Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 04-034

Program Execution Monitoring with Scenario Implementation Models

Donglin Liang and Kai Xu

September 29, 2004

Program Execution Monitoring with Scenario Implementation Models

Donglin Liang and Kai Xu
Univeristy of Minnesota
Minneapolis, MN 55455, {dliang,kai}@cs.umn.edu

Abstract

Scenarios have become an important concept for understanding and specifying the required behaviors for a system and its components. To gain confidence for the correctness of the implementation, software developers must compare the scenarios identified during requirements analysis and software design with the behaviors observed during execution. This paper presents an execution monitoring framework that facilitates such a comparison. The framework allows the software developers to specify, in scenario implementation models, how the scenarios are expected to be implemented and used in the program. Enhanced with statements that check properties of the scenarios or visualize the progress of the scenarios, these models can be used to check at runtime whether the scenarios have been implemented and used correctly. Our initial case study indicates that the framework can be useful for uncovering bugs.

1 Introduction

"The presence of bugs in programs can be regarded as a fundamental phenomenon." [15]. This claim is supported by a recent report published by the National Institute of Standards and Technology. The report estimated that software bugs cost American industries up to $60 billion annually [13]. This result suggests that improved testing and debugging techniques are urgently needed to facilitate effective detection, localization, and removal of the bugs.

To detect and to localize software bugs[1], it is important to be able to observe and to reason about the dynamic behaviors of a software system and its components during the execution. It is also important to compare such behaviors with those that have been identified and specified during requirements solicitation, analysis, and software design. Such comparison allows the software developers to check whether the system and its components behave as intended. Many execution monitoring techniques have been developed to facilitate the observation of and the reasoning about the dynamic behaviors of software execution. Although they may be useful for understanding the run-time behaviors of a system, there is a conceptual *gap* between the notations used in these techniques and the notations used for specifying the system behaviors at the requirement and design levels. The gap prevents an effective comparison between the program behaviors observed and those specified. Therefore, using these techniques, detecting and localizing bugs in a system are still laborious and ineffective.

Modern software development methodologies (e.g., [8]) encourage the identification of *scenarios*, each of which represents a sequence of actions in which a system or its components are involved to accomplish a specific task.[2] These scenarios can be modelled with various notations during requirements solicitation, analysis, and software design [5]. A scenario presents a focused view of the system's behaviors. Identifying and modelling these views allow software developers to employ a divide-and-conquer strategy for specifying and reasoning about these behaviors. Such a strategy is crucial for understanding complex systems.

Software developers attempt to ensure that the scenarios identified in the requirements and design phases are correctly implemented in a program. This task is complicated by a number of factors. In a program, the actions of a scenario are often implemented with statements that are scattered at various locations in several modules. The control-flow that determines the execution order of these statements is often tangled with those that implement other scenarios. The data that are operated by the actions often become obscured in the presence of aliasing and heap-allocated objects. To deal with these complications effectively, an execution monitoring technique must enable the software developers to monitor the progress of a scenario and to verify that the actions of the scenario occur in the right order, on the right data, and under the right execution contexts.

Existing execution monitoring techniques provide inadequate support for specifying how the progress of the scenarios should be monitored for verifying the correctness of

[1] The bugs mentioned here include implementation and design flaws.

[2] The scenario defined here extends the notion from requirements level to design and implementation levels. The notion includes those identified in use cases, message sequence diagrams, state charts, etc.

the implementation of the scenarios. Many techniques used in practice, such as logging, assertions, and breakpoints, focus on monitoring the program execution at individual program points. However, because of the lack of mechanisms for correlating the actions performed at different program points, using these techniques for monitoring the progress of a scenario may involve intensive manual efforts. Some more advanced techniques (e.g., [2, 12]) can detect the occurrences of specific sequences of actions through mechanisms that define patterns of events to be matched in the execution history. Some other advanced techniques (e.g., [3, 4, 7, 10]) can check temporal logic properties specified over sequences of actions performed by the system. However, these advanced techniques provide little support for monitoring the individual actions that occur during the progress of a specific sequence. Such monitoring is crucial for verifying that each action of a scenario occurs on the right data and under the right execution context.

This paper presents an execution monitoring framework for verifying the scenario implementations. The framework provides notations that allow the software developers to specify, in *scenario implementation models*, how the scenarios of interest are expected to be implemented in a program. A scenario implementation model captures, for a scenario being modelled, both the sequencing order of the actions and the runtime circumstances under which the actions should occur during program execution. The model may also include *monitoring* statements that inspect the program states when an action occurs. These statements may access the runtime program entities, perform computation to check properties, and output debugging information to visualize the progress of the scenarios. The framework also extends the host language and the runtime environment so that the scenario implementation models will be used for monitoring the execution of the scenarios. The extension to the host language allows the specification of when a scenario should be observed during program execution. The extension to the runtime environment can recognize, based on the models, the occurrences of the actions for the scenarios. Once the occurance of an action is detected, the extension will execute the corresponding monitoring statements.

The paper also presents a case study that investigates the use of the proposed framework in detecting and localizing non-trivial bugs in a realistic software system. Using our framework, the software developers can develop scenario implementation models for the scenarios of interest. These models can be enhanced with monitoring statements for checking various properties that encode important design decisions or implementation assumptions for the scenarios. Our case study suggests that checking these properties is useful for uncovering software bugs.

