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 05-013

Debugging Object-Oriented Programs with Behavior Views

Donglin Liang and Kai Xu

April 06, 2005

Debugging Object-Oriented Programs with Behavior Views

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

Abstract

A complex software system may perform many program tasks during execution to provide the required functionalities. To detect and localize bugs related to the implementation of these tasks, the software developers must be able to monitor the progress of the tasks during execution and check whether the actions for these tasks have been performed correctly. This paper presents a debugger to facilitate this monitoring. The debugger introduces a new kind of abstraction, the behavior views, that can be used to specify how the actions for a program task are expected to occur in various scenarios. Enhanced with statements that can check properties at various steps during the progress of the task, a behavior view can be used to monitor whether the actions for the tasks have been performed at the right time, on the right set of data, and with the right effects on the program states. Our initial case study indicates that the debugger can be useful for localizing bugs.

1. INTRODUCTION

"The presence of bugs in programs can be regarded as a fundamental phenomenon." [14] This claim is supported by a recent report [13] published by the National Institute of Standards and Technology. The report estimated that bugs in information technology software cost American industries up to $60 billion annually. This suggests that improved testing and debugging techniques are urgently needed to facilitate effective detection, localization, and removal of bugs during software development.

Software development involves gathering requirements for a system, analyzing the requirements, designing the components for the system, and implementing these components. During program implementation, testing and debugging are performed regularly to detect, localize, and remove software bugs. A software system that we develop nowadays has increasingly complex behaviors. The increasing complexity has prompted the need for more effective approaches to perform the software development activities.

To understand the expected behaviors of a system and its components, requirements analysis and design techniques (e.g., [8]) encourage the identification and the specification of the scenarios in which a program task may be performed during program execution. A system or a component is often expected to perform many program tasks during execution to provide the required functionalities. A *scenario* is identified as a sequence of actions that the program will perform under a specific situation for a program task.[1] Focusing on the scenarios during requirements analysis and design would allow the software developers to employ a divide-and-conquer strategy in understanding and communicating the expected behaviors of a complex system.

Bug detection and localization activities should also be organized based on scenarios. Detecting and localizing bugs typically require the software developers to observe and to inspect the dynamic behaviors during the program execution and to check the consistency between the observed behaviors and the expected behaviors of the software. Because the expected behaviors of the software are often identified, specified, and analyzed as scenarios at the requirements and the design levels, the software developers should perform their observation and inspection based on scenarios. This *scenario-driven* debugging approach allows the software developers to effectively use their knowledge of scenarios built during the requirements analysis and design to detect and pinpoint problems in the implementation. Thus, using this approach can improve the effectiveness of debugging.

Although many debugging techniques have been proposed to facilitate debugging activities, these techniques cannot effectively support the scenario-driven debugging approach. Source level debugging mechanisms, such as assertions and breakpoints, allow software developers to inspect the program states when the program control reaches specific code locations. Event-based debugging techniques (e.g., [1, 2, 6, 12]), on the other hand, allow the software developers to specify the inspections to be performed automatically when specific execution events occur. These techniques let the user conveniently observe and monitor the dynamic behaviors during debugging. However, because these techniques do not emphasize on correlating the monitoring of the program actions performed at different points of time during execution, they provide inadequate support for the direct

[1]In literature, a scenario is sometimes only used to refer to a sequence of interactions between a software system and its users. In this paper, we broaden its meaning to include any sequence of program actions that a system would perform to achieve a meaningful goal.

observation and inspection of the progress of the scenarios for a particular task. Such support is essential for the scenario-driven debugging.

This paper presents a scenario-based debugger that we developed to facilitate the observation and the inspection of the progress of scenarios during debugging. Our debugger is based on a new kind of abstraction, the behavior view. A *behavior view* specifies how the actions for the scenarios of a particular task are expected to occur during program execution. The behavior view contains information that characterizes the runtime circumstances under which the actions for these scenarios are expected to occur. It also contains a state machine that models the progress of these scenarios. The behavior view may further contain *monitoring statements* that can be executed in response to the state changes of the state machine. These monitoring statements can check the properties that are expected to be maintained at various steps during the progress of the task, and output information that can be used for problem diagnosis.

Our scenario-based debugger provides a debugging language for specifying the behavior views, and a debugging console for using these views to monitor the progress of the scenarios. Through the console, software developers can create monitors from the behavior views to automatically monitor the progress of and to inspect properties related to the scenarios; software developers can also set breakpoints on or single-step through the transitions of the state machine maintained by a monitor to interactively investigate the progress of the scenarios. In this way, an error in the implementation of the scenarios modelled by a behavior view can be detected and pinpointed.

One benefit of using our debugger is that it allows the software developers to modularize, within a behavior view, their expectations for the scenarios of a specific program task. Such expectations can then be checked automatically against the program execution. This feature is especially useful when investigating multiple program tasks simultaneously within a debugging session. Another benefit of using our debugger is that it allows the software developers to build, with behavior views, a layer of high-level abstractions for behavior observation during debugging. These abstractions hide the details for interpreting the meanings of individual program statements, and presents a direct interface for observing the progresses of the program tasks. Thus, the software developers can focus on the properties that are directly related to the scenarios of the tasks. Our initial experience shows that these capabilities can be very useful for localizing bugs in Java programs.

The major contribution of this paper is that it presents the core concepts for our scenario-driven debugger and a case study in which two realistic bugs are investigated using the debugger. Section two gives an overview of the scenario-driven debugging approach; Section three discusses the runtime model for execution monitoring with behavior views; Section four discusses the debugging language; Section five presents the case study; Section six discusses the related work; and Section seven concludes and discusses the future work.

