a Sequence diagram *D*
b Interaction overview diagram *G*

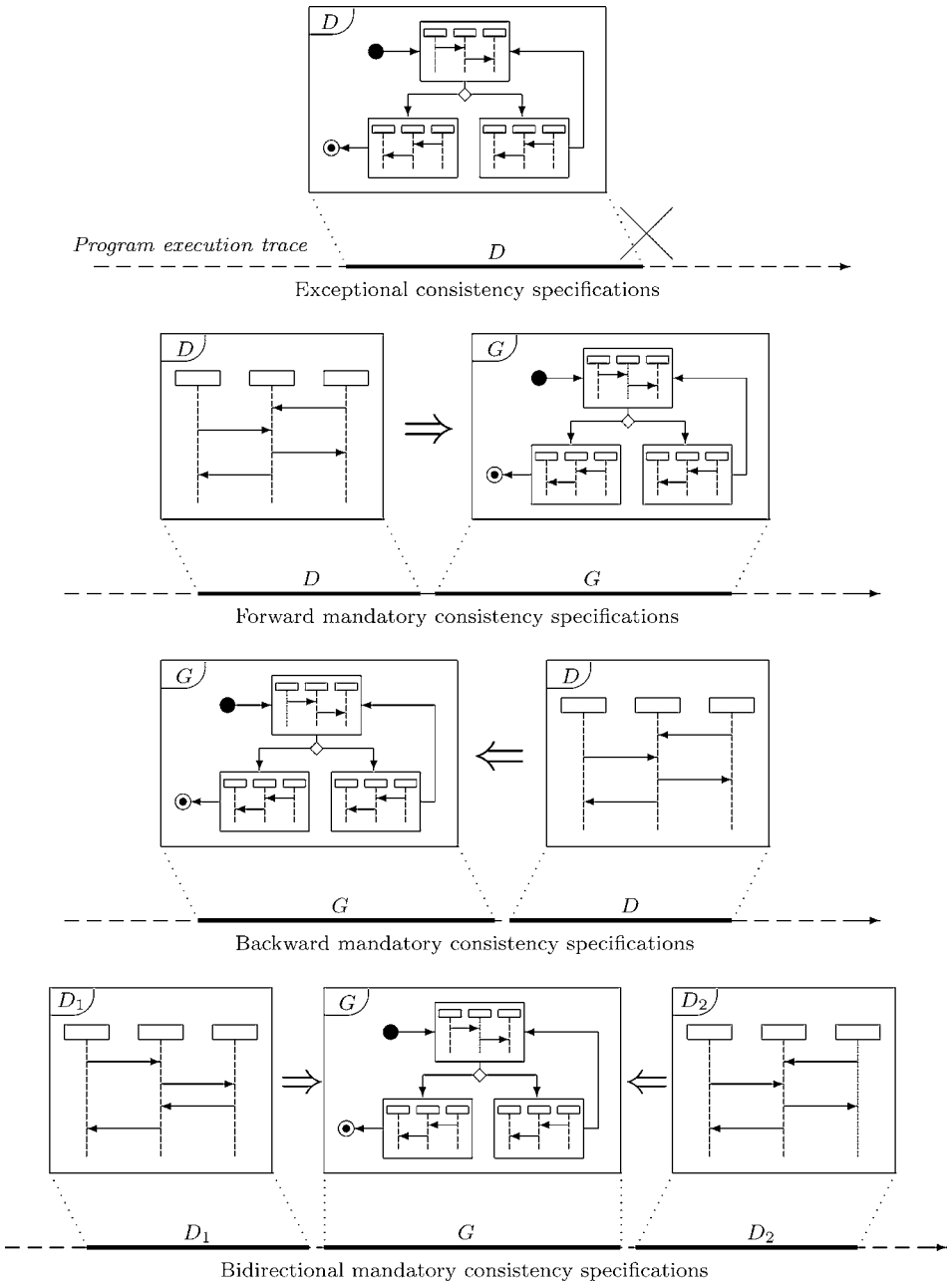


Fig. 2 *Scenario-based specifications*

• 'Forward mandatory consistency specifications' require that if a reference scenario described by a given sequence diagram D occurs during the execution of a program, then a scenario described by a given interaction overview diagram G must follow immediately.
• 'Backward mandatory consistency specifications' require that if a reference scenario described by a given sequence diagram D occurs during the execution of a program, then it must follow immediately from a scenario described by a given IOD G.
• 'Bidirectional mandatory consistency specifications' require that if a reference scenario described by a given sequence diagram D_1 occurs during the execution of a program and another reference scenario described by another given sequence diagram D_2 follows, then in between these two scenarios, a scenario described by a given IOD G must occur exactly.

In this paper, we present an approach to runtime verification of Java programs for the above four kinds of scenario-based specifications. As depicted in Fig. 3, the runtime verification process is mainly composed of three steps. First, guided by UML interaction models in a given scenario-based specification we instrument a program under verification so as to trace the corresponding events in program execution. Then, the instrumented program is driven to execute with given test cases, and the program execution traces are gathered by the instrumented probes in the program. Last, we check if the collected program execution traces are consistent with the specification on temporal ordering of message interaction.

Instead of checking program execution traces online, which is mainly used in program monitoring, our approach does the consistency checking off-line, which falls into the field of testing. Our purpose is to develop a supporting tool for testing in which UML interaction models are used as automatic test oracles, and use it to detect the wrong temporal ordering of message interaction in programs.

The paper is organised as follows. In the next section, we introduce UML interaction models, and give their formal definitions for runtime verification. Section 3 introduces the scenario-based specifications considered in this paper, which are expressed by UML interaction models. The detailed approach is given in Section 4 to the runtime verification of Java programs for scenario-based specifications. The related works are discussed in Section 5, and some conclusions are given in the last section.

2 UML interaction models

In this paper, UML interaction models are used as scenario-based specifications for runtime verification of Java programs, which consist of UML2.0 IODs and sequence diagrams.

2.1 Sequence diagrams

Here we just use a simplified version of sequence diagrams, which describe exactly one scenario without any alternative and loop. A sequence diagram considered in this paper has two dimensions: the vertical dimension represents time and the horizontal dimension represents different objects. Each object is assigned a column, and the messages are shown as horizontal, labelled arrows.

In a sequence diagram, by events we mean the message sending and the message receiving. The semantics of a sequence diagram essentially consists of the sequences (traces) of the message sending (receiving) events. The order of events (i.e. message sending or receiving) in a trace is deduced from the visual partial order determined by the flow of control within each object in the sequence diagram along with a causal dependency between the events of sending and receiving a message [5–7, 14]. In accordance with [14], without losing generality, we assume that for a pair of events e and e' in a sequence diagram, e precedes e' (denoted by $e \prec e'$) in the following cases:

• *Causality:* e is a sending event, and e' is its corresponding receiving event. For example, in the sequence diagram depicted in Fig. 1*a*, e_5 precedes e_6.
• *Controllability:* The event e appears above the event e' on the same object column, and e' is a sending event. This order reflects the fact that a send event can wait for other events to occur. On the other hand, we sometimes have less control on the order in which receive events occur. For example, in the sequence diagram depicted in Fig. 1*a*, e_6 precedes e_9.
• *FIFO order:* The receiving event e appears above the receiving event e' on the same object column, and the corresponding sending events e_1 and e'_1 appear on a mutual object column where e_1 is above e'_1. For example, in the sequence diagram depicted in Fig. 1*a*, since the receiving event e_4 appears above the receiving event e_{10} on the barrier object and their corresponding sending events e_3 and e_9 appear together on the controller object where e_3 is above e_9, e_4 precedes e_{10}.

For giving the formal definition of scenario-based specifications, we formalise sequence diagrams as follows.

Definition 1: A sequence diagram is a tuple $D = (O, E, M, L, V)$ where

• O is a finite set of objects. For each object $o \in O$, we use $\zeta(o)$ to denote the class which o belongs to.
• E is a finite set of events corresponding to sending or receiving a message.
• M is a finite set of messages. Each message in M is of the form (e, g, e') where $e, e' \in E$ corresponds to sending and

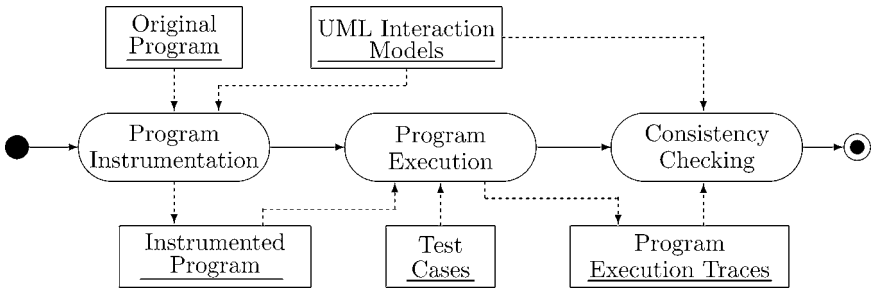


Fig. 3 *Runtime verification process*

doi: 10.1049/iet-sen.2009.0009

receiving the message, respectively, and g is the message name which is a character string.

- $L : E \rightarrow O$ is a labelling function which maps each event $e \in E$ to an object $L(e) \in O$ which is the sender (receiver) while e corresponds to sending (receiving) a message.
- V is a finite set whose elements are pairs (e, e') $(e, e' \in E)$ such that $e \prec e'$, which defines a visual order.

We use 'event sequences' to represent the 'traces' of sequence diagrams, which describes the temporal ordering of the message flow. An event sequence is of the form $e_0 \hat{} e_1 \hat{} \cdots \hat{} e_m$, which represents that e_{i+1} takes place after e_i for any i $(0 \leq i \leq m-1)$.