A major benefit of using our framework is that it can improve the effectiveness of testing and debugging. Scenarios and their properties identified in a pre-implementation phase can be conveniently specified with scenario implementation models. Such models can be used in the place of an oracle during testing to check whether the program behaves correctly. This enables software developers to systematically detect the faults existing in the implementation of the scenarios. In addition, because a scenario presents a focused view of the system's behaviors, relating bugs to the implementation of a specific scenario could help software developers to locate and to fix the software bugs faster.

Scenarios have been viewed as a central notion for understanding and reasoning about software behaviors. The *significance* of the proposed technology is that it enables a new execution monitoring paradigm that extends the use of scenarios from pre-implementation phases to post-implementation phases. This extension is critical for completing a promising step in the further improvement of software development methodologies: the use of scenarios to model and to reason about software behaviors throughout the entire software development process. This *scenario-driven* approach complements the object-oriented approach. Together, they should allow developers to build more reliable software in shorter time.

2 Execution Monitoring with Scenario Implementation Models

This section presents the basic ideas of the use of scenario models for monitoring the execution of programs.

2.1 An Overview of The Scenario Implementation Model

A scenario implementation model captures both the sequencing among the actions of the modelled scenarios and the information that characterizes the runtime circumstances under which these actions occur. Runtime circumstances can be identified with execution events. An execution event represents a point of time that can be characterized with a logical term that conceptually gets evaluated after each program instruction is executed. We refer to such a logical term as an *event characterizing term*. When an event characterizing term evaluates true during the execution, we say that an event identified by this term *occurs*. To refine the set of events that will be identified, an event characterizing term can be conjuncted with *filtering conditions* that will be evaluated together with the characterizing term.

An event identified by a characterizing term and the associated filtering conditions in a scenario implementation model can be interpreted as a witness to the occurrence of an action for a modelled scenario. By associating monitoring statements with this term and the conditions, the execution of the scenario can be monitored: when an event is ob-

served, the program execution will be intercepted, the monitoring statements will be executed to examine the execution context for the action or to output debugging information. We refer to this process as *scenario observation*. This process can check important properties of the scenario, and can also visualize the progress of the scenario.

A scenario implementation model uses a variant of hierarchical state machine [6] for specifying the admissible sequences of actions for the scenarios. Hierarchical state machines have been adopted in UML to model scenarios at the design level. Some other scenario models (e.g., message sequencing charts [14]), can also be easily translated into such a formalism [9, 17]. A state machine used in a scenario implementation model has a set of discrete states. Each state may have a set of transitions that will take the state machine from this state to the next state. A transition is triggered by a specific kind of event that is identified by an event characterizing term. The transition typically involves three steps: processing the triggering event, leaving the current state, and entering the next state. The filtering conditions can be evaluated at the first step. If these conditions are evaluated to true, then the rest of the first step, the second and the third steps will be carried out. The monitoring statements for the event can be performed at these steps. A state machine can also have state variables. These variables can record the program information extracted during the progress of the scenario. These variables can also hold the results computed by the monitoring statements. In a sense, these variables and the discrete states of the state machine provide a view to the states of the scenario execution. This view is important for verifying that the actions of the modelled scenarios are performed as intended.

A scenario implementation model may use multiple state machines to specify the admissible sequences of actions for the modelled scenarios. These state machines can be organized in a hierarchical way. One state machine M_a can simplify its structure by employing another state machine M_b. In this case, M_b monitors a specific set of subsequences of actions for the modelled scenarios. M_a will then observe the transitions of M_b to detect and to respond to the occurrences of the subsequences that it is interested in. In this way, M_a may use a few states to monitor rather complicated scenarios. This approach also allows the software developers to specify the monitoring tasks in a modular way: detailed monitoring statements for individual actions will be specified in M_b whereas high-level monitoring statements for the subsequences of actions will be specified in M_a.

There are two points of time at which M_a can observe and respond to a transition of M_b: right before leaving the current state and right after entering the next state. These points of time can also be represented by events that can be identified with event characterizing terms. For example, the characterizing term "entering S_1 in M_b" identifies events for the point of time right after M_b enters state S_1. We refer to such events as *transition events*.

2.2 Controlling The Scenario Observation

The occurrence of a scenario is sensitive to the program contexts. Therefore, the state machines for a scenario implementation model should be created only when program execution enters a context in which a modelled scenario is expected to occur; and the state machines should be destroyed when program execution leaves this context. Even within the context, there may be some periods of time (or sub-contexts) in which none of the actions for the scenario are expected to occur. In this case, on entering and leaving of such a sub-context, the state machines will be disabled and re-enabled accordingly.

The life-time activities of the state machines can be controlled manually through debugging commands at breakpoints set at various locations in the program context. These activities can also be controlled automatically by the program context through state machine managers. A state machine manager is like an object. It provides a set of operations that can be invoked by the program context or a state machine for creating, enabling, disabling, and destroying the state machines that it manages.

2.3 The Specification and the Use of Scenario Implementation Models

To use scenario implementation models for execution monitoring, the software developers must provide a specification that describes both the state machines and how the state machines should be used. We use a *scenario observation specification* language (SOS) for writing such specifications. SOS can be viewed as an extension to the host language in which the target program is written. The extension will rely on existing notations in the language, and will introduce new constructs only when it is necessary.