2. SCENARIO-DRIVEN DEBUGGING

This section presents the scenario-driven debugging approach and a debugger that supports this approach.

2.1 The Approach

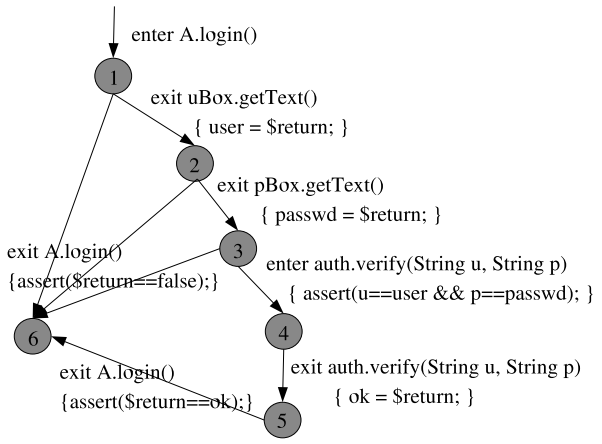
During the debugging process, software developers often need to speculate about the possible causes for a bug, derive hypotheses based on these speculations, and prove or disprove these hypotheses by gathering information from the program source code and the program execution [16]. To form effective hypotheses in this process, software developers must use their knowledge about the expected software behaviors. Because the expected behaviors are often identified, analyzed, and specified as scenarios at the requirements and the design levels, we argue that software developers should use scenarios to guide their debugging processes. We refer to this approach as the *scenario-driven debugging* approach.

In the scenario-driven approach, a software developer first attempts to associate the bug with a particular program task based on the symptom and her understanding of the scenarios for this task. That is, she hypothesizes that the error is in the implementation of the scenarios for performing this task. She then may run some tests on the program and observes the execution of the program actions that are required for the suspected task. If these actions have been executed correctly according to her understanding of the scenarios for the task, then the original hypothesis is probably wrong. In this case, she may need to form a new hypothesis and repeat this process. If some of the actions are executed incorrectly according to her understanding of a particular scenario, she can now focus on this scenario and try to find out the reason for this erroneous behavior. If necessary, she may need to further identify a suspicious subtask and form a new hypothesis. Occasionally, she may need to revisit or to refine a previous hypothesis to inspect a particular task in more details. This process continues until the software developer is able to identify the program elements that account for the erroneous behavior.

The scenario-driven approach seems a natural way for debugging software developed with the modern design methodology. Following this design methodology, software developers would have built an intimate understanding of the expected software behaviors in terms of program tasks and scenarios for these tasks. They may even produce various artifacts (e.g., use cases, UML sequence diagrams) to specify the scenarios. Therefore, forming and verifying hypotheses based on scenarios would allow software developers to effectively use their knowledge of the expected software behaviors to guide the debugging process.

2.2 Debugging with Behavior Views

We introduce behavior views as a new type of abstraction for supporting the scenario-driven debugging. A behavior view specifies how the actions for the scenarios of a particular program task are expected to occur during program execution. A behavior view is built on top of the execution event abstractions. The view uses execution events for detecting the occurrences of the program actions that are expected for a program task, and uses a state machine to specify the expected sequencing among these actions under various scenarios. A transition of the state machine is triggered by an execution event that serves as a witness to the performance of a particular action related to the task. Therefore, the changes of states in the state machine would reflect the progress of the task being monitored. In this sense, the state machine models the progress of the task. By associating monitoring statements with the transitions



Note: "$return" is a special variable representing the value returned by the current method. "user", "passwd", and "ok" are used to pass values from one state to another. "assert(exp)" is a special function that prints out error messages if the "exp" is evaluated false.

Figure 1: A behavior view for a login task.

and the states of the state machine, we may output information to visualize the progress of the task; and we may also verify important properties that must be maintained at various steps when the task progresses.

Figure 1 uses a diagram to show a behavior view that captures the expected scenarios for a login task in a program. In the diagram, nodes represent states and edges represent the transitions for a state machine. Each edge is labeled with the type of execution events that may trigger the represented transition. The edge may also be labeled with the monitoring statements that are executed when the transition is fired. According to the behavior view in Figure 1, the state machine is initialized when the program execution enters method `A.login()`. In a normal scenario, the program gets the user name from a GUI textbox `uBox` by invoking `uBox.getText()`; it then gets the password from another GUI textbox `pBox` by invoking `pBox.getText()`. The program then invokes `verify()` on an authorization object `auth` to verify the user and the password. In other scenarios, `A.login()` may exit without going through all these steps. The monitoring statements on the path 1,2,3,4 ensure that the user name and the password input through the GUI are indeed used correctly when `auth.verify()` is invoked. The monitoring statements on the edges to node 6 ensure that `A.login()` returns `true` only when `auth.verify()` returns `true`. This behavior view can help to determine whether the scenarios for the login task are implemented correctly.

From the above discussions, we can see that a behavior view can be used to specify the monitoring activities that the software developers intend to perform for a specific program task during debugging. A behavior view can capture the important properties for checking the correctness of the implementation for the scenarios of a specific task. Because the progress of the task is explicitly modelled with a state machine, the properties that range over several steps in the task can be easily specified. Therefore, we expect that behavior views would be useful for software developers to quickly prove or disprove their scenario-based hypotheses during debugging.