Definition 2: For any sequence diagram $D = (O, E, M, L, V)$, an event sequence $e_0 \hat{} e_1 \hat{} \cdots \hat{} e_m$ is a trace of D if and only if the following condition holds:

- $e_0, e_1, \ldots, e_m$ is a permutation of the events in E, and
- $e_0, e_1, \ldots, e_m$ satisfy the visual order defined by V, that is, for any e_i $(0 \leq i \leq m)$ and e_j $(0 \leq j \leq m)$, if $(e_i, e_j) \in V$, then $0 \leq i < j \leq m$.

The formal definitions of message sequence charts and sequence diagrams have been discussed in [14–21] for various verification purposes, and the general formal definitions for UML2.0 sequence diagrams have also been given in [22–24]. Our definition here does not differ from those definitions essentially, but are based on the simple version of sequence diagrams and the specific verification purpose in this paper.

2.2 Interaction overview diagrams

A sequence diagram considered in this paper just describes one scenario. For describing multiple scenarios, we need to use a simplified version of UML2.0 IODs [7], which focuses on the overview of the flow of control where the nodes are simple sequence diagrams. An IOD defines a composition of a set of sequence diagrams, which describes potentially iterating and branching system behaviour.

For example, Fig. 4 depicts an IOD, which specifies the FIPA Iterated Contract Net Iteration Protocol [25]. This protocol implements the interaction between the agents Initiator and Participant such that the Initiator seeks to get better bid from the Participant by modifying the call and requesting a new (equivalently, revised) bid. The Initiator issues an initial call for proposals with the cfp message (in sequence diagram cfp). If the Participant is willing and able to do the task under the proposed conditions (in sequence diagram propose), then it replies a propose message, otherwise it can refuse (in sequence diagram refuse). When receiving a propose message, the Initiator may decide that this is the final iteration and accept the bid (in sequence diagram inform and failure), or reject it (in sequence diagram reject). After the Initiator accepts the bid, once the Participant completes the task, it sends a inform message to the Initiator (in sequence diagram inform). However, if the Participant fails to complete the task, a failure message is sent (in sequence diagram failure). Alternatively the Initiator may decide to iterate the process by issuing a revised cfp to the Participant (in sequence diagrams propose and cfp). The process terminates when the Initiator refuses a proposal and does not issue a new message cfp, the Initiator accepts a bid, or the Participant refuses to bid.

Definition 3: An IOD is a tuple

$$G = (U, N, \text{succ}, \text{ref})$$

where

- U is a finite set of sequence diagrams;
- $N = \{\top\} \cup I \cup \{\bot\}$ is a finite set of nodes partitioned into the three sets: the singleton-set of *start* node, the set of *intermediate* nodes and the singleton-set of *end* node, respectively;
- succ $\subset N \times N$ is the relation which reflects the connectivity of the nodes in N such that any node in N is reachable from the start node; and
- ref: $I \mapsto U$ is a function that maps each intermediate node to a sequence diagram in U.

For an IOD $S = (U, N, \text{succ}, \text{ref})$, a 'path segment' is a sequence of intermediate nodes $v_0 \hat{} v_1 \hat{} \cdots \hat{} v_n$ satisfying that $(v_{i-1}, v_i) \in \text{succ}$ for any i $(0 < i \leq n)$. A 'path' is a path segment $v_0 \hat{} v_1 \hat{} \cdots \hat{} v_n$ such that $(\top, v_0) \in \text{succ}$ and $(v_n, \bot) \in \text{succ}$.

In UML2.0, IODs are defined as a specialisation of activity diagrams in a way that promotes overview of the control flow [7]. It follows that the concatenation of two sequence diagrams in an IOD should be interpreted as the 'synchronous mode',

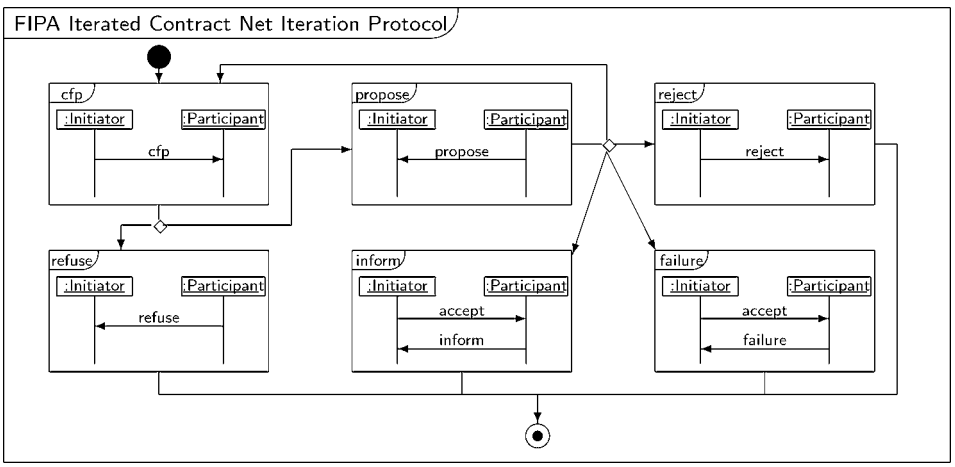


Fig. 4 *IOD specifying the FIPA Iterated Contract Net Iteration Protocol*

which means that when moving one node to the other, all events in the previous sequence diagram finish before any event in the following sequence diagram occurs, which is the same as the synchronous interpretation in MSC specifications [19]. Therefore, we define the 'traces' of an IOD G as the event sequences, which are the concatenation of the traces of the sequence diagrams that make up G. We use ^ to denote the concatenation of sequences.

Definition 4: For an IOD $G = (U, N, \text{succ}, \text{ref})$, an event sequence

$$\sigma = e_0 \hat{} e_1 \hat{} \cdots \hat{} e_n$$

represents a trace of G if and only if there is a path $v_0 \hat{} v_1 \hat{} \cdots \hat{} v_m$ in G such that $\sigma = \sigma_0 \hat{} \sigma_1 \hat{} \cdots \hat{} \sigma_m$, where σ_i is a trace of $\text{ref}(v_i)$ for each i $(0 \leq i \leq m)$.

3 Scenario-based specifications

In the above section, we have introduced the simple versions of sequence diagrams and IODs, which are used to construct the scenario-based specifications in our runtime verification approach. In UML 2.0 [7], sequence diagrams support the notations of branch, iteration and parallel themselves, which can be used to describe the compositions of simple scenarios. Instead of using those notations in sequence diagrams to compose simple scenarios, we use simple sequence diagrams to describe exactly one scenario without any alternative and loop, and simple IODs to construct scenario compositions. Such a hierarchical way is beneficial to construct large-scale and complex specifications and facilitate the use of specifications. As the IODs can compose simple scenarios through alternatives, parallel and loops, our specifications can express any scenarios described using common sequence diagrams.

As we will discuss below, UML interaction models considered in this paper are used to construct expressive scenario-based specifications, which are the exceptional consistency specifications and mandatory consistency specifications (including forward, backward and bidirectional mandatory consistency specifications), and these consistency specifications can express many important properties frequently concerned in safety critical systems. An exceptional consistency specification consists of one IOD G, denoted by $\mathcal{S}_S(G)$, and requires that any forbidden scenario described by G never happens during the execution of a program. For example, there are two interaction models depicted in Fig. 5, which are about the railway crossing system. The left one is a sequence diagram D that describes a normal scenario for the preparation for the train crossing, which should occur during the program execution. The right one is an IOD G specifying an exceptional scenario in which the message Crossing_secured is sent to the monitor before the barrier is put down, which is forbidden to occur during the program execution, and forms an exceptional consistency specification. The forbidden scenarios represent the negative requirements derived during requirement analysis.

A forward mandatory consistency specification consists of a sequence diagrams D and an IOD G, denoted by $\mathcal{S}_F(D, G)$, and requires that if a reference scenario described by D occurs during the execution of a program, then a scenario described by G must follow immediately. For example, a forward mandatory consistency specification for the railway crossing system is depicted in Fig. 6, which requires that from the scenario for the preparation for the train crossing, the scenario for raising the barrier after the train passes must follow immediately. Such kind of specifications can specify that the system under verification must leave dangerous states in time.

A backward mandatory consistency specification consists of a sequence diagram D and an IOD G, denoted by $\mathcal{S}_B(D, G)$, and requires that if a reference scenario described by D occurs during the execution of a program, then it must follow immediately from a scenario described by G. For example, a backward mandatory consistency specification for the railway crossing system is depicted in Fig. 7, which requires that the scenario for raising the barrier after the train passes must follow immediately from

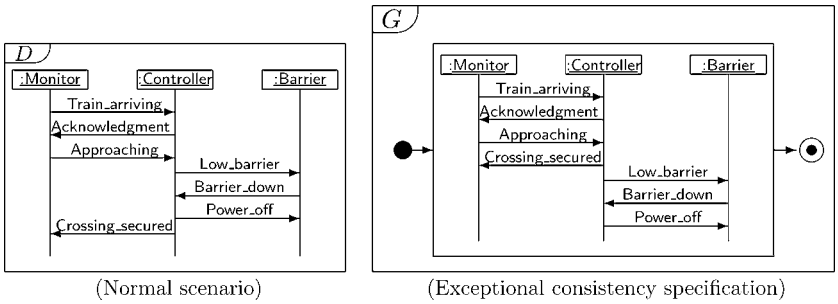


Fig. 5 *UML interaction models for the railway crossing system*

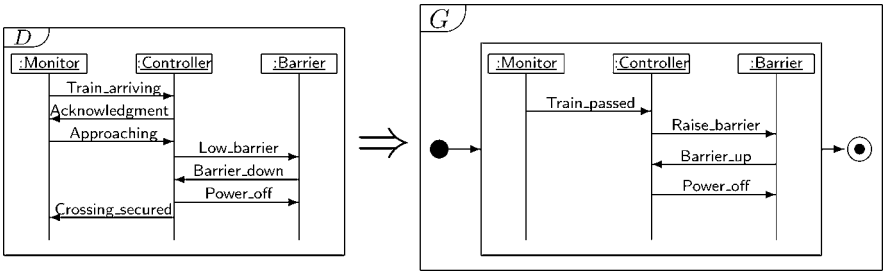


Fig. 6 *Forward mandatory consistency specification for the railway crossing system*

doi: 10.1049/iet-sen.2009.0009

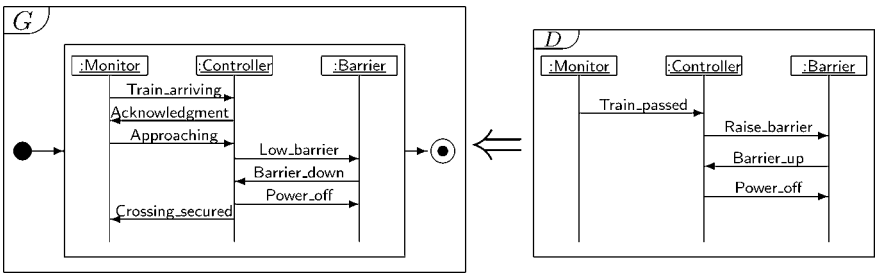


Fig. 7 *Backward mandatory consistency specification for the railway crossing system*

the scenario for the preparation for the train crossing. Such kind of specifications can specify that a critical action can only take place after a premises action occurs.