In SOS, state machines will be specified using state machine models (see Figure 1 (b)). A state machine model can declare states, state variables, and methods that operate on the state variables. Each state may contain a set of *guarded command statements* that specify the transitions for this state. A guarded command statement (a statement starting with the keyword `on` in Figure 1(b)) consists of a *guarding expression* that specifies an event characterizing term and a *response block* that specifies the response statements for this term. The guarding expression provides an *observation interface* that consists of a set of variables that allow the response statements to access run-time entities. Some of methods declared in a model are constructors, one of which will be called when a state machine is created using the model. Like an object constructor, a state machine constructor can accept parameters so that state machines can be

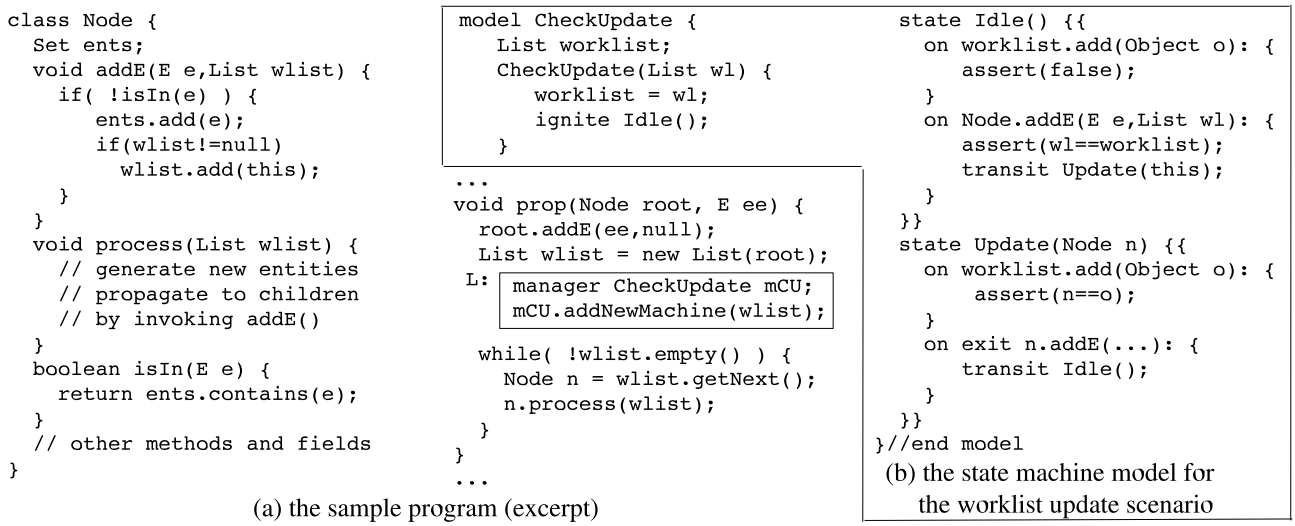


Figure 1. Execution monitoring with a scenario implementation model.

created from the model for monitoring scenarios related to different objects. The constructor will also instruct the state machine to move to an initial state.

State machine models will be used by state machine mangers to create state machines. A state machine manager can be declared in a program context as a variable. Within the scope where the manager variable is visible, operations can be invoked by the program context on this variable to create state machines and to manage the state machines. After a state machine is created and enabled, it can observe and respond to events until it is destroyed or disabled.

Figure 1 shows an example that illustrates how scenario implementation models can be used for checking scenario properties. The Java program excerpt in Figure 1 (a) sketches an implementation of a worklist algorithm that propagates entities of type `E` from the root of a tree to all the nodes in the tree. The nodes are instances of class `Node` and the algorithm is captured in `prop()`. During the progress of the algorithm, when new entities are added to a node, additional entities may be created and propagated to the children of this node (`Node.process()`). When this happens, the children will be added to the worklist so that they will be further processed in the future (`Node.addE()`). Note that `Node.process()` and `Node.addE()` may also be invoked by methods that are not shown here.

Figure 1 (b) shows the model for a state machine that will be used to monitor the progress of the updating scenario for the object representing the worklist within the `while` loop in `prop()`. The model declares two states: `Idle` and `Update`. The model declares a state variable `worklist` that will store a reference to the object representing the worklist. The model also declares a constructor method that initializes the state variable with the formal parameter and moves the state machine into the initial state `Idle` through an *ignite* statement.

`Idle` represents the situation in which the worklist is not ready to be updated. In this state, `add()` should not be invoked on the object referenced through `worklist`. This is specified in the body of `Idle` by the first guarded command statement. In the `Idle` state, when `Node.addE()` is invoked, the state machine will move to the `Update` state through a `transit` statement. When this occurs, the state machine verifies that the worklist is passed in as the second parameter to `Node.addE()`. These are specified in the body of `Idle` by the second guarded command statement. `Update` represents the situation in which the worklist may be updated. If the worklist does get updated, the state machine will check that the right object is added to the worklist (the first guarded command statement in the body of `Update`). On the exit of `Node.add()`, the state machine will move the state back to `Idle` (the second guarded command statement in the body of `Update`).

To use this model for monitoring the worklist updating scenario in the `while` loop in `prop()`, a state machine manager variable will be declared at the location labeled with "`L:`" (see the two statements in the box in Figure 1 (a)). This declaration is followed by an invocation to `addNewMachine()` that creates a state machine from the model. From this point on, the state machine will be able to process the events. Once program execution reaches the end of `prop()`, the state machine manager and the associated state machines will be destroyed. The example shows that the scenario implementation model can capture important information about how this scenario are expected to be implemented. This model can be used to monitor the execution of the actual implementation. Such monitoring may not be easy to specify without a scenario model.