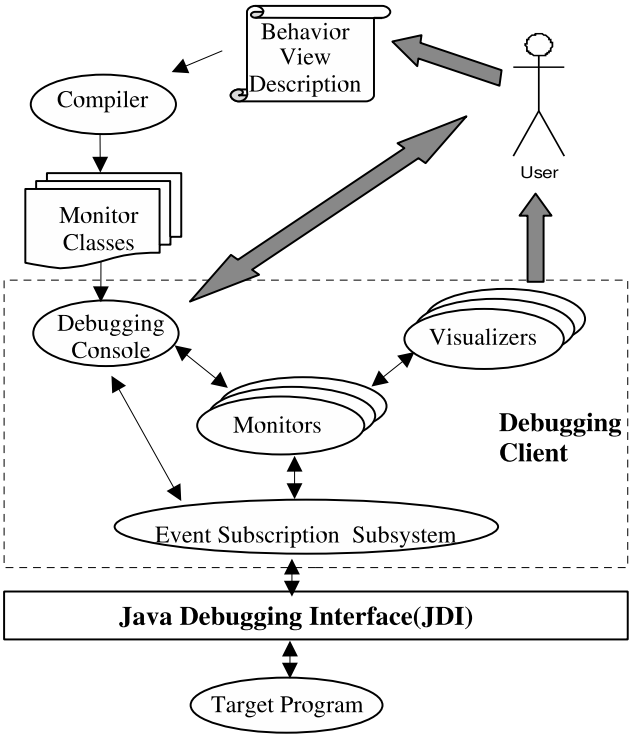


Figure 2: The architecture of the scenario-driven debugger.

Note that a behavior view does not need to specify all the actions that are required for the modelled scenarios. Instead, the view may contain only those actions that are relevant to the properties that are interesting to the software developers. During the debugging process, the software developers may gradually add more actions to be monitored by a behavior view. This flexibility enables the software developers to optimize their debugging strategies.

2.3 A Scenario-Driven Debugger

We developed a scenario-driven debugger that allows the software developers to use the behavior views for monitoring the progress of scenarios in Java programs. Figure 2 shows the architecture of the debugger. The debugger consists of two parts: a behavior view compiler and a debugging client. The compiler translates the description of the behavior views into monitor classes that can be used by the debugging client. The debugging client controls and observes the execution of the target program with the monitors that are created from the monitor classes. The debugging client is built on top of the Java Platform Debugger Architecture (JPDA).[2] Like other debuggers built on JPDA, our debugger uses two Java Virtual Machines (JVMs): one for running the target program, and one for running the debugging client. The debugging client can interact with the target JVM through the Java Debugging Interface (JDI) for monitoring the execution of the target program.

The debugging client consists of four kinds of components: a debugging console, a set of monitors, a set of visualizers,

[2]See http://java.sun.com/products/jpda/index.jsp.

and an event subscription subsystem. The debugging console accepts commands from the user. The console supports source-level debugging commands (e.g., loading a program or setting breakpoints) that are available in many traditional debuggers. The console can also accept commands that are specific for supporting the scenario-driven debugging approach. Through these commands, a user can instruct the debugging client to create monitors from the monitoring classes generated from the description of the behavior views. The user can also set breakpoints on or single-step through the transitions of the state machine maintained by a monitor. Through these commands, the user can debug the program based on the layer of abstractions defined by the behavior views.

A monitor is created for monitoring the progress of the scenarios modelled by a behavior view. Once created, the monitor can observe the execution events that occur during the execution of the target program. When an event occurs, the target program will stop and the monitor will be notified. The monitor can then execute the monitoring statements and fire a transition on the state machine based on the specification of the behavior view. If a breakpoint is set on this transition, the monitor will communicate with the debugging console to solicit user's interactions. In any case, the execution of the target program may be resumed and continue after the current monitoring activity finishes.

A monitor can be associated with a set of visualizers. These visualizers can receive information output by the monitoring statements executed by the monitor and present the information to the user with various visualization techniques.

Both the debugging console and the monitors must interact with the event subscription subsystem to handle the execution events. The event subscription subsystem allows the console and the monitors to subscribe for execution events. The subsystem can then interact with JDI to intercept the execution of the target program at appropriate points of time and to notify the event subscribers accordingly.

3. RUNTIME MODEL FOR MONITORING WITH BEHAVIOR VIEWS

Execution monitoring with a behavior view involves observing events, intercepting the program execution when an event occurs, firing transitions in the state machine, and executing the associated monitoring statements. These activities are carried out in our debugger by the components of the state machine with the assistance of an event subscription mechanism.

In our debugger, events are identified from event sources (e.g., the target program) using *event characterizing terms*. An event characterizing term is a boolean term that gets evaluated by the associated event source when the source performs its actions. An entity that intends to process events from a particular source must subscribe for the events by registering an event characterizing term to this source. During the execution of an event source, if a registered event characterizing term evaluates "true", then the execution of the event source will be suspended to let the subscriber entity associated with this term perform its actions.

The most basic event subscriber entities in our debugger are guarded commands. A guarded command is used to process a specific kind of event. It consists of an event characterizing term and a response action description that specifies

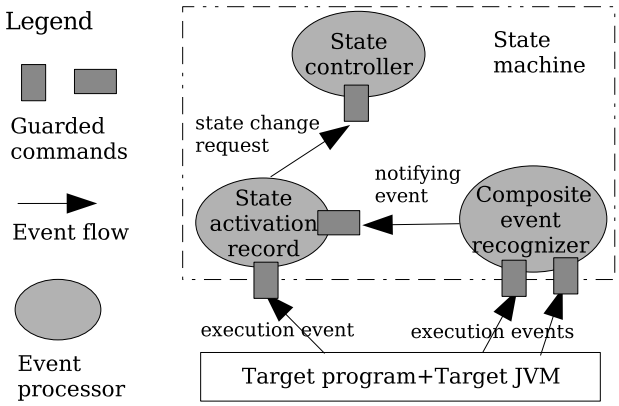


Figure 3: The components for a state machine.

the action to be performed when an event is detected by the term. To allow the response action to inspect the program state, the event characterizing term may be associated with a set of *observing interface variables* that will be initialized with values when an event is detected through the term. A set of guarded commands that process events for a specific purpose are managed by an *event processor*. This event processor will create these guarded commands. The event processor may also allocate memory for the *local state variables* that will be used to hold the information extracted from the program execution. When an event processor is destroyed, all its guarded commands are also destroyed.