A bidirectional mandatory consistency specification consists of two sequence diagrams D_1 and D_2, and an IOD G, denoted by $\mathcal{S}_{\mathrm{D}}(D_1, D_2, G)$, and requires that if a reference scenario described by D_1 occurs during the execution of a program and a reference scenario described by D_2 follows, then in between these two scenarios, a scenario described by G must occur exactly. For example, a bidirectional mandatory consistency specification for the railway crossing system is depicted in Fig. 8, which requires that between the scenarios for confirming the train arriving and for permitting the train crossing, the scenario for lowering the barrier must exist exactly.

The UML interaction models in requirements and design given by customers or experts could be directly reused as scenario-based specifications, and they could be incomplete provided they describe a complete concerned property. But sometimes it is necessary to give an elaborate design for the specifications. For example, we often need to construct a mandatory consistency specification for testing and verification purpose. Suppose that we attempt to detect some errors in a program related to a given scenario, which means that the scenario is not implemented correctly in the program. Since the scenario will not occur during the program execution if it is not implemented correctly, it is difficult for us to decide where and when the scenario should occur in order to find the related errors further. In this case, we can decompose the scenario into two parts that constitute a mandatory consistency specification, and use one part as a reference scenario, and test or verify if there is any error in the other part, which is depicted in Fig. 9.

Since an IOD defines a composition of a set of sequence diagrams, the scenario-based specifications we present here are essentially composed of sequence diagrams. For a scenario-based specification, we define its 'object set' as the union of the object sets of all the sequence diagrams occurring in the specification. Furthermore, we can extend each sequence diagram occurring in a scenario-based specification such that its object set is just the object set of the scenario-based specification. Therefore, without losing generality, we assume that all sequence diagrams occurring in a scenario-based specification focus on the same set of objects, that is, they describe the interaction scenarios on the same set of objects.

In our runtime verification approach, since the scenario-based specifications are used for checking program execution traces, we need to map the objects in a scenario-based specification to the ones in a program under verification. Notice that in many cases an object in a sequence diagram has no name (the object is just assigned with its class name), and that since during the execution of a program there are multiple objects belonging to the same class, there may be multiple object compositions corresponding to the object set of a given scenario-based specification. Therefore for a scenario-based specification $\mathcal{S}$ and a program P, we map one object of class A in the object set of $\mathcal{S}$ to all objects of class A in P, and the object set of $\mathcal{S}$ to all corresponding object compositions in P (illustrated in Fig. 10), which means that $\mathcal{S}$ is enforced on all the corresponding object compositions in P.

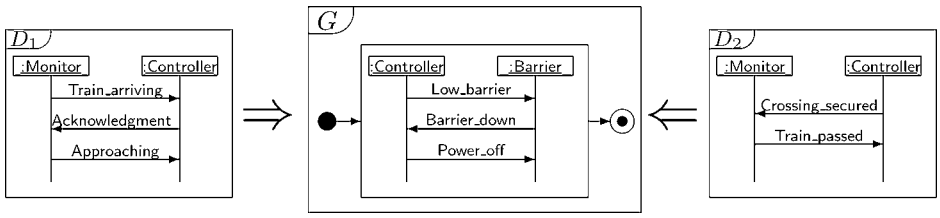


Fig. 8 *Bidirectional mandatory consistency specification for the railway crossing system*

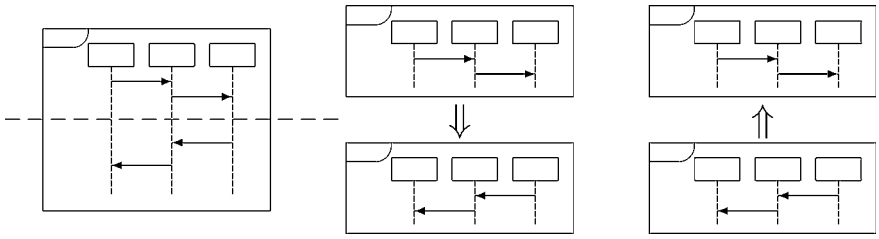


Fig. 9 *Decomposing scenarios under verification into mandatory consistency specifications*

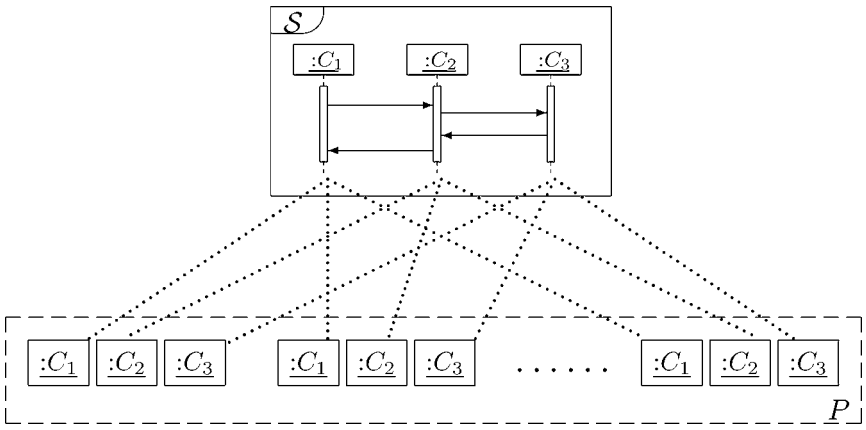


Fig. 10 *Mapping objects between a scenario-based specification and a program*

4 Runtime verification of Java programs for scenario-based specifications

In this section, we give the details of the approach to runtime verification of Java programs for the scenario-based specifications expressed by UML interaction models. The verification process consists of three main steps: program instrumentation, program execution and consistency checking, which is depicted in Fig. 3.

4.1 Program instrumentation

The purpose of program instrumentation is to trace all the events involved in a given scenario-based specification. For a Java program under verification, we insert some instructions into its bytecode. Compared with source code instrumentation, bytecodes instrumentation brings more flexibility, since it is impossible to obtain applications' source codes in many cases.

For a Java program under verification, all the sending or receiving events of a concerned message in a given scenario-based specification must be logged for the runtime verification purpose. For each event, the logged information should include the message type, the message sender or receiver, and the class of the sender or receiver.

In a Java program, a method call corresponds to a message sending event, and the beginning of a method execution corresponds to a message receiving event. Thus we insert instructions for information gathering before each relevant invoke instruction and at the beginning of each relevant method body. For our verification purpose, we still need to pair each sending event and its corresponding receiving event. This task is not a trivial one because in parallel Java programs, the sending events and receiving events of several messages may interleave with each other. We solve this problem based on the fact that in one process a sending event and its corresponding receiving event are always executed in the same thread and they must happen continuously in that thread [26]. So we log also the ID of the thread in which the method (and also the inserted instructions) is executed. Thus, we can pair each logged sending event with the next receiving event with the same thread ID.

The instrumentation algorithm is depicted in Fig. 11*a*. Let $D = (O, E, M, L, V)$ be a sequence diagram in a given scenario-based specification. For a message $m = (e, g, e')$ in M, we use m.method to denote the corresponding method in a program under verification. This method is defined in $\zeta(L(e'))$ (the class of the receiver of m) or in an ancestor of $\zeta(L(e'))$ (if it is an inherited one). We can statically find the method definition based on the class hierarchy. This method begins its execution when a receiving event e' of m happens. To log the receiving events, our instrumentation algorithm inserts some instructions at the beginning of m.method. These inserted instructions invoke the method 'Logger.logEvent' using 'this' (the receiver), the string 'meth_exec', and the method name as the real parameters. Fig. 11*b* shows an example of such instructions. These instructions first push the real parameters into the stack, then invoke the method Logger.logEvent. For each sending event e of m, it is triggered in a method of $\zeta(L(e))$ (the class of the sender of m) and corresponds to an invoke instruction in the method. This method is defined in $\zeta(L(e))$ or in an ancestor of $\zeta(L(e))$. There are four kinds of JVM invoke instructions: INVOKEVIRTUAL, INVOKEINTERFACE, INVOKESPECIAL and INVOKESTATIC. The INVOKESTATIC and INVOKESPECIAL instructions are out of the scope of our consideration because they are used to invoke class methods, instance initialisation methods, private methods of 'this' and methods in a superclass of 'this'. The format of an INVOKEVIRTUAL instruction is