Related work. By weaving advice codes to various locations (joinpoints) in a program, an aspect specified in AOP languages such as AspectJ [1] can monitor the execution of a group of actions by inspecting the program states when the execution reaches these locations. Unfortunately, this mechanism does not directly support the specification of action sequences. In addition, the weaving cannot be controlled by the program contexts. Therefore, this mechanism is not suitable for monitoring the progress of scenarios.

3 The Scenario Observation and the Specification Language

This section first presents the runtime model for the scenario observation. It then briefly discusses the key constructs for a scenario observation specification language.

3.1 The Runtime Model for Scenario Observation

The process of scenario observation involves observing events, intercepting the program execution, and inserting new computations that perform the monitoring tasks. This process is supported by an event subscription mechanism in our framework. Each event characterizing term is associated with an event source that will identify events using this term. The event source for identifying execution events is the runtime environment. The event source for identifying transition events for a state machine is the state machine itself. An entity that intends to process events must subscribe itself to the necessary event sources by associating appropriate event characterizing terms with these sources. During scenario observation, an event source will evaluate its associated event characterizing terms at appropriate points of time during the execution of its actions. If a characterizing term evaluates true at a point of time, then the event source will suspend the execution of its actions and let the subscriber entity that provides this term execute its actions. If multiple characterizing terms evaluate true at the same point of time, then the corresponding subscriber entities for these terms will execute their actions in a predefined sequential order. For example, the execution order may be based on the creation time of the subscriber entities.

The most basic event subscriber entities in our framework are guarded commands. A guarded command is used to process a specific kind of event. It is created from a guarded command statement. During the creation of the guarded command, the guarding expression specified in the guarded command statement will be evaluated. This evaluation produces an event characterizing term and a reference to an event source. The guarded command then associates the term with the event source and subscribes itself to this source. During its creation, a guarded command will also create an *action context* that maintains the information that is needed for the statements in the response block to access runtime entities during event processing.

A set of guarded commands that process events for a specific purpose are managed by an *event processor*. This event processor is responsible for creating, enabling, disabling, and destroying these guarded commands. The event processor may also allocate memory for variables for maintaining the results that are computed by the response blocks of the guarded commands during event processing. We refer to these variables as the *local state variables*.

Our framework uses event processors to represent the components of a state machine (see Figure 2). The framework uses an event processor, the *state activation record*, to represent the current state of a state machine. For each transition τ that may occur at the current state, the state activation record has a corresponding guarded command to process the triggering events for τ. During the event processing, if the condition for the transition becomes true, the guarded command will issue a *state change request*—a special kind of event. This request will be processed by another event processor, the *state controller* of the state machine, to move this machine to the next state. The state activation record may also contain guarded commands that process the events related to checking specific properties.

A state controller in a state machine manages the state activation records for the state machine. It has a guarded command for handling the state change request issued by the current state activation record. Once the state controller intercepts the request, it will destroy the current activation record, and will create a new activation record for the next state that is identified by the parameter of the state change request. Because the state controller is in charge of changing the states for the state machine, this controller will act as the event source for the transition events of this machine.

In some state machines, a transition may be triggered by events characterized by a compound condition that cannot be directly expressed using an event characterizing term. For example, a transition in a state machine may be triggered by the entry of method m when m is called after method m_1 returns with "0". We refer to this kind of event as the *composite event*. Our framework uses a special kind of event processor, the *composite event recognizer*, to recognize the composite events. A composite event recognizer uses guarded commands to process the events required for evaluating the compound conditions. When a guarded command detects that the condition for the composite event is true, it terminates the response action with a special "event-raise" command. When the event-raise command is executed, any guarded command that subscribes for the composite event will be activated.

As mentioned in the previous section, a state machine A can simplify its structure by employing another state ma-

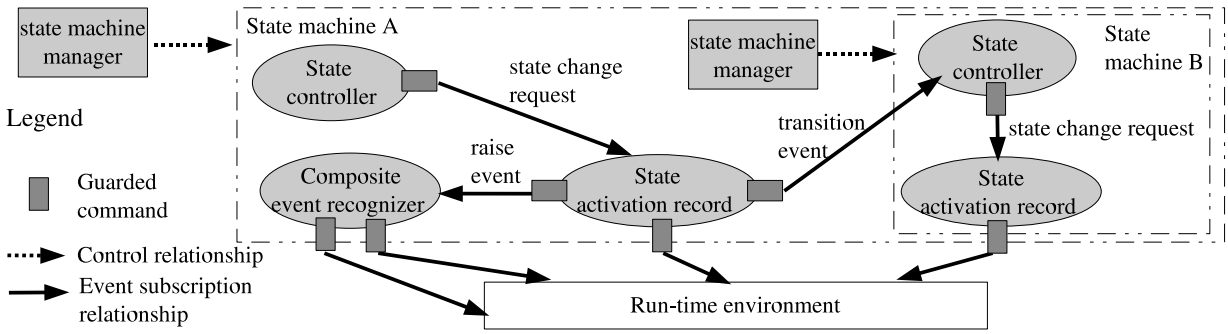


Figure 2. The representation of state machines.

chine B. In this case, A will maintain the state machine manager for B; and the current state activation record of A will have guarded commands that subscribe to the transition events of B. Figure 2 conceptually illustrates the components of the two state machines and the relationships among these components.