A state machine may contain several different event processors to perform the monitoring (see Figure 3). It uses an event processor, the *state activation record*, to represent its current state. The state activation record has a guarded command for processing each type of event that may trigger transitions. During the event processing, a guarded command can evaluate whether a transition should be fired. If this is the case, 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: it deletes the state activation record for the current state, and creates a new state activation record for the next state.

In some state machines, a transition may be triggered by events characterized by a compound condition that cannot be directly expressed using a simple 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 a *composite event*. A state machine uses a new 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 an "event-raise" command. The event-raise command issues a notifying event. This notifying event can be subscribed by another guarded command through a special event characterizing term.

In our debugger, a state machine M_a is allowed to observe and to respond to the state changes in another state machine M_b. In this case, M_a may contain guarded com-

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 and command declaration list>}}
P5: <guarded command declaration> :=
on <guarding expression>**:** <Java block>
P6: <guarding expression> := <short-hand> |
<event type id>[<expression list>](<variable list>)
P7: <transit statement> := **transit** <state id>(<expression list>)
P8: <on leave command> := **onleave** <Java block>
P9: <ignite statement> := **ignite** <state id>(<expression list>)
P10: <event raise statement> := **raise** (<expression list>)
P11: <state machine manager declaration> :=
manager <model id> <manager name>

Figure 4: The syntax productions for key constructs for specifying behavior views.

mands that subscribe to the state controller of M_b. Making the state changes observable allows the behavior views to be specified in a hierarchical way: low-level behavior views can be defined over execution events, whereas high-level behavior views can be defined over the state change events of the state machine defined by lower level behavior views. This approach would allow the software developers to better deal with the monitoring of complex program tasks.

4. A DEBUGGING LANGUAGE

This section presents our debugging language. The language consists of two components: a specification language for behavior views; and a command language to support the user's interaction with the debugging console. We will discuss these two components separately.

4.1 A Language for Specifying Behavior Views

We propose a domain-specific language for specifying the behavior views. This language extends Java: it relies on the constructs provided by Java, and introduces new constructs only when it is necessary. This approach can lower the learning curve. Figure 4 shows a list of syntax productions for the key constructs for specifying the behavior views.

4.1.1 The state machine models

The components that are required for the state machine of a behavior view are specified within a state machine model declaration. A state machine model declaration (production $P1$ in Figure 4) is specified with a keyword `model`, followed by a model id and a model body that contains a list of declarations for state variables, methods, states, and composite events. The state variables are used for storing the information extracted from the program execution during the progress of the monitored task. Such information may be needed for determining the relevancy of a program action to this task. Such information may also be needed for checking properties that range over several actions performed for the task. The methods are used for specifying the computation over the information extracted from the target program.

Some of the methods declared in a model are constructors, one of which will be called when a state machine is created using the model. A constructor method initializes the state variables. The constructor may use an ignite statement (production $P9$) to set the initial state for the created state machine. Like a return statement, an ignite statement will terminate the control flow of the constructor. 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.

4.1.2 The bootstrap blocks

In a state machine model declaration, *bootstrap blocks* are used to specify the event processors that represent the state activation records and the composite event recognizers. A boostrap block (production $P4$) consists of a list of declarations for the local state variables and the guarded commands that will be created for the event processor. These declarations are enclosed in double curly brackets.

A guarded command declaration (production $P5$) is specified with a keyword `on` followed by a guarding expression, a colon, and a response block, which is a normal Java block. A guarding expression specifies the event characterizing terms for the guarded commands that will be created from this declaration. A guarding expression (production $P6$) is composed with an event type id, a list of attribute expressions, and the declaration of a list of observing interface variables. For execution events or state change events, the event type ids are predefined. For composite events, the event type ids are defined in the declarations of the composite events.

The attribute expressions in a guarding expression are used for configuring the event characterizing terms created from this guarding expression. An event characterizing term used in our debugger may have a set of attribute parameters that can be used for refining the events being identified. For example, an event characterizing term for the method entry event may have attribute parameters to allow identifying the events only for a specific method. When an event characterizing term is created from a guarding expression, the attribute expressions of this guarding expression will be evaluated. The resulted values will be bound to the attribute parameters in the event characterizing term.

Table 1 shows an initial set of primitive event type ids and the formats of the corresponding guarding expressions. For most of the event types, the guarding expressions can be type-specific or instance-specific. A type-specific guarding expression identifies events that occur under the contexts of all the instances of the specific type (a class or a model). An instance-specific guarding expression identifies events that occur only under the context of the specific instance.

To improve the readability of the state machine model, we have also developed a set of shorthands for specifying the guarding expressions. For example, we use `enter A.m(int p)` for identifying the method entry events for `A.m(int)`; this shorthand also provides an observing interface variable for the formal parameter of the method. These shorthands will be translated into the basic form during compiling.