$$\text{invokevirtual} < \text{methodSpec} >$$

where methodSpec specifies a class and a method. Such an instruction may send the message $m = (e, g, e')$ only if the specified class is $\zeta(L(e'))$ or an ancestors of $\zeta(L(e'))$ and the specified method is g. The format of an INVOKEINTERFACE instruction is

$$\text{invokeinterface} < \text{methodSpec} >$$

where methodSpec specifies an interface and a method. Such an instruction may send the message $m = (e, g, e')$ only if the class $\zeta(L(e'))$ implements the specified interface and the specified method is g. To log the sending events, our algorithm inserts instructions before each of these invoke instructions. The inserted instructions are similar to those depicted in Fig. 10*b*. However, they invoke the method Logger.logEvent using 'this' (the sender), the string 'meth-call', and the method name as the real parameters. The instructions inserted by the instrumentation algorithm may log some irrelevant sending/receiving events. For example, an object of an ancestor of $\zeta(L(e'))$ may also execute m.method so that an irrelevant receiving event is logged, and an invoke instruction may also invoke the method g of an ancestor of $\zeta(L(e'))$, which makes the instructions inserted before it log irrelevant sending events. To solve

doi: 10.1049/iet-sen.2009.0009

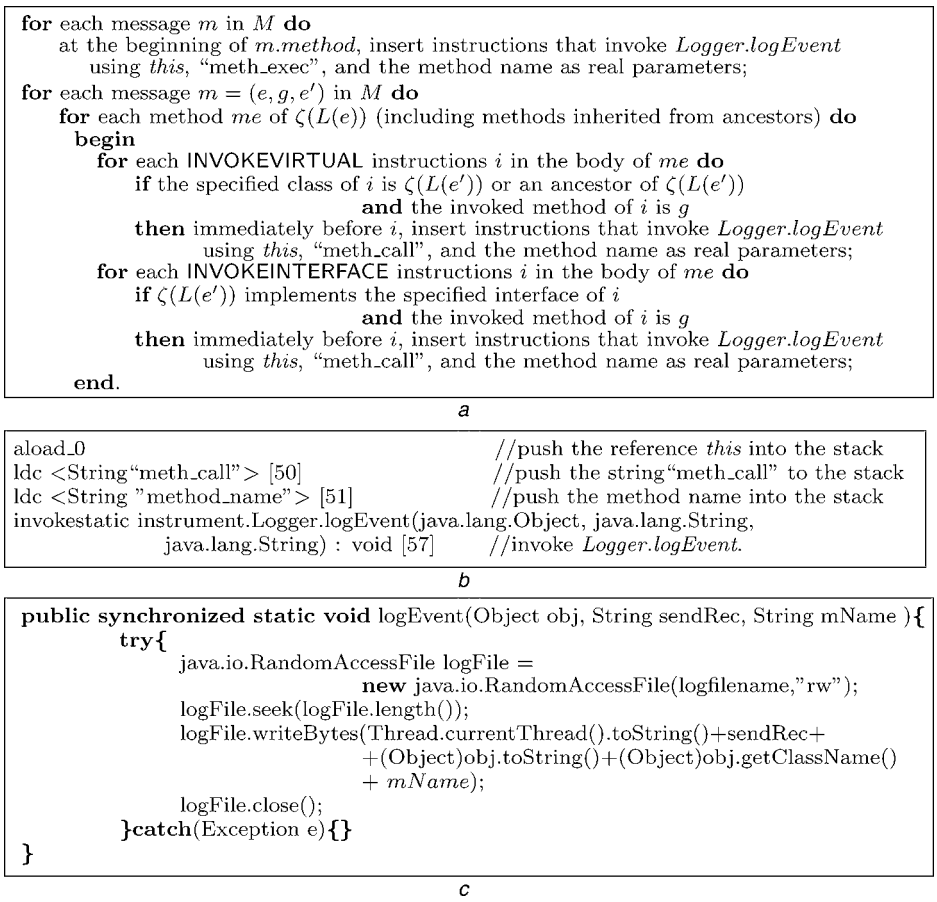


Fig. 11 *Instrumentation algorithm and inserted code segments*

a Instrumentation algorithm
b Example of the instrumented instructions
c Method LogEvent of the class logger

this problem, we also log the runtime class names of message senders/receivers (see Fig. 10*c*) so that those irrelevant events can be filtered out easily before the off-line consistency checking.

The method Logger.logEvent is depicted in Fig. 11*c*. This method logs all sending/receiving events. It calls Thread.currentThread to obtain the thread ID, calls (Object)obj.getClassName to obtain the runtime class name of the sender/receiver and calls (Object)obj.toString to obtain the ID string of the sender/receiver. Before running the instrumented version of a program, the bytecode of 'Logger' should be put into proper directory so that this method can be invoked by the inserted instructions.

The (Object)obj.toString method defined by the class Object does return distinct hash codes for distinct objects, but since this is implemented by converting the internal address of the object into a hash code, it is still possible for different objects that exist in different time to return the same hash code. To determine the life cycles of these dynamic objects, we need to instrument the finaliser of the concerned classes. Whenever one object is finalised, its hash code is logged so that we know what a hash code exactly refers to at different time.

There are two assumptions for the above instrumentation method. One is that object creation/destruction and return messages are ignored in our scenario-based specifications so that we do not need to instrument the corresponding codes for tracing the corresponding events. The other assumption is that all the objects occurring in a scenario-based specification belong to one process in the program under verification (no distributed object exists), which assures that any sending event and its corresponding receiving event are always executed in the same thread. According to the Java virtual machine specification [26], an object created by a process cannot be accessed by other processes. That means one object cannot be shared among several processes. Only threads belonging to the same process can share one object with each other. Under such circumstances, a sending event and its corresponding receiving event are always in the same thread. Notice that now with the support of various middlewares an object can be accessed by any process distributed in a network, but this case is away from the main verification focus in this paper. Therefore although the above two assumptions form a light limitation of our approach, we think they do not influence the entire verification purpose.

4.2 Program execution

For a program under verification, we gather its execution traces by running its instrumented version. The program executions are driven by previous prepared test inputs in a data pool. During system-level program execution, all the test inputs are what the users need to provide by keyboard and/or mouse operation. We can also directly reuse the test data pool if there exists one for the system testing.

Owing to absence of sufficient real-world data, creating suitably test data is often a difficult task. In our approach, test data are generated mainly by random method. Random

test generation is a black-box technique, and needs no information on the internal structure of a program other than the input type and domain. Random method is inexpensive charge and could be implemented in an automatic fashion. Randomness can also increase the variety of input values so as to exercise and profile different behaviour of a program under verification.

Having only the test inputs are not enough for the execution of a program. How and when is the test data fed to the program? How the program is executed? These problems should be solved before execution. We need a driver to ensure the program execution process in a mode without human intervention. The driver activates a program under verification, controls the execution, obtains test data from the data pool and feeds the test inputs to the program upon request.

In our approach, we provide a heuristic wizard in interactive mode to customise the random test data generation. We assume that users have the knowledge of input type and domain. Users need to specify the input sequence, type, domain and sample number. This allows us to take advantage of randomness but still have control over test input generation for the program execution at the system level. Here we just handle simple input type such as integer, real, char, enumerable set and so on.

4.3 Consistency checking

According to the algorithm for instrumenting programs given in Section 4.1, for an event corresponding to sending (receiving) a message, we can obtain its sender (receiver) and the class the sender (receiver) belongs to. We also can pair a sending event and its corresponding receiving event for the same message. For simplicity, from now on we represent any program execution trace we gather by a sequence which is of the form $\varepsilon_0\hat{}\varepsilon_1\hat{}\cdots\hat{}\varepsilon_n$ where each ε_i $(0 \le i \le n)$ is an event corresponding to sending (receiving) a message, and the class which the sender (receiver) of ε_i belongs to is denoted by $\tau(\varepsilon_i)$.

According to Section 3, the object set of a scenario-based specification is mapped to all the corresponding object compositions in a program under verification. In a program execution trace, since the events may be triggered by the different objects with the same class, there may be multiple scenarios generated by different object compositions, and thus during consistency checking we need to consider the scenarios generated by those object compositions respectively, that is, we select one object composition at a time and check their message sending or receiving event sequence in the program execution trace for the specification without considering the events triggered by other objects. Therefore for simplicity we assume that in any program execution trace we consider, there is just one object composition corresponding to the object set of a given scenario-based specification, that is, any program execution trace satisfies that there is a bijection function that maps each object triggering the events in the trace to an object with the same class in the object set of the given specification.

For matching the program execution traces and the traces of a given sequence diagram, we define the 'trails' of a sequence diagram as follows. Given a sequence diagram $D = (O, E, M, L, V)$, a program execution trace $\varepsilon_0\hat{}\varepsilon_1\hat{}\cdots\hat{}\varepsilon_n$ is a trail of D if it can be mapped into a trace of D, that is, there is a corresponding trace of D of the form $e_0\hat{}e_1\hat{}\cdots\hat{}e_n$ such that

- for each i $(0 \le i \le n)$, the class which the sender or receiver of ε_i belongs to is the same as the one which the sender or receiver of e_i belongs to, that is, $\tau(\varepsilon_i) = \zeta(L(e_i))$;
- for each i $(0 \le i \le n)$, if e_i is a message sending (receiving) event, then ε_i corresponds the same message sending (receiving) event; and
- if (e_i, g, e_j) is in M $(0 \le i < j \le n)$, then ε_i and ε_j is a pair of the sending and receiving events for the same message.

Similarly, we define the trails of an IOD G as a program execution trace, which is the concatenation of the trails of the sequence diagrams that make up G. Given an IOD $G = (U, N, \text{succ}, \text{ref})$, a program execution trace $\rho = \varepsilon_0\hat{}\varepsilon_1\hat{}\cdots\hat{}\varepsilon_n$ is a trail of G if it can be mapped into a trace of G, that is, there is a path $v_0\hat{}v_1\hat{}\cdots\hat{}v_m$ in G such that

- $\rho = \rho_0\hat{}\rho_1\hat{}\cdots\hat{}\rho_m$, and
- ρ_i is a trail of $\text{ref}(v_i)$ for each i $(0 \le i \le m)$.