3.2 A Scenario Observation Specification Language for Java

One important design objective for a scenario observation specification language is the seamless integration with the host language. To achieve this, we introduce only a few new constructs that are essential to the scenario observations. This approach can lower the learning curve for the use of the language. We will present some key ideas behind a scenario observation specification language for Java (Java-SOS). These ideas can be applied when designing the specification language for other object-oriented languages. The syntax productions for the new constructs introduced in Java-SOS are presented in Figure 3. Many of these constructs have been used in the example in Figure 1 (b).

The state machine models. As shown in Figure 1 (b), the state machines used for monitoring scenario execution are specified with state machine models in Java-SOS. A state machine model declaration (production $P1$ in Figure 3) contains a list of declarations for state variables, methods, states, and composite events. The state variables and the methods are similar to the fields and the methods in a regular class. As shown in Figure 1 (b), a constructor method can use an ignite statement (production $P10$) to transfer the created state machine into an initial state. Note that, an ignite statement is similar to a return statement. It will terminate the control flow. A constructor may contain multiple ignite statements so that the created state machines may be initialized with different initial states based on the parameters to the constructor.

A state declaration (production $P2$) or a composite event declaration (production $P3$) define the event processors that

P1: <state machine model declaration> :=
    model <model id> {<declaration list>}
P2: <state declaration> :=
    state <state id>(<parameter list>) <bootstrap block>
P3: <composite event declaration> :=
    event <event type id>[<parameter list>](<variable list>)
        <bootstrap block>
P4: <bootstrap block> :=
    {{<variable declaration and statement list>}}
P5: <guarded command statement> :=
    on <guarding expression>**:** <Java block>
P6: <guarding expression> :=
    <event type id>[<expression list>](<variable list>)
P7: <transit statement> := **transit** <state id>(<expression list>)
P8: <on leave statement> := **onleave** <Java block>
P9: <event raise statement> := **raise** (<expression list>)
P10: <ignite statement> := **ignite** <state id>(<expression list>)
P11: <state machine manager declaration> :=
    manager <model id> <manager name> |
    import manager <model id> <manager name>

Figure 3. The syntax productions for key constructs in Java-SOS.

will be used to represent the state activation record for a state or a composite event recognizer for a specific type of composite event. Both kinds of declarations may declare formal parameters that can be used to configure the creation of the event processors. In both kinds of declarations, the body is defined with a *bootstrap block*. A bootstrap block (production $P4$) extends a regular Java block to allow guarded command statements to appear within the block. To highlight such an extension, a bootstrap block is enclosed by double curly brackets. A bootstrop block is executable. When the block is executed, it will create and configure an event processor.

A bootstrap block can declare local variables and local state variables. A local variable exists only during the execution of the bootstrap block whereas a local state variable exists throughout the entire life-time of the created event

Type	*Formats of Guarding Expressions*
break point	`breakpoint`[⟨class⟩,⟨method⟩,⟨line⟩](⟨ENV⟩†)
method entry	`enter`[⟨class⟩,⟨method⟩](⟨ENV⟩)
method exit	`exit`[⟨class⟩,⟨method⟩](⟨ENV⟩)
field read	`read`[⟨class⟩,⟨field⟩](⟨ENV⟩) `read`[⟨variable⟩,⟨field⟩(⟨ENV⟩)
field write	`write`[⟨class⟩,⟨field⟩](⟨ENV⟩) `write`[⟨variable⟩,⟨field⟩](⟨ENV⟩)
exception throw	`throw`[⟨exception class⟩](⟨ENV⟩)
state entry	`enter`[⟨model id⟩,⟨state id⟩](⟨ENV⟩)
state exit	`exit`[⟨model id⟩,⟨state id⟩](⟨ENV⟩)

†⟨ENV⟩ is a variable refering to an environment object that allows the response block of a guarded command statement to access to the various runtime entities and to reflect on the program syntax information.

Table 1. Identifying various kinds of events.

processor. To distinguish these two different kinds of variables, the name of a local state variable is preceded with character "$" in its declaration (it can be understood as being pinned to the event processor).

The statements in a bootstrap block are executed in a sequential order. Before the first statement is executed, an uninitialized event processor will be created. This event processor will allocate memory for the local state variables. When a guarded command statement is encountered during the execution, a guarded command will be created for the event processor. When the execution reaches the end of the bootstrap block, the creation of the event processor is finished. From this point on, the guarded commands of this event processor will be able to process events.

A guarded command statement (production $P5$) is specified with a key word `on` followed by a guarding expression, a colon, and a response block, which is a normal Java block. A guarding expression (production $P6$) is composed with an event type id, a list of actual parameters, and the declaration of a list of observing interface variables.[3] For execution events or transition events, the event type ids are predefined by our framework. For composite events, the event type ids are provided in the declarations of the composite events. An event type id is often associated with a set of formal parameters whose values will be assigned with the corresponding actual parameters specified in the guarding expression. For execution events or transition events, these values will be used to configure the event characterizing term computed during the evaluation of the guarding expression. For composite events, these values will be passed to the formal parameters for the composite event declarations. Table 1 shows an initial set of event type ids and the associated parameters and observing interface variables.

[3]Note that the guarding expressions in Figure 1(b) do not follow this syntax. They are specified with shorthands that will be explained later.