Type	*Formats of the Guarding Expression*
reach a line	`reach`[⟨class⟩,⟨method⟩,⟨line⟩](⟨Env⟩†) `reach`[⟨object⟩,⟨method⟩,⟨line⟩](⟨Env⟩†)
method entry	`enter`[⟨class⟩,⟨method⟩](⟨Env⟩) `enter`[⟨object⟩,⟨method⟩](⟨Env⟩)
method exit	`exit`[⟨class⟩,⟨method⟩](⟨Env⟩) `exit`[⟨object⟩,⟨method⟩](⟨Env⟩)
field	`read`[⟨class⟩,⟨field⟩](⟨Env⟩) `read`[⟨object⟩,⟨field⟩(⟨Env⟩)
field write	`write`[⟨class⟩,⟨field⟩](⟨Env⟩) `write`[⟨object⟩,⟨field⟩](⟨Env⟩)
exception throw	`throw`[⟨exception class⟩](⟨Env⟩)
state entry	`enterSt`[⟨model⟩,⟨state⟩](⟨Env⟩) `enterSt`[⟨machine⟩,⟨state⟩](⟨Env⟩)
state exit	`exitSt`[⟨model⟩,⟨state⟩](⟨Env⟩) `exitSt`[⟨machine⟩,⟨state⟩](⟨Env⟩)
transition	`go`[⟨model⟩,⟨state⟩,⟨state⟩](⟨Env⟩) `go`[⟨machine⟩,⟨state⟩,⟨state⟩](⟨Env⟩)

†⟨Env⟩ is a variable referring to an environment object for accessing the runtime entities and their program syntax information of the target program.

Table 1: Identifying various kinds of events.

4.1.3 The state declarations

A *state declaration* specifies the state activation record for a particular state in a state machine. A state declaration (production $P2$) starts with a keyword `state` followed by a state id, a list of formal parameters, and a bootstrap block as its body. The formal parameters of the state declaration can be used for initializing the local state variables declared in the body and for configuring the guarded commands created from the guarded command declarations. The response block for a guarded command declaration in the body of a state declaration can contain *transit* statements whose executions can cause state changes. A transit statement (production $P7$) is specified with a keyword `transit` followed by the id of the target state and a list of actual parameters whose values will be used to configure the creation of the state activation record for the target state. A transit statement computes the values for the actual parameters, terminates the control flow of the enclosing response block, and issue a state change request that contains both the id of the target state and the values of the actual parameters. When the state controller receives this request, it will destroy the current state activation record, bind the values of the actual parameters to the formal parameters of the target state, and create a new state activation record for the target state. The body of a state declaration may also contain an *on-leave* command declaration (production $P8$) that specifies the actions to be performed when a state machine leaves the state specified by this declaration.

4.1.4 The composite event declarations

A *composite event declaration* specifies the event recognizer that will be used to identify a specific kind of composite event. This declaration (production $P3$) is specified with a keyword `event`, followed by an event type id, a list of attribute parameters, a list of observation value types, and a bootstrap block as its body. The attribute parameters are used for initializing an event recognizer when the recognizer is being created from the body of this declaration. This creation occurs when a guarding expression referring to the event type id of this declaration is being evaluated.

The response block for a guarded command declaration in a composite event declaration can contain *raise* statements whose executions signal the recognition of a composite event. A raise statement (production $P10$) is specified with a keyword `raise`, followed by a list of expressions whose types match the observation value types defined at the beginning of the composite event declaration. A raise statement computes the values for these expressions, terminates the control flow of the enclosing response block, and issues a notifying event. The values of these expressions will then be bound to the observing interface variables of the event characterizing terms that subscribe for this event.

4.1.5 Accessing variables/objects in target program

A monitoring statement in the response block of a guarded command should be able to access the values of local variables (including "this" and formal parameters that are visible at the location where the target program is interrupted when the guarded command is processing an event. These variables can be accessed through a special API provided by JPDA. In a state machine model declaration, the value for a variable in the target program may be referred to by extending "@" to the beginning of the name for the variable. A monitoring statement should also be allowed to access to the fields and the methods of an object in the target program through the API provided by JPDA. We require such accesses to be specified by the "->" operator, instead of the usual "." operator. The "->" operator is not governed by Java's accessibility rule. Thus, it allows the monitoring statements to inspect the object states without any restriction.

In some situations, a state machine model may declare variables for storing references to the objects in the target program. In this case, we annotate the type names for such variables with "@" in the variable declarations.

4.1.6 An example

Figure 5 illustrates the state machine model `Login` that is specified using our specification language for the behavior view defined in Figure 1. Note that, in this model, the guarding expressions are specified with shorthands; and the transitions to state `S6` are now triggered by composite events built on top of the exit event of `A.login()`. In addition, the model uses a state parameter to pass the return value of `auth.verify()` from state `S4` to state `S5`, instead of through a shared variable. This example shows that the use of composite event declarations and state parameters may lead to more concise model declarations.

4.1.7 The creation of state machines

A state machine can be created from a model in a way similar to the creation of Java class instances. In this case, the created state machine must be explicitly destroyed. A state machine can also be created by a state machine manager. A state machine manager manages a group of state machines of a specific model. It offers methods for creating/destroying individual state machines and iterating through all its state machines. A state machine manager can be declared (production $P11$) as a local state variable or a state variable in a state machine model. In these cases, the life-time of this state machine manager would be restricted by the life-time of its container. When a state machine manager is destroyed, all its state machines will also be destroyed.

```
model Login {
  @TextBox uBox, pBox;
  @Authorizer auth;
  @String user, password;
  Login(@Athorizer au,
     @TextBox uB, @TextBox pB) {
    uBox = uB; pBox = pB;
    auth = au;
    ignite S1();
  }
  state S1() {{
    on exit uBox.getText(): {
      user = $return;
      transit S2();
    }
    on loginDone[false](): {
      transit S6();
    }
  }}
```

```
state S2() {{
  on exit pBox.getText(): {
    password = $return;
    transit S3();
  }
  on loginDone[false](): {
    transit S6();
  }
}}
state S3() {{
  on enter auth.verify
        (@String u,@String p): {
    assert(u==user&&p==password);
    transit S4();
  }
  on loginDone[false](): {
    transit S6();
  }
}}
```

```
state S4() {{
  on exit auth.verify(*): {
    transit S5($return);
  }
}}
state S5(boolean b) {{
  on loginDone[b](): {
    transit S6();
  }
}}
state S6() {{   }}

event loginDone[boolean rt](){{
  on exit A.login(): {
     assert($return==rt);
     raise;
  }
}}
} // end of model Login.
```

Figure 5: A specification of the behavior view for the login task.