In the following, we give the solutions to exceptional consistency checking and mandatory consistency checking.

4.3.1 Exceptional consistency checking: Let $\mathcal{S}_S(G)$ be an exceptional consistency specification which consists of one IOD G. It requires that any forbidden scenario described by G never happens during the execution of a program. For a program execution trace $\rho = \varepsilon_0\hat{}\varepsilon_1\hat{}\cdots\hat{}\varepsilon_n$, if there is a subsequence $\varepsilon_i\hat{}\varepsilon_{i+1}\hat{}\cdots\hat{}\varepsilon_j$ $(0 \le i < j \le n)$ in ρ which is a trail of G, then we say that a scenario described by G occurs in ρ. Thus, we define that a program execution trace satisfies $\mathcal{S}_S(G)$ if no scenario described by G occurs in the program execution trace.

Let $G = (U, N, \text{succ}, \text{ref})$ be an IOD. For a program execution trace $\rho = \varepsilon_0\hat{}\varepsilon_1\hat{}\cdots\hat{}\varepsilon_n$, a path segment $v_0\hat{}v_1\hat{}\cdots v_m$ in G such that $(\top, v_0) \in \text{succ}$ is a 'pre-left image' of ρ if there is i $(0 \le i \le n)$ such that

$$\varepsilon_0\hat{}\varepsilon_1\hat{}\cdots\hat{}\varepsilon_i = \rho_0\hat{}\rho_1\hat{}\cdots\hat{}\rho_m$$

where each ρ_j $(0 \le j \le m)$ is a trail of $\text{ref}(v_j)$. It is clear that if there is a path in G which is a pre-left image of ρ, then a scenario of G occurs in ρ, and we call such a path by 'left image' of ρ. We can develop an algorithm to check if there is a left image of ρ in G, which is depicted in Fig. 12. The algorithm traverses each path in G in a depth first manner from the start node $\top$. The path segment we have so far traversed is stored in a list variable 'currentpath'. For each successive node 'node' of the last node of 'currentpath', we first check whether it is such that the path segment corresponding to currentpath is a left image of ρ. If yes, then return 'true', and we are done. If 'node' is such that the path segment corresponding to 'currentpath' is a pre-left image of ρ, then we add node to currentpath and start the search from it, otherwise we search the other successive nodes. The algorithm backtracks when all the successive nodes of the last node of currentpath are explored. Based on this algorithm, the exceptional consistency checking of G for ρ is simple. We just need to check if there is a left image in G for each subsequence $\varepsilon_k\hat{}\varepsilon_{k+1}\hat{}\cdots\hat{}\varepsilon_n$ $(0 \le k \le n)$. This algorithm can also be used in the forward mandatory consistency checking, which will described in the following subsection.

Similarly, we define the 'pre-right images' and 'right images' of a program execution trace $\rho = \varepsilon_0\hat{}\varepsilon_1\hat{}\cdots\hat{}\varepsilon_n$ in an IOD $G = (U, N, \text{succ}, \text{ref})$. A path segment $v_0\hat{}v_1\hat{}\cdots v_m$

doi: 10.1049/iet-sen.2009.0009

```
currentpath := ⟨⊤⟩;
repeat
   node := the last node of currentpath;
   if all successive nodes of node are explored through currentpath
   then /*backtracking*/ delete the last node of currentpath
   else begin /*explore an unexplored successive node through currentpath*/
          node := a successive node of node not explored through currentpath;
          if node is such that the path segment corresponding to currentpath
             is a left image of ρ
          then return true;
          if node is such that the path segment corresponding to currentpath
             is a pre-left image of ρ
          then append node to currentpath;
        end
until currentpath = ⟨⟩;
return false.
```

Fig. 12 *Algorithm for checking left images in an IOD*

in G such that $(v_m, \perp) \in$ succ is a pre-right image of ρ if there is i $(0 \leq i \leq n)$ such that $\varepsilon_i \hat{} \varepsilon_{i+1} \hat{} \cdots \hat{} \varepsilon_n = \rho_0 \hat{} \rho_1 \hat{} \cdots \hat{} \rho_m$ where each ρ_j $(0 \leq j \leq m)$ is a trail of ref(v_j). It is clear that if there is a path in G which is a pre-right image of ρ, then a scenario of G occurs in ρ, and we call such a path by right image of ρ. We can develop an algorithm to check if there is a right image of ρ in G, which is depicted in Fig. 13. The structure of the algorithm is the same as the one of the algorithm for checking left images depicted in Fig. 12. The difference from the algorithm for checking left images is that the depth first search is reversed and starts from the end node $\perp$. Clearly, this algorithm can support the existential consistency checking. This algorithm can also be used in the backward mandatory consistency checking, which will be described in Section 4.2.3.

4.3.2 Forward mandatory consistency checking: Let $\mathcal{S}_{\mathrm{F}}(D, G)$ be a forward mandatory consistency specification, which consists of a sequence diagrams D and an IOD G. It requires that if a reference scenario described by D occurs during the execution of a program, then a scenario described by G must follow immediately. Thus, for a program execution trace $\rho = \varepsilon_0 \hat{} \varepsilon_1 \hat{} \cdots \hat{} \varepsilon_n$, ρ satisfies $\mathcal{S}_{\mathrm{F}}(D, G)$ if for any subsequence $\varepsilon_i \hat{} \varepsilon_{i+1} \hat{} \cdots \hat{} \varepsilon_j$ $(0 \leq i \leq j < n)$ of ρ which is a trail of D, there is a subsequence $\varepsilon_{j+1} \hat{} \varepsilon_{j+2} \hat{} \cdots \hat{} \varepsilon_k$ $(j < k \leq n)$ of ρ which is a trail of G (i.e. there is a left image of $\varepsilon_{j+1} \hat{} \varepsilon_{j+2} \hat{} \cdots \hat{} \varepsilon_n$ in G).

Based on the algorithm for checking left images in an IOD depicted in Fig. 12, we can check a program execution trace $\rho = \varepsilon_0 \hat{} \varepsilon_1 \hat{} \cdots \hat{} \varepsilon_n$ for a forward mandatory consistency specification $\mathcal{S}_{\mathrm{F}}(D, G)$ as follows: finding out each subsequence $\varepsilon_i \hat{} \varepsilon_{i+1} \hat{} \cdots \hat{} \varepsilon_j$ $(0 \leq i \leq j < n)$ of ρ which is a trail of D, and checking if there is a left image of $\varepsilon_{j+1} \hat{} \varepsilon_{j+2} \hat{} \cdots \hat{} \varepsilon_n$ in G.

4.3.3 Backward mandatory consistency checking: Let $\mathcal{S}_{\mathrm{B}}(D, G)$ be a backward mandatory consistency specification, which consists of a sequence diagram D and an IOD G. It requires that if a reference scenario described by D occurs during the execution of a program, then it must follow immediately from a scenario described by G. Thus, for a program execution trace $\rho = \varepsilon_0 \hat{} \varepsilon_1 \hat{} \cdots \hat{} \varepsilon_n$, ρ satisfies $\mathcal{S}_{\mathrm{B}}(D, G)$ if for any subsequence $\varepsilon_i \hat{} \varepsilon_{i+1} \hat{} \cdots \hat{} \varepsilon_j$ $(0 \leq i \leq j < n)$ of ρ which is a trail of D, there is a subsequence $\varepsilon_k \hat{} \varepsilon_{k+1} \hat{} \cdots \hat{} \varepsilon_{i-1}$ $(0 \leq k < i)$ of ρ which is a trail of G (i.e. there is a right image of $\varepsilon_0 \hat{} \varepsilon_1 \hat{} \cdots \hat{} \varepsilon_{i-1}$ in G).

Based on the algorithm for checking right images in an IOD depicted in Fig. 13, we can check a program execution trace $\rho = \varepsilon_0 \hat{} \varepsilon_1 \hat{} \cdots \hat{} \varepsilon_n$ for a backward mandatory consistency specification $\mathcal{S}_{\mathrm{B}}(D, G)$ as follows: finding out each subsequence $\varepsilon_i \hat{} \varepsilon_{i+1} \hat{} \cdots \hat{} \varepsilon_j$ $(0 \leq i \leq j < n)$ of ρ which is a trail of D, and checking if there is a right image of $\varepsilon_0 \hat{} \varepsilon_1 \hat{} \cdots \hat{} \varepsilon_{i-1}$ in G.