As a part of a guarded command statement gc, the statements in the response block of gc should be able to access the variables that are syntactically visible at the beginning of gc. We refer to these variables as the *environment* variables to gc. However, the response block for gc is not executed when gc is executed. Instead, this block is executed only when a guarded command created from gc is processing an event. Therefore, information about these environment variables must be forwarded from the point of time when gc is being executed to the point of time when the response block is being executed. A data structure called *action context* is created together with the guarded command to forward such information. There are two cases. In the first case, an environment variable is a local variable in the bootstrap block containing gc. This variable will be destroyed on the exit of the bootstrap block and becomes inaccessible during event processing. In this case, the value of this variable will be extracted and stored in the action context. This value will be used when the response block is executed to process events for the guarded command. In the second case, the environment variable is a local state variable in the bootstrap block or a state variable in the state machine model that contains this bootstrap block. In this case, the variable will live at least as long as the created guarded command. Therefore, the address of the memory allocated for this environment variable will be extracted and stored in the action context. This address will be used to access to the variable when the response block is executed to process events for the guarded command.

A guarded command statement in a state declaration can use transit statements (production $P7$) to signal state changes. A guarded command statement in a composite event declaration can use raise statements (production $P9$) to signal the recognition of a composite event. Both transit statements and raise statements will terminate the execution of their containing response block. These statements can also compute a list of values from the actual parameters: for a transit statement, these values will be passed to the formal parameters of the next state; for a raise statement, these values will be passed to the event observation interfaces of the guarded commands subscribing to the event.

The bootstrap block in a state declaration may contain an *on-leave* statement (production $P8$) that specifies the actions to be performed when a state machine leaves the state specified by this declaration. An on-leave statement is treated as a special guarded command statement. This statement must be specified at the end the bootstrap block.

The bootstrap block in a state declaration will be executed when a state machine enters the state defined by this declaration. The entering is caused by the execution of an ignite/transit statement. In this case, the actual parameters computed by the ignite or transit statement will be bound to the formal parameters of the state declaration. The boot-

strap block in a composite event declaration will be executed during the evaluation of a guarding expression that refers to the event type id defined for the composite events. In this case, the actual parameters specified in the guarding expression will be evaluated and bound to the formal parameters of the composite event declaration. The semantics discussed in this paragraph for the execution of a bootstrap block resembles the semantics for procedure invocation.

The state machine manager variables. Java-SOS uses state machine manager variables (or manager variables in short) to define state machine managers. A manager variable can be declared in a way similar to the declaring of a regular variable (production $P11$). Like a regular variable, a state machine manager is created when the program execution reaches the point where a manager variable is declared, and is destroyed when the inner-most containing program entity is destroyed.

When a manager variable is declared as an instance field, this variable is used for monitoring the program execution under the context of the instance that contains this field: the state machines managed by the corresponding manager can observe events only if an activation record for a method of the containing instance is still on the stack. This restriction enforces an *encapsulation* principle: the effect of an instance field should be contained within the context of this instance. When a manager variable is declared as a state variable in a state machine model or a bootstrap block, this manager variable is used for state machine *composition*: the state machines managed by the manager corresponding to this variable will be used to enhance the capability of the state machine or the event processor that encloses the manager. In this case, the state machines managed by the manager will observe events occurred in the program execution that is observed by the enclosing state machine or event processor. To highlight this semantics meaning, Java-SOS requires the manager variable declaration in the second case to be preceded with the key word `import`.

Shorthands for identifying events. Java-SOS provides a set of built-in shorthands for identifying common-used composite events. Some example shorthands have been used as the guarding expressions in Figure 1 (b). The examples show that the syntax for the shorthands can be succinct and self-explained. When a state machine model is being compiled, each shorthand will be translated into a regular guarding expression that subscribes to a built-in composite-event recognizer.

4 A Design for Our Framework

Our design of the framework uses the Java Platform Debugger Architecture (JPDA) [16] to implement the observation of execution events and the debugging capabilities. Our design uses Java objects to represent the components (e.g., event processors, guarded commands) for a state machine. We refer to these objects as *monitoring objects*. Using JPDA, the monitoring objects will be run on a JVM different from the one on which the target program is being run. These objects interact with an event subscription subsystem that extracts execution events from the target JVM through the Java Debugging Interface (JDI). When an event is detected, the subsystem invokes the methods of appropriate monitoring objects to process this event. During event processing, the monitoring objects can access the runtime entities of the target program through JDI.

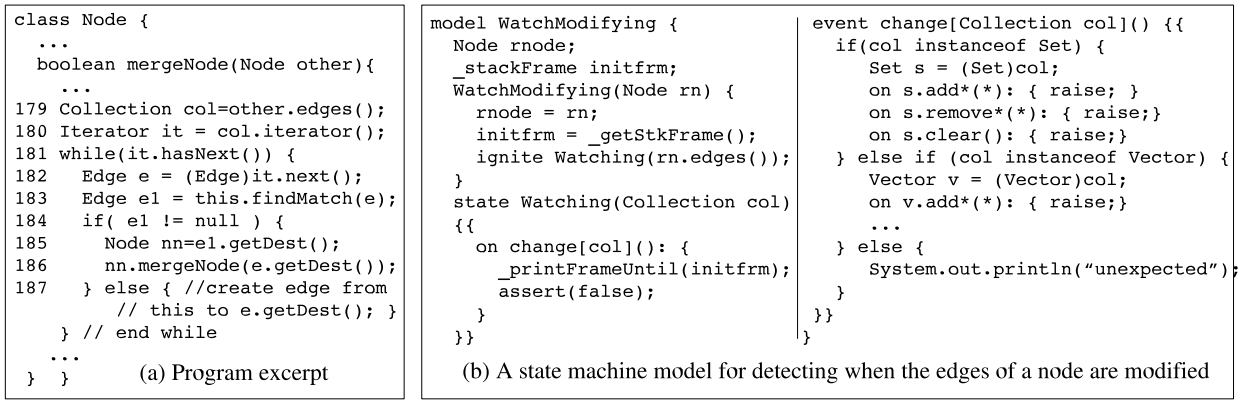
Our framework must translate the specification written in Java-SOS into Java class files. Our framework introduces a preprocessor to handle the special syntax required for specifying monitoring tasks. The preprocessor translates a state machine model into a set of Java classes, and translates the declaration of and the operations on a state machine manager variable into those of a regular reference variable. After the preprocessing, the standard Java compiler is invoked to generate Java class files.