4.2 The Command Language

The command language for our debugging console extends the standard commands that are available in many traditional debuggers. It includes a set of new commands that allow software developers to use behavior views for monitoring the progress of program tasks. We will discuss several important features for this language.

The command language allows the use of convenient variables for storing information during debugging. To distinguish from other kinds of variables, the name of a convenient variable starts with "$". A convenient variable can be used to name a state machine or a state machine manager created during debugging. Other convenient variables are untyped. They can be used to hold any type of values. Such a variable is defined when it is assigned with value for the first time. A convenient variable is global: once the variable is defined, it can be used by debugging commands until the variable is explicitly deleted with a *delete* command.

The most outstanding commands provided by the command language are the observing commands. An observing command is essentially a guarded command. It is specified with the same syntax rule as a guarded command declaration ($P5$ in Figure 4). Observing commands provide a powerful mechanism for performing light-weighted observation on the program execution. An observing command may define an *event-based breakpoint* by including a "stop" command in its response block. When a "stop" command is executed, it terminates the control of the response block, and transfers the control to the debugging console to solicit additional interactions from the software developers. This breakpoint mechanism offers a highly flexible way for controlling the execution of the target program. An observing command may use "now" as its guarding expression. In this case, the action specified in the observing command will be executed immediately after this command is entered. Such a command allows software developers to use the full power of the Java statements for inspecting the state of the target program when a breakpoint is hit.

By default, an observing command can be fired only once. That is, once the response action is executed, the observing command will be automatically deleted from the system. To prevent the command from being deleted, the software developers must add a keyword `repeat` in front of this command. We refer to such an observing command as a *repeatable* observing command. A repeatable observing command must be explicitly deleted using a delete command. When a repeatable observing command uses "now" as its guarded expression, the response action of this observing command will be executed every time when a breakpoint is hit. In this case, the observing command is very similar to a monitor in the Java debugger "jdb".

The following sequence of debugging commands illustrates an interactive debugging session that uses the behavior view defined in Figure 5 to monitor the login task in a program.

```
1> load program A;
  Class A loaded, stop at the entry of A.main().
2> load model Login;
  Model Login loaded.
3> on reach A.15: { $uBox = @textbox1; };
4> on reach A.20: { $pBox = @textbox2; };
5> on enter Authorizor.<init>(): {$auth=@this;};
6> on enter A.login(): { stop; };
7> continue;
  Hit breakpoint at the entry of A.login().
8> create $X Login($auth,$uBox,$pBox);
9> on enterSt $X.S6: { stop; };
10> continue;
  Hit breakpoint at the entry of Login.S6.
11>
```

In the above debugging session, the user first loads the target program whose main method is in `A` (command 1). Once the program is loaded, the debugger will invoke `A.main()` and stop the execution at the entry of this method. The user then loads the Login model (command 2), and enters the observing commands (commands 3, 4, 5) to get the references to the objects that will be monitored by the state machine.[3] The user sets a breakpoint at the entry of `A.login()` (command 6) and continues the program (command 7). When the breakpoint on the entry of `A.login()` is hit, the user creates a state machine with the model `Login` (command 8). The user then sets a breakpoint on the entry of the state

[3]We assumes that the text box for user name is created and assigned to "textbox1" before line 15, the text box for password is created and assigned to "textbox2" before line 20 in class A, and "Authorizor" is instantiated only once.

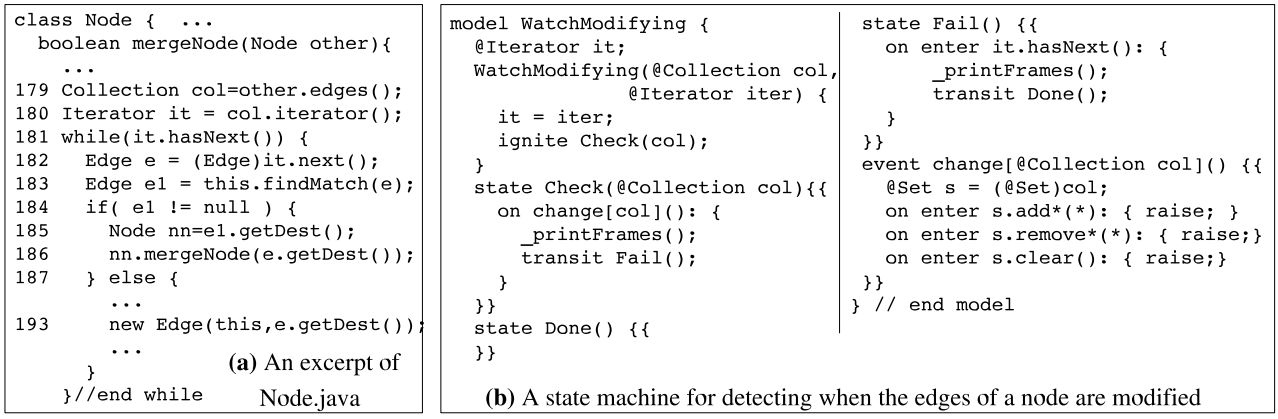


Figure 6: Investigating Bug 1 with a behavior view.