4.3.4 Bidirectional mandatory consistency checking: Let $\mathcal{S}_{\mathrm{D}}(D_1, D_2, G)$ be a bidirectional mandatory consistency specification which consists of two sequence diagrams D_1 and D_2, and an IOD G. It requires that if a reference scenario described by D_1 occurs during the execution of a program and a reference scenario described by D_2 follows, then in between these two scenarios, a scenario described by G must occur exactly. Thus, for a program execution trace $\rho = \varepsilon_0 \hat{} \varepsilon_1 \hat{} \cdots \hat{} \varepsilon_n$, ρ satisfies $\mathcal{S}_{\mathrm{D}}(D_1, D_2, G)$ if for any

```
currentpath := ⟨⊥⟩;
repeat
   node := the last node of currentpath;
   if all preceding node of node are explored through currentpath
   then /*backtracking*/ delete the last node of currentpath
   else begin /*explore an unexplored preceding node through currentpath*/
          node := a preceding node of node not explored through currentpath;
          if node is such that the reversal of the path segment corresponding to
             currentpath is a right image of ρ
          then return true;
          if node is such that the path segment corresponding to currentpath
             is a pre-right image of ρ
          then append node to currentpath;
        end
until currentpath = ⟨⟩;
return false.
```

Fig. 13 *Algorithm for checking right images in an IOD*

subsequence of ρ of the form

$$\varepsilon_i \hat{} \varepsilon_{i+1} \hat{} \cdots \hat{} \varepsilon_j \hat{} \varepsilon_{j+1} \hat{} \varepsilon_{j+2} \hat{} \cdots \hat{} \varepsilon_{k-1} \hat{} \varepsilon_k \hat{} \varepsilon_{k+1} \hat{} \cdots \hat{} \varepsilon_m$$

where

- $0 \leq i < j < k < m \leq n$,
- the subsequence $\varepsilon_i \hat{} \varepsilon_{i+1} \hat{} \cdots \hat{} \varepsilon_j$ is a trail of D_1,
- the subsequence $\varepsilon_k \hat{} \varepsilon_{k+1} \hat{} \cdots \hat{} \varepsilon_m$ is a trail of D_2, and
- any subsequence of the form $\varepsilon_a \hat{} \varepsilon_{a+1} \hat{} \cdots \hat{} \varepsilon_b$ $(j < a < b < k)$ is not any trail of D_1 or D_2,

the subsequence $\varepsilon_{j+1} \hat{} \varepsilon_{j+2} \hat{} \cdots \hat{} \varepsilon_{k-1}$ is a trail of G.

We can check a program execution trace $\rho = \varepsilon_0 \hat{} \varepsilon_1 \hat{} \cdots \hat{} \varepsilon_n$ for a bidirectional mandatory consistency specification $\mathcal{S}_D(D_1, D_2, G)$ as follows: finding out each subsequence of ρ of the form

$$\varepsilon_i \hat{} \varepsilon_{i+1} \hat{} \cdots \hat{} \varepsilon_j \hat{} \varepsilon_{j+1} \hat{} \varepsilon_{j+2} \hat{} \cdots \hat{} \varepsilon_{k-1} \hat{} \varepsilon_k \hat{} \varepsilon_{k+1} \hat{} \cdots \hat{} \varepsilon_m$$

where

- $0 \leq i < j < k < m \leq n$,
- the subsequence $\varepsilon_i \hat{} \varepsilon_{i+1} \hat{} \cdots \hat{} \varepsilon_j$ is a trail of D_1,
- the subsequence $\varepsilon_k \hat{} \varepsilon_{k+1} \hat{} \cdots \hat{} \varepsilon_m$ is a trail of D_2, and
- any subsequence of the form $\varepsilon_a \hat{} \varepsilon_{a+1} \hat{} \cdots \hat{} \varepsilon_b$ $(j < a < b < k)$ is not any trail of D_1 or D_2,

and checking if the subsequence $\varepsilon_{j+1} \hat{} \varepsilon_{j+2} \hat{} \cdots \hat{} \varepsilon_{k-1}$ is a trail of G by the algorithm for checking left images or right images in an IOD.

4.4 Implementation and evaluation

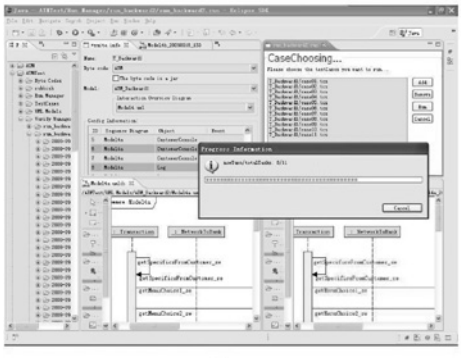
We have implemented a tool prototype UIMDRIVER to support the runtime verification approach presented above, which can be downloaded from http://seg.nju.edu.cn/UIMDRIVER/. It has been used to perform several case studies for evaluating the potential and usability of our runtime verification approach.

4.4.1 Tool prototype: UIMDRIVER is implemented in Java as a plug-in component on the Eclipse platform [27], and its GUI is shown in Fig. 14*a*. It consists of five components as shown in Fig. 14*b*. The UML model editor can help users to create or edit both sequence diagrams and IODs, which is built on the top of the eclipse plug-in component Topcased [28]. The test case manager sets up test configurations, and supports users to manually establish an object mapping from a given scenario-based specification to a program (bytecodes) under verification. Based on this mapping the instrumentor can automatically instrument probe codes into the program (bytecodes) with the help of the Byte Code Engineering Library [29]. The run manager starts up the instrumented program and collects its execution traces. The verifier checks if the collected program execution traces are consistent with the specification on temporal ordering of message interaction.

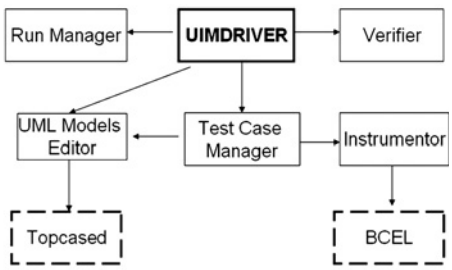
When using UIMDRIVER to check a program (bytecodes) for a scenario-based specification, one should create the models by using the UML model editor. Then with the help of the test case manager, he needs to map the objects in the models to their counterpart in the bytecodes, and set up running configurations of test cases by hand. Once the object mapping and test configurations are ready, the following process is automated: the test case manager invokes the instrumentor to generate the probe codes, the run manager drives the execution of test cases based on their running configurations and collects the execution traces, and the verifier performs the conformance checking.

For a scenario-based specification and a program, once an inconsistent case between them is detected by UIMDRIVER, there are two causes for the case: one is the program bugs resulting from the wrong temporal orders of method calls, the other cause is that the UML models used as the specification are imperfect (incorrect or incomplete) themselves. This implies that UIMDRIVER can also be used to detect imperfect UML interaction models. For example, in reverse engineering of legacy systems, we often need to derive UML models from codes. In those cases UIMDRIVER can be used to check if the derived UML models are imperfect.

The mandatory consistency specifications in our approach have been interpreted in a 'tight' view, which means that if a reference scenario described by the given sequence diagrams occurs during the execution of a program, then it must 'immediately' adhere to a scenario described by the given IOD. In this view, in program execution no other scenario is allowed to occur in between the two scenarios described in the specification. There is also a 'loose' view to interpret the mandatory consistency specifications, which means that if a reference scenario described by the given sequence diagram occurs during the execution of a program, then it must adhere to, 'possibly intermittently', a scenario described by the given IOD. According to this view, in program execution the other scenarios are allowed to occur in between the two scenarios described in the specification. UIMDRIVER implements both the interpretations for the mandatory consistency specifications.



a



b

Fig. 14 *GUI and architecture of tool prototype*

doi: 10.1049/iet-sen.2009.0009

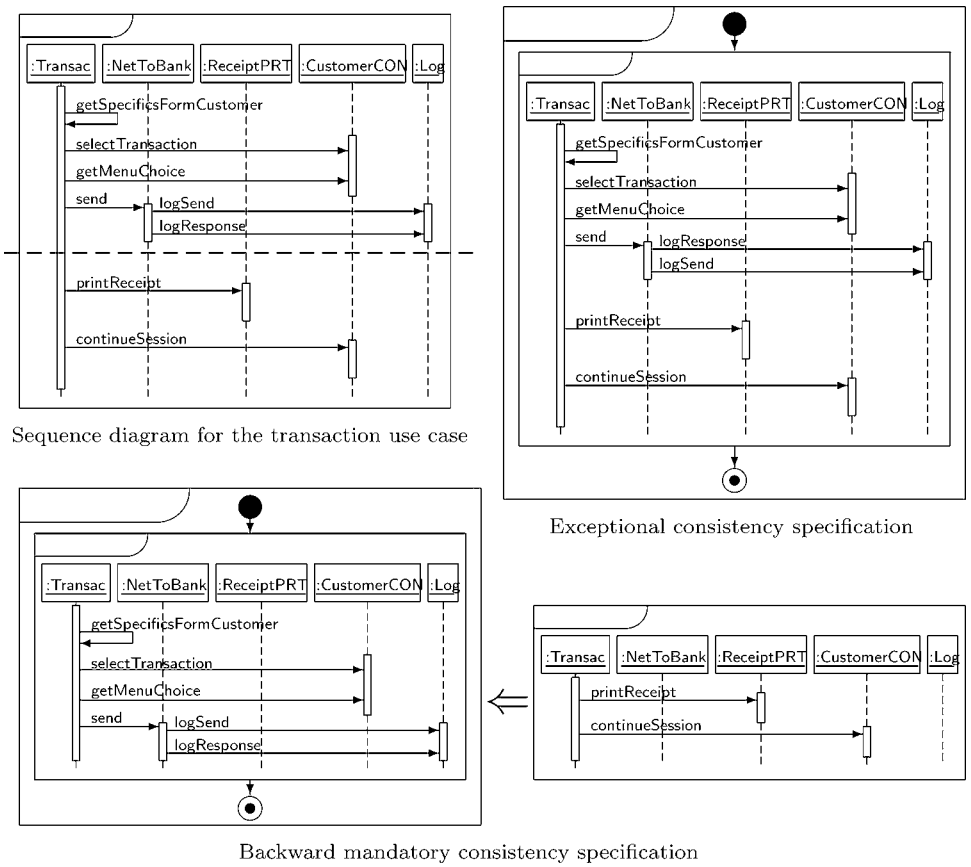


Fig. 15 *Sequence diagrams and specifications in the ATM system*

4.4.2 Case studies: UIMDRIVER has been uesd to perform several case studies for evaluating the potential and usability of our runtime verification approach.