Our framework also provides an interactive interface that allows the software developers to control the program execution and the execution monitoring. Our framework extends the traditional debugging capabilities to allow software developers to control the program execution at the scenario level: the developers can set a breakpoint on a transition or single-step through the transitions of a particular state machine. The framework also allows the software developers to load or to reload state machine model descriptions manually at breakpoints through special commands. The framework then allows to create state machines using the the loaded models, and to control these state machines manually. Such capabilities are important for debugging.

5 A Case Study

This section reports our initial experience of using scenario implementation models for detecting/localizing faults in a Java program. The experience is obtained from a case study that investigates how the proposed framework can be used to solve problems in real debugging situations. The study is demonstrative. Comparative studies must be performed to evaluate its effectiveness in comparison with the traditional debugging techniques.

In this study, we will examine two bugs that we encountered during the implementation of a Java alias analysis system and identify the opportunities in which the proposed framework may be useful. These bugs are related to the construction of points-to graphs [11]. The construction involves about 80 Java classes. The bugs had been detected and localized through code reading and execution with in-



```
class Node {
  ...
  boolean mergeNode(Node other){
    ...
179 Collection col=other.edges();
180 Iterator it = col.iterator();
181 while(it.hasNext()) {
182   Edge e = (Edge)it.next();
183   Edge e1 = this.findMatch(e);
184   if( e1 != null ) {
185     Node nn=e1.getDest();
186     nn.mergeNode(e.getDest());
187   } else { //create edge from
        // this to e.getDest(); }
    } // end while
   ...
 }  }
```

(a) Program excerpt

```
model WatchModifying {
  Node rnode;
  _stackFrame initfrm;
  WatchModifying(Node rn) {
    rnode = rn;
    initfrm = _getStkFrame();
    ignite Watching(rn.edges());
  }
  state Watching(Collection col)
  {{
    on change[col](): {
      _printFrameUntil(initfrm);
      assert(false);
    }
  }}
```

```
event change[Collection col]() {{
  if(col instanceof Set) {
     Set s = (Set)col;
     on s.add*(*): { raise; }
     on s.remove*(*): { raise;}
     on s.clear(): { raise;}
  } else if (col instanceof Vector) {
     Vector v = (Vector)col;
     on v.add*(*): { raise;}
     ...
  } else {
     System.out.println("unexpected");
  }
 }}
}
```

(b) A state machine model for detecting when the edges of a node are modified

Figure 4. Investigating a bug with a scenario implementation model.

serted print statements, and had been fixed in the implementation. Because of the ineffectiveness of this debugging approach, locating the root causes for the bugs had required significant efforts. Note that, because the implementation of our framework is not yet complete, the scenario implementation models used in this study are translated manually into Java classes. This experience actually guides the design and the implementation of our framework.

Bug 1**.** The symptom of this bug is that, when the algorithm is run on a relatively sophisticated subject program, it terminates abnormally with exception "java.util.ConcurrentModificationException" in "Iterator.next()" called at line 182 in file "Node.java". Figure 4(a) shows an excerpt of the source code around this line. The problem is obvious: the edges leaving the node referenced by "other" must have been changed in the previous iteration of the loop. The real challenge is to find out what statements change these edges and why. This task is difficult to perform with traditional debugging techniques because (1) each iteration of the loop may involve many method calls, some of which are recursive calls, and (2) "mergeNode()" will be invoked many times on the subject program.

Figure 4(b) shows a state machine model that can be used to determine when the edges of a node may be changed. In the model, when a method is invoked to change the content of the vector or the set that maintains the outgoing edges for a node, the stack frames will be printed.[4] To use this model, we declare a state machine manager variable between lines 179 and 180, and create a state machine for the instance referenced by "other". When the program was rerun, one state machine reported that "Set.add()" is invoked within the "else" block starting at line 187. This occurs when "mergeNode()" is recursively called at line 186 within the invocation of "mergeNode()" in which the state machine is created. By tracing the propagation of the value for "other" among the methods, we discovered that the problem is caused by an edge connecting the node referenced by "this" to the node referenced by "other" in the invocation. In this case, new edges may be added to the node referenced by "other" when "mergeNode()" is recursively invoked at line 186. The problem had been fixed by changing the recursive version of "mergeNode()" into a worklist algorithm.

[4]The shorthands used for identifying the modifying methods are similar to those used in AspectJ [1].