S6 for this state machine (command 9) and continues the program (command 10). The target program is suspended when the state machine transit to the state S6. The user now can enter other commands to continue the debugging.

The commands for a debugging session can be captured in a debugging script that may be automatically read by the debugging console. This can facilitate the repeating of a previous debugging session. This would also allow the monitoring to be performed without user's intervention.

5. EVALUATION: A CASE STUDY

We have implemented a prototype for the scenario-driven debugger and performed a case study to evaluate the effectiveness of localizing faults using the debugger. Our prototype is built on top of the Java Platform Debugger Architecture (JPDA). The prototype uses Java objects to represent the components (e.g., event processors, guarded commands) for a state machine. These objects are created from the monitor classes generated by a compiler from the state machine model declarations. Our current version of the compiler uses a preprocessor to translate the special syntax in the model declarations into the standard Java syntax, and invokes the standard Java compiler for generating bytecode for the monitor classes. To support the observing commands, our debugging console is implemented as an event processor whose guarded commands are created from the description of the observing commands that the user submits. These guarded commands will be deleted when the observing commands are deleted by the user. Our current prototype only provides a terminal-style text-based visualizer. Graphical visualizers will be developed in the future.

We used our prototype debugger to investigate two bugs that we encountered during the implementation of a Java alias analysis system. These bugs are related to the construction of points-to graphs [9]. This construction involves about 80 Java classes. The bugs had been detected and localized through code reading and execution with inserted print statements, and had been fixed. Because of the ineffectiveness of this debugging approach, locating the root causes for the bugs had required significant efforts.

5.1 Bug 1

The symptom of this bug is that, when the analysis system is run on some subject programs, it terminates abnormally with exception "java.util.ConcurrentModificationException" in "Iterator.next()" called at line 182 in file "Node.Java". Figure 6(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 the 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 6(b) shows a state machine model for monitoring when the edges of a node may be changed within the loop in an invocation of `mergeNode()`. The debugger creates a state machine before the target program enters the loop in this invocation. The state machine monitors the "set" object that maintains the edges for the "other" node and the iterator object that is used to go over these edges. The state machine first enters the `Check` state to monitor the method calls that can change the content of the set. If the change does occur, the stack frames will be printed and the state machine will transit to the `Fail` state. Within `Fail`, the state machine will wait until `hasNext()` method of the iterator is invoked. At this point of time, the program execution must be back to the beginning (line 181) of the `while` loop in the original invocation of `mergeNode()`. The state machine prints out the stack frames and transit to the `Done` state. Comparing the stack frames printed out in `Check` and in `Fail` would reveal the method calls in the loop that lead to the changes of the edges. To use the model, we enter the following commands

```
2> load model WatchModifying
3> createmanager $M WatchModifying
4> repeat on reach Node.181:{ $M.new(@col,@it); };
5> repeat on enterSt WatchModifying.Done: {stop;};
6> continue;
```

We used a state machine manager to create a state machine from the model when the loop is about to start in each invocation of `mergeNode()` (command 4). We wanted to stop the execution of the target program when a state machine reach `Done` (command 5). During the debugging, by comparing the stack frames printed out in `Check` and `Fail`, we discovered that the edges of the node referenced by "other" in one invocation of `mergeNode()` may be changed by the statement at line 193 in the recursive invocation of `mergeNode()` that is called at line 186. That is, "nn" at

line 186 must point to the same object as "other" in the first invocation of `mergeNode()`. We changed the recursive `mergeNode()` into a worklist algorithm to avoid the concurrent modification exception.

5.2 Bug 2

This bug appeared in the implementation after Bug 1 was fixed. The symptom is that, after the graphs are constructed, 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"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". It has to ensure that the labels on the edges leaving "n1" are unique after the merging. This requires the successors of "n1" and "n2" to be merged when necessary. A worklist is introduced to maintain the list of pairs of nodes that need to be merged.

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 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 7 shows the model for the state machine. To use this model, we entered the following commands

```
3> repeat on reach Analysis.50: { \
     create $M CheckEquiv(@worklist); \
   };
4> repeat on reach Analysis.92: { delete $M;};
5> continue;
```

The worklist algorithm is captured in a method in `Analysis`. This method may be called many times when the target program is being run. The worklist is created before line 50; and the algorithm terminates at line 92. Therefore, we want to create a state machine to monitor the worklist when the execution reaches line 50 (command 3); and we want to delete this state machine when the execution reaches line 92 (command 4). We observed that the property is violated in some cases. Further investigation suggested that the problem would occur 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. We were able to confirm this suspicion by enhancing the the model with additional monitoring statements.

5.3 Discussion

This case study demonstrates the benefits of using the scenario-driven debugger. The bugs under investigation are

```
model CheckEquiv {
  @List wlist;
  Map node2equiv;
  Vector equivs;
  // equivs keeps all the
  // equivalence classes
  CheckEquiv(@List wl) {
    wlist = wl;
    ignite Checking();
  }
  state Checking() {{
   on enter 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++) {
   Object[] v=((Vector)equivs
               .elementAt(i))
               .toArray();
   @Node n0 = ((@Node)v[0])
              ->traceForward();
   for(int j=1;
       j<v.length;j++) {
      @Node n=((@Node)v[j])
              ->traceForward();
      assert(n==n0);
   } // end for
 } // end for
}
} //end model
```

Figure 7: A behavior view for investigating Bug 2.

not caused by faulty program statements, but caused by an erroneous algorithm design that may go wrong only when the objects are related to one another in certain ways. Therefore, monitoring the execution of individual program statements with traditional debugging techniques would not be effective in finding the errors in the design. To find out these errors, the user must collect information from multiple steps at which a particular set of objects are being processed, and verify properties over the collected information at appropriate points of time during the program execution. As shown in the study, because a behavior view can model the progress of the scenarios in which these objects may be processed, it is quite convenient to specify and to check these properties using the scenario-driven debugger.