Experiments on an ATM system: One experiment is taken on an automated teller machine simulation system [30], which is a complete example of object-oriented analysis, design and programming applied to a moderate size problem. In [30], the detailed documentations about analysis, design and implementation are given. As depicted in Fig. 15, a sequence diagram derived from transaction use case is selected to construct specifications, which describes the event flow that all the individual types of transaction (withdrawal, deposit, transfer, inquiry) must conform to. It includes two successive scenarios: the first is the business logic of one transaction in which the messages getSpecificsFromCustomer, selectTransaction and getMenuChoice occur in the proper order, which corresponds to ask the customer to select the transaction type and the menu choice; the second is the transaction tracing process which consists of receipt printing and information logging. This sequence diagram is used to construct an exceptional consistency specification and a backward mandatory consistency specification, respectively, which are depicted in Fig. 15. The exceptional consistency specification is constructed by exchanging the positions of message logSend and logResponse in the sequence diagram, which violates the requirement that messages sent to the bank should be logged first and then messages got from the bank are logged, and forms a forbidden scenario in the program. The mandatory consistency specification is constructed by splitting the sequence diagram, which indicates that a transaction cannot enter the tracing process until its corresponding business process has been completed. We conduct the following experiments. First, UIMDRIVER drives the system execution 20 times with random test cases to check if the program execution traces are consistent with the specifications. The experiment is conducted 30 times, and UIMDRIVER reports no inconsistent case. Then, we embed manually a bug in the program by changing the order of the events corresponding to message logSend and logResponse. The experiment conducts 30 times again, and each time UIMDRIVER drives the program execution 20 times with random test cases. Although UIMDRIVER finds out the inconsistent case in each experiment, the first occurrences of the inconsistent case in the experiments are different because of using random test cases. Fig. 16 depicts

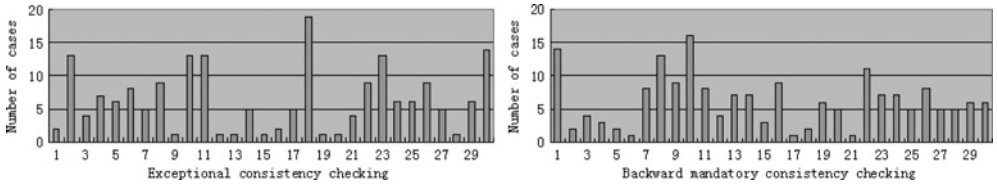


Fig. 16 *Experimental results of the ATM system*

how many times the program has been executed in each experiment when the first occurrence of the inconsistent case is detected.

Experiments on FIPA Iterated Contract Net Iteration Protocol: Another case study is about the FIPA Iterated Contract Net Iteration Protocol (FIPA-ICNIP)[25], which is specified in Section 2.2 by an IOD shown in Fig. 4. We check if the FIPA-ICNIP protocol is implemented soundly by the FIPA-OS v2.2.0 [31] which is an open source system. The first step is splitting the specification in Fig. 4 into a forward mandatory consistency specification, as depicted in Fig. 17. Then using FIPA-OS v2.2.0, we implement several agents that interact with each other for bidding. UIMDRIVER instruments its byte code and generated 5000 tasks for those agents randomly as inputs, and get 5000 execution traces accordingly. The result of consistency checking shows that of the total 5000 execution traces, 259 traces violate the forward mandatory specification. All of them do not reach the end node of the IOD in the specification, and terminate in the sequence diagram propose. After a deeper inspection, we discovered that all these violations are caused by the same deficiency. In FIPA-OS 2.2.0, after receiving the message propose, the Initiator is not allowed to send the message cfp to Participant, which is inconsistent with the FIPA-ICNIP specification [25].

In addition to the above case studies, we also conduct experiments which are derived from a bridge toll station system with 21 classes and 193 methods totally and an official retirement insurance system with 17 classes and 241 methods totally. All the experiments have considerably supported our approach that UML interaction models are used as automatic test oracles to detect the wrong temporal ordering of message interaction in programs.

5 Related work

The runtime verification techniques have been used for Java programs [8, 9, 32–36] to monitor temporal properties and detect program errors such as deadlocks, data races and memory leak. In those works, the specification languages are based on formal notations or event-based programming notations. In general, the advantage of our runtime verification approach is to use UML interaction models to construct simple and expressive scenario-based specifications so as to facilitate the runtime verification application in industry.

Some works on runtime verification of Java programs are based on the idea of Design by Contract [37], and support to represent the design assertions directly in programs which are used for monitoring and verification. Jass [34] allows Java classes to be annotated with specifications based on CSP, which is a pre-compiler and supports assertion monitoring. JML [32] is a notation for specifying the detailed design of Java classes and interfaces using a slight extension of Java's expression syntax, and its tools support runtime debugging of Java code. The literature [8] and [9] extend JML with temporal logic and regular expressions, respectively. In those works, the specifications need to be elaborated based on programs, and it is difficult to reuse design or other specifications directly. Relatively, in our approach, a program under verification could be regarded as a black box, and the specifications in requirements and design could be directly reused.

To our knowledge, there has been few literature on runtime verification of Java programs for scenario-based specifications expressed by UML interaction models which focus on the temporal ordering of message interactions among objects. A scenario-based testing approach is presented in [38] based on a simple subset of live sequence

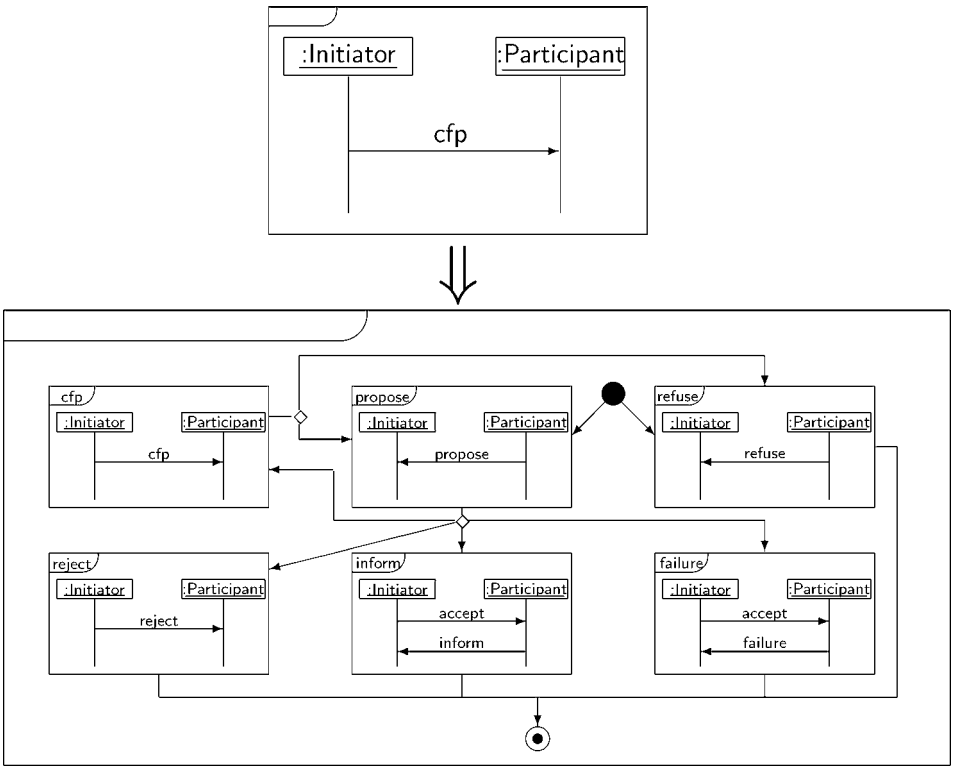


Fig. 17 *Forward mandatory consistency specification for the FIPA Iterated Contract Net Iteration Protocol*

doi: 10.1049/iet-sen.2009.0009

charts [39]. In that work, no implementation technique is given, and the specification language cannot be used to describe the backward and bidirectional mandatory consistency specifications considered in this paper. A preliminary work [21] has been given by us for runtime verification of Java programs for scenario-based specifications. But in that work, we just used simple sequence diagrams to construct the scenario-based specifications, which cannot describe potentially iterating and branching system behaviour, and instrument programs on their source codes. The idea of using UML models as specifications in runtime verification has been extended by us to UML state machine diagrams [40]. In that work, UML state machine diagrams are used as specifications to runtime verification of Java programs for the consistency on object message receiving temporal ordering, and the techniques of program instrumentation and consistency checking are simple. There are also several works [41–43] on verifying Java programs based on model checking techniques [44] whose capacity is restricted by the huge program state spaces.

The field of runtime verification overlaps with the field of testing from the perspective of test oracles. Our runtime verification approach does the consistency checking off-line, which essentially leads to a supporting tool for testing in which the scenario-based specifications are used as automatic test oracles. There are a number of works on UML model-based testing [45–49] whose intentions are different from the one of our work, which are focused on deriving test cases and constraints from UML models. Those work could be integrated in our approach for generating test cases to drive programs under verification.

The existing works on runtime verification have typically focused on program monitoring, which interleaves the analysis and recording program information with program execution [50]. For this kind of online analysis, it is necessary to improving the consistency checking algorithms in our approach for producing the analysis result faster. It is also promising to establish our approach on multi-core platforms for achieving better performance [51].

6 Conclusion

In this paper, we use UML2.0 IODs and sequence diagrams to construct simple and expressive scenario-based specifications, and give an approach to runtime verification of Java programs. This approach leads to a supporting tool for testing in which UML interaction models are used as automatic test oracles to detect the wrong temporal ordering of message interaction in programs.

Since UML has become widely accepted as a modelling standard for object-oriented software development, our work could facilitate runtime verification application in industry. In this paper, our work focuses on the runtime verification of Java programs, but the underlying approach and ideas are more general and may also be applied to the runtime verification of the other object-oriented programs.