Bug 2**.** This bug appeared in the implementation after the introduction of the worklist version of "mergeNode()". The symptom is that, after the alias information is computed, the reference fields of some objects may erroneously point to an empty set of objects. Unlike Bug 1, the cause for this symptom is not obvious. Because the implementation of "mergeNode()" was quite sophisticated, we suspected that its implementation was incorrect.

Method "mergeNode()" is used to put two nodes into an equivalence class. To implement this functionality, when "n1.mergeNode(n2)" is invoked, all information related to "n2", including the incoming and the outgoing edges, will be moved to "n1". The merging will also ensure that the labels on the edges leaving a node are unique. This uniqueness is guaranteed by merging the successors of "n1" and "n2" when necessary. A worklist is introduced to maintain the list of pairs of nodes that need to be merged. This worklist is used in a way similar to the one in Figure 1 (a).

Based on the implementation plan, if two nodes are in the same equivalence class, then after the graph is constructed, the following property should hold: the information related to these two nodes should have been moved to one single node. To check this property, an auxiliary field "forward" has been added to a node to remember where the information of this node has been moved to; an auxiliary method "traceForward()" has been added to trace the "forward" fields and find the node that holds the information for the current node; and a state machine is used to

```
model CheckEquiv {
  List wlist;
  Map node2equiv;
  Vector equivs;
  // equivs keeps all the
  // equivalence classes
  CheckEquiv(List wl) {
    wlist = wl;
    ignite Checking();
  }
  state Checking() {{
   on wlist.add(Object p):
   {
      addPair((Pair)p);
   }
   onleave {
      checkProperty();
   }
  }}
 void addPair(Pair p) {
   // update node2equiv and
   // equivs by considering p.
 }
 void checkProperty() {
  for(int i=0;
      i<equivs.size();i++) {
    Vector v=(Vector)equivs[i];
    Node n0 = ((Node)v[0])
              .traceForward();
    for(int j=1;
        j<v.size();j++) {
      Node n=((Node)v[j])
              .traceForward();
      assert(n==n0);
    } // end for
  } // end for
 }
} //end model
```

Figure 5. The scenario implementation model for investigating Bug 2.

observe the insertion of pairs into the worklist and compute the equivalence classes based on the observation. On leaving the only state of this state machine (this happens when the state machine is destroyed), the property will be checked for the nodes in each equivalence class. Figure 5 shows the model for the state machine. To use this model, we declare a state machine manager variable and create a state machine through the variable right after the worklist is created. When we ran the program, we observed that the property is violated in some cases. We enhanced this model to further investigate when the property is violated. Through the investigation, we determined that the problem occurs when pairs like (n1,n2) and (n2,n3) are presented in the worklist. In this case, when (n2,n3) is processed after (n1,n2), n3 should have been merged with n1, not n2.

Discussion. The case study shows that our framework provides a convenient mechanism for verifying assumptions and localizing problems during debugging. One limitation with this study is that the root causes for the bugs were known beforehand. This is acceptable for a demonstrative study. When the root causes are unknown, we hypothesize that our framework is even more useful because, in such a situation, more assumptions or properties are required to be checked. A tool like our framework that facilitates such checking should improve the effectiveness of debugging. Studies need to be performed to verify our hypothesis.

6 Conclusion

This paper presents the core concepts of a framework that monitors the program execution with scenario implementation models. Our case study shows that the framework may be useful for detecting and localizing software bugs. In our future work, we will complete a prototype implementation for the framework and will perform comparative studies to evaluate the effectiveness of using the framework in testing and debugging. Guided by the experiences and the results of these studies, we will further improve the useability of the framework. We will also integrate our prototype implementation with Eclipse—a widely adopted open source software development platform—and evaluate the proposed technology in real world software production.

References

[1] AspectJ. http://www.eclipse.org/aspectj/.

[2] M. Auguston, C. Jeffery, and S. Underwood. A framework for automatic debugging. Technical Report TR-CS-004/2002, New Mexico State University, 2002.

[3] M. Brorkens and M. Moller. Jassda trace assertions, runtime checking the dynamic of java programs. In *International Conference on Testing of Communicating Systems*, pages 39–48, March 2002.

[4] F. Chen and G. Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Electronic Notes in Theoretical Computer Science*, volume 89, 2003.

[5] O. M. Group. OMG unified modeling language specification. www.omg.org, 2001.

[6] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[7] K. Havelund and G. Rosu. Monitoring java programs with java pathexplorer. In *Proceedings of the First Workshop on Runtime Verification*, 2001.

[8] I. Jacobson, G.Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[9] I. Khriss, M. Elkoutbi, and R. Keller. Automating the synthesis of uml statechart diagrams from multiple collaboration diagrams. In *UML'98: Beyond the Notation*, pages 132–147, 1998.

[10] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: A run-time assurance approach for java programs. *Formal Methods in System Design*, 24(2), 2004.

[11] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for java. In *PASTE'01*, pages 73–79, June 2001.

[12] R. A. Olsson, R. H. Cawford, and W. W. Ho. A dataflow approach to event-based debugging. *Software - Practice and Experience*, 21(2):209–230, 1991.

[13] RTI. The echonomic impacts of inadequate infrastructure for software testing. NIST Planning report 02-3, 2002.

[14] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.

[15] J. T. Schwartz. *Debugging Techniques in Large Systems*, chapter An Overview of Bugs, pages 1–16. Prentice-Hall, 1971.

[16] JPDA. http://java.sun.com/products/jpda/.

[17] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proceedings of the 22nd international conference on Software engineering*, pages 314–323, 2000.