One limitation with this study is that we only investigated the scenarios for low-level program tasks. Thus, we must be cautious in generalizing the conclusion to higher level program tasks, like those identified during requirement analysis. Because a higher level task may involve more objects and more codes, we expect our debugger to be even more convenient for checking the implementation of this task. Another 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 debugger is even more useful because, in such a situation, more assumptions or properties need to be checked. Studies need to be performed to verify these hypotheses.

6. RELATED WORK

Event-based behavioral abstractions were first used in a high-level debugger for distributed systems [3]. The ideas have been extended to build execution monitoring frameworks for sequential programs (e.g., [1, 2, 6, 12]). Each of these techniques offers a mechanism for identifying the execution events of interest, defining high-level events, and specifying actions to be performed when an event is detected. Our debugger also provides these functionalities. In addition, our debugger introduces a higher level behavioral abstraction for capturing the expectations for the scenarios of the program tasks. It allows the software developers to effectively use their design knowledge of these tasks during debugging.

The design-level debugging techniques have been proposed to allow "driving and monitoring the debugging process from a design model viewpoint" [15], and have been implemented in Rhapsody [15] and Fujaba [7]. Both Rhapsody and Fujaba can generate codes from the design specified with UML or UML-like diagrams. By utilizing the model/code associativity derived during the code generation, the debugger will be able to map the statement under execution to the elements in the UML diagrams, and thus, would allow the software developers to visualize and control the execution based on the design diagrams. Our debugger can be viewed as a new design-level debugger that targets the application domains where automatic code generation may be impractical. Our debugger also provides strong support for checking, automatically or interactively, important properties with respect to the progress of of a program task. No comparable support has been discussed in [7, 15].

Assertions have been widely used in practice for debugging purpose. Annotation languages, such as ANNA [10] and JML [5], have been introduced for specifying pre-/post-condition and invariant assertions that can be automatically checked during runtime. Similar properties can also be specified with the UML Object Constraint Language (OCL) and checked at runtime for Java programs [11]. Similar to these techniques, our debugger also allows software developers to specify properties that can be checked automatically during runtime. However, unlike the existing techniques that specify the assertions only at the class level or the method level, our debugger allows the specification of the assertions at the behavior-view level. Because a behavior view explicitly models the progress of the program task, using the model may make it easier to specify properties that span several steps during the progress of the task. In addition, in our debugger, monitors that check assertions can be created from behavior views interactively without recompiling and restarting the target program. This flexibility may be needed during the investigation of a bug.

Some existing techniques (e.g.,[1, 2, 3, 4, 12]) allow the specification of properties for checking the correct sequencing among the program actions. A behavior view differs from such a specification in that it allows sophisticated monitoring to be specified on top of a state machine that not only specifies the sequencing among the actions, but also models the progress of the scenarios.

7. CONCLUSION

This paper presents the core concepts for a scenario-driven debugger that provides the behavior view as a new abstraction for monitoring and inspecting the progress of scenarios during debugging. A behavior view captures the user's expections for the scenarios. Our case study shows that behavior views may be quite useful for identifying the root causes for software bugs during debugging.

Comparing to the traditional debugging techniques, monitoring program execution with behavior views may require additional time for writing the behavior view specifcations. However, because these specifications allow many monitoring activities to be automated, this up-front time investment may significantly reduce the time required for performing and managing these activities. Therefore, using behavior views is expected to cut down the total debugging time for finding the root cause for a particular bug. Comparative studies are required to assess this time saving.

In our future work, we will focus on improving the usability of the scenario-driven debugger. We will construct a GUI front end for editing the models and for visualizing the progress of the modelled scenarios, and integrate the debugger with the Eclipse development environment. We will also investigate techniques for deriving the models from various design artifacts (e.g., message sequence diagrams) created to model scenarios during the software design. We will perform more studies to evaluate the usefulness and the usability of our debugger.

8. REFERENCES

[1] M. Auguston. A program behavior model based on event grammar and its application for debugging automation. In *AADEBUG'95*, May 1995.

[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] P. Bates and J. Wileden. High-level debugging of distributed systems: The behavioral abstraction approach. *The Journal of Systems and Software*, (3):255–264, 1983.

[4] 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.

[5] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (to appear)*, 2005.

[6] M. Ducasse. Coca: An automated debugger for C. In *ICSE'99*, pages 504–513, May 1999.

[7] L. Geiger and A. Zundorf. Graph based debugging with fujaba. *Electronic Notes on Theorectical Computer Science*, 72(2), 2002.

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

[9] 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.

[10] D. C. Luckham, S. Sankar, and S. Takahashi. Two-dimensional pinpointing: Debugging with formal specifications. *IEEE Software*, pages 74–84, 1991.

[11] D. J. Murray and D. E. Parson. Automated debugging in java using OCL and JDI. In *AADEBUG 2000*, 2000.

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

[15] J. Stanglewicz. Design-level debugging. *Real-Time Magazine*, pages 68–72, 1999.

[16] M. Telles. *The Science of Debugging*, chapter 8: The General Process of Debugging, pages 205–240. The Coriolis Group, 2001.