7 Acknowledgement

Thanks to the anonymous reviewers for their valuable comments and suggestions. This work is supported by the National Natural Science Foundation of China (No. 90818022, No. 60721002), the National 863 High-Tech Programme of China (No. 2009AA01Z148, No. 2007AA010302), the National Grand Fundamental Research 973 Program of China (No. 2009CB320702), and by the National S&T Major Project (2009z01036-001-001-3).

8 References

1 Havelund, K., Rosu, G. (Eds.): Proc. First Workshop on Runtime Verification, Electronic Notes in Theoretical Computer Science, 2001, vol. 55, Issue 2
2 Runtime Verification: http://www.runtime-verification.org/
3 Wikipedia encyclopedia. 'Runtime Verification'. http://en.wikipedia.org/wiki/Runtime_verification
4 Finkbeiner, B., Sankaranarayanan, S., Sipma, H.: 'Collecting statistics over runtime executions', *Electron. Notes Theoret. Comput. Sci.*, 2002, **70**, (4), pp. 36–55
5 ITU-T: 'Recommendation Z.120. ITU – telecommunication standardization sector'. Geneva, Switzerland, May 1996
6 Rumbaugh, J., Jacobson, I., Booch, G.: 'The unified modeling language reference manual' (Addison-Wesley, 1999)
7 OMG. 'UML2.0 superstructure specification'. http://www.uml.org, October 2005
8 Trentelman, K., Huisman, M.: 'Extending JML specifications with temporal logic'. Proc. Ninth Int. Conf. on Algebraic Methodology and Software Technology (AMAST2002), 2002, (*LNCS,* **2422**), pp. 334–348
9 Cheon, Y., Perumandla, A.: 'Specifying and checking method call sequences of Java programs', *Softw. Qual. J.*, 2007, **15**, pp. 7–25
10 Drusinsky, D.: 'Semantics and runtime monitoring of TLCharts: statechart automata with temporal logic conditioned transitions', *Electron. Notes Theoret. Comput. Sci.*, 2005, **113**, pp. 3–21
11 Dobing, B., Parsons, J.: 'How UML is used?', *Commun. ACM*, 2006, **49**, (5), pp. 109–113
12 Kluge, O.: 'Modelling a railway crossing with message sequence chatrs and Petri Nets', in Ehrig, H. (Ed.): 'Petri technology for communication-based systems – advance in Petri Nets', 2003, (*LNCS*, **2472**), pp. 197–218
13 Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G.: 'Comparing different approaches for specifying and verifying real-time systems'. Proc. 10th IEEE Workshop on Real-Time Operating Systems and Software, New York, 1993, pp. 122–129
14 Peled, D.A.: 'Software reliability methods' (Springer, 2001)
15 Alur, R., Holzmann, G.J., Peled, D.: 'An analyzer for message sequence charts', *Softw. – Concepts Tools*, 1996, **17**, pp. 70–77
16 Ben-Abdallah, H., Leue, S.: 'Timing constraints in message sequence chart specifications'. Proc. FORTE/PSTV'97, 1997
17 Seemann, J., Gudenberg, J.W.: 'Extension of UML sequence diagrams for real-time systems'. Proc. Int. UML Workshop, 1998, (*LNCS*, **1618**), pp. 240–252
18 Firley, T., Huhn, M., Diethers, K., Gehrke, T., Goltz, U.: 'Timed sequence diagrams and tool-based analysis – a case study'. Proc. Second Int. Conf. on UML, (UML99), 1999, (*LNCS*, **1732**), pp. 645–660
19 Alur, R., Yannakakis, M.: 'Model checking of message sequence charts'. Proc. 10th Int. Conf. on Concurrency Theory, 1999, (*LNCS*, **1664**), pp. 114–129
20 Li, X., Lilius, J.: 'Timing analysis of UML sequence diagrams'. UML'99 – The Unified Modeling Language, 1999, (*LNCS*, **1723**), pp. 661–674
21 Li, X., Wang, L., Qiu, X., Lei, B., *et al.*: 'Runtime verification of Java programs for scenario-based specifications'. Proc. 11th Int. Conf. on Reliable Software Technologies, (Ada-Europe'2006), 2006, (*LNCS*, **4006**), pp. 94–106
22 Lund, M.S., Stolen, K.: 'A fully general operational semantics for UML 2.0 sequence diagrams with potential and mandatory choice'. Proc. 14th Int. Symp. on Formal Methods (FM2006), 2006, (*LNCS*, **4085**), pp. 380–395
23 Haugen, O., Husa, K.E., Runde, R.K., Stolen, K.: 'STAIRS towards formal design with sequence diagrams', *Softw. Syst. Model.*, 2005, **4**, (4), pp. 355–367
24 Cengarle, M.V., Knapp, A.: 'Operational semantics of UML 2.0 interactions'. Technical report TUM-I0505, (Technische Universitat Munchen, 2005
25 Foundation for Intelligent Physical Agents: 'FIPA Iterated Contract Net Iteration Protocol specifications'. http://www.fipa.org/specs/fipa00030/, 2002
26 Lindholm, T., Yellin, F.: 'Java virtual machine specification' (Prentice-Hall PTR, 1999, 2nd edn.)
27 Eclipse – an open development platform. http://www.eclipse.org/
28 Topcased. http://www.topcased.org/
29 The homepage of BCEL. http://jakarta.apache.org/bcel/index.html

30 Bjork, R.C.: 'The simulation of an automated teller machine'. http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample/Links.html
31 Nortel Networks Corporation. 'FIPA-OS distribution notes'. http://fipa-os.sourceforge.net, 2002
32 Leavens, G.T., Leind, K.R.M., Poll, E., Ruby, C., Jacobs, B.: 'JML: notations and tools supporting detailed design in Java'. Addendum to the 2000 Proc. Conf. on Object-oriented Programming, Systems, Languages, and Applications, ACM Press, 2000, pp. 105–106
33 d'Amorim, M., Havelund, K.: 'Event-based runtime verification of Java programs'. Proc. Int. Workshop on Dymaic Analysis, (WOAD2005), 2005, pp. 1–7
34 Bartetzko, D., Fischer, C., Moller, M., Wehrheim, H.: 'Jass – Java with assertions', *Electr. Notes Theoretical Comput. Sci.*, 2001, **55**, (2), pp. 103–117
35 Havelund, K., Rou, G.: 'An overview of runtime verification tool Java PathExplorer', *Formal Meth. Syst. Des.*, 2004, **24**, (2), pp. 189–215
36 Kim, M., Kannan, S., Lee, I., Sokolsky, O., Viswanathan, M.: 'Java-MaC: a run-time assurance tool for Java programs', *Formal Meth. Syst. Des.*, 2004, **24**, (2), pp. 129–155
37 Meyer, B.: 'Applying "design by contract"', *Computer*, 1992, **25**, (10), pp. 40–51
38 Lettrai, M., Klose, J.: 'Scenario-based monitoring and testing of real-time UML models'. Proc. Fourth Int. Conf. on Unified Modeling Language, (UML2001), 2001, (*LNCS*, **2185**), pp. 317–328
39 Damm, W., Harel, D.: 'LSCs: breathing life into message sequence charts', *Formal Meth. Syst. Des.*, 2001, **19**, (1), pp. 45–80
40 Li, X., Qiu, X., Wang, L., Lei, B., Wong, W.E.: 'UML state machine diagram driven runtime verification of Java programs for message interaction consistency'. Proc. 23rd Annual ACM Symp. on Applied Computing, (ACM SAC2008), 2008, pp. 384–389
41 Park, D.Y.W., Stern, U., Skakebak, J.U., Dill, D.L.: 'Java model checking'. Proc. First Int. Workshop on Automated Program Analysis, Testing, and Verification, 2000
42 Holzmann, G.J., Smith, M.H.: 'Software model checking: extracting verification models from source code'. Proc. 12th Int. Conf. on Formal Description Techniques FORTE/PSTV'99, Beijing, China, October 1999
43 Havelund, K., Pressburger, T.: 'Model checking JAVA programs using JAVA PathFinder', *Int. J. Softw. Tools Technol. Transfer*, 2000, **2**, pp. 366–381
44 Clarke, E.M., Grumberg, O., Peled, D.A.: 'Model checking' (The MIT Press, 1999)
45 Offutt, J., Abdurazik, A.: 'Generating tests from UML specifications'. Proc. Second Int. Conf. on Unified Modeling Language, (UML1999), 1999, (*LNCS*, **1723**), pp. 416–429
46 Chevalley, P., Thevenod-Fosse, P.: 'Automated generation of statistical test cases from UML state diagrams'. Proc. Int. Computer Software and Applications Conf., 2001, pp. 205–214
47 Kim, Y.G., Hong, H.S., Cho, S.M., Bae, D.H., Cha, S.D.: 'Test case generation from UML state diagrams', *IEEE Proc. Softw.*, 1999, **146**, (4), pp. 187–192
48 Abdurazik, A., Offutt, J.: 'Using UML collaboration diagrams for static checking and test generation'. Proc. Third Int. Conf. on Unified Modeling Language, (UML2000), 2000, (*LNCS*, **1939**), pp. 383–395
49 Ali, S., Jaffar-ur Rehman, M., Briand, L.C., Ashar, H., Zafar, Z., Nadeem, A.: 'A state-based approach to integration testing for object-oriented programs'. Technical report SCE-05–08, Department of Systems and Computer Engineering, Carleton University, Canada, 2005
50 Dwyer, M.B., Kinneer, A., Elbaum, S.: 'Adaptive online program analysis'. Proc. Int. Conf. on Software Engineering, (ICSE2007), 2007, pp. 220–229
51 Yang, L., Tang, J., Zhao, J., Li, X.: 'A case study for monitoring-oriented programming in multi-core architecture'. Proc. Int. Workshop on Multicore Software Engineering, (IWMSE2008), Germany, 2008, pp. 47–52

doi: 10.1049/iet-sen.2009.0009