

Monitoring with Behavior View Diagrams for Debugging

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

Abstract

UML sequence diagrams are widely used during requirements analysis and design for specifying the expected message exchanges among a set of objects in various scenarios for the program to perform a certain task. In this paper, we present the behavior view diagrams, a type of extended sequence diagrams, to facilitate execution monitoring during debugging. Using a behavior view diagram, software developers can precisely specify the runtime objects whose behaviors will be monitored during debugging. Software developers can also specify the important message exchanges to be observed among these objects during the progress of various scenarios, and may further define the monitoring actions to be performed for inspecting the program state when a message exchange is observed. We also present a debugger that can automatically monitor the program execution using the information specified in a behavior view diagram. Through this monitoring, the debugger can not only check whether the scenarios are progressed as intended, but also check whether the actions performed by the program have the the desired effects on the program states. Therefore, it will be useful for detecting and localizing bugs.

1 Introduction

Modern software systems must implement very complex behaviors to meet their requirements. To identify and document the expected behaviors of such a system, software development methodologies (e.g., [6]) often encourage the identification and specification of the scenarios in which the system may proceed to perform program tasks. A *scenario* is identified as a sequence of interactions among the components of the system for performing a particular task.¹ Focusing on scenarios during requirements analysis and design allows developers to employ a divide-and-conquer strategy in capturing the expected behaviors of a system.

¹In literature, a scenario sometimes is only used to refer to a sequence of interactions between a software system and its users. Our definition extends this meaning.

Software developers often make mistakes during the analysis, design, and implementation of a complex system. These mistakes introduce software bugs that must be uncovered and removed through debugging. Software debugging is an extremely challenging cognitive activity. During debugging, software developers often monitor the program execution, observe the dynamic behaviors of the system, and check the consistency between the observed behaviors and the expected behaviors of the system. Because the expected behaviors have been identified as scenarios during requirements analysis and software design, the software developers should perform their monitoring activities based on scenarios. This *scenario-driven* debugging approach allows software developers to effectively utilize their knowledge of scenarios built during analysis and design to detect and pinpoint problems in the program.

Existing debugging 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, 4, 9]), 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 monitor and observe the dynamic behaviors during debugging. However, because these techniques do not emphasize on correlating the monitoring of the program actions at different points of time during execution, they provide inadequate support for the observation and inspection of the progress of the scenarios for a particular task. Such support is essential for the scenario-driven debugging.

The goal of our research is to develop debugging techniques that can better support the scenario-driven debugging approach. In the previous work [7], we proposed behavior views as a new abstraction to facilitate the automation of the execution monitoring during scenario-driven debugging. A behavior view uses execution events to identify the important milestones that characterize the progress of the scenarios for a task. A behavior view may also contain monitoring statements for inspecting the program states when the

program execution reaches these milestones. Using the information provided by a behavior view, a debugger can automatically track the progress of the scenarios and check various properties at each step to ensure that the program actions performed by the program have the right effects on the program states. Our preliminary studies show that behavior views may be quite useful for locating the root cause for design and implementation bugs.

In this paper, we investigate the specification of behavior views based on an important type of execution event in an object-oriented (OO) system: the message exchanges among objects. In an OO system, objects collaborate to perform a particular task. This collaboration is carried out by a sequence of message exchanges among these objects. In a sequential program, an object can perform its actions only when it receives messages from other objects. Therefore, the occurrences of message exchanges often mark the important milestones in the progress of the scenarios. We refer to the behavior views specified based on the message exchanges as the *message-based* behavior views.

This paper presents our extension to UML(the Unified Modelling Language [10]) 2.0 sequence diagrams for defining the message-based behavior views along with a debugger that uses the information specified by such extended notations to monitor the program execution automatically. A sequence diagram uses a graphical notation to represent the possible sequences of message exchanges that would occur among the objects under various scenarios to perform a certain task. Therefore, this notation can also be used for specifying the message exchanges that we want to monitor during debugging. We further extend the syntax and the semantics of this graphical notation to precisely specify the runtime objects whose behaviors will be monitored, and the properties to be checked or the monitoring statements to be executed when the occurrence of a message exchange is detected. We refer to a diagram specified with the extended notation as a *behavior view* diagram.

One benefit of using the behavior view diagram is that it provides a high-level abstraction for software developers to specify their expectations of the scenarios for performing a specific task. These expectations can be automatically checked against the program execution by our debugger. Specified on top of a graphical notation that has been widely used for documenting scenarios during analysis and design, a behavior view diagram enables the software developers to directly use their design knowledge to investigate bugs. Therefore, it directly supports the scenario-driven debugging approach.

Another benefit of using the behavior view diagram is that it enables a methodological improvement in debugging: the reuse of design artifacts for execution monitoring. A behavior view diagram can be viewed as a sequence diagram enhanced with monitoring specific information. Therefore,

it should be easy to derive a behavior view diagram from a sequence diagram specified during design. In fact, using some of the extensions that we propose, it may be possible to produce “monitored” sequence diagrams during design. This would enable the direct reuse of these sequence diagrams for debugging. This reuse not only improves the efficiency of debugging, but also adds incentives for a software team to produce and maintain the design artifacts.

2 Debugging with Message-Based Behavior Views

This section gives an overview of the message-based behavior views and a debugger that uses these views.

2.1 The Message-Based Behavior Views

One important activity in the scenario-driven debugging approach is to monitor the program execution and check whether the scenarios for a program task progress as intended. To perform such monitoring, the software developer must detect the important milestones during the progress of the scenarios for performing this task, stop the program execution at these milestones, and inspect the program state to ensure that the actions performed during the execution have the expected effect on the program state. We develop the message-based behavior views for specifying the monitoring requirements and a tool to automate the monitoring process based on these monitoring requirements.

A message-based behavior view precisely specifies the set of runtime objects whose behaviors will be monitored. It also specifies the sequences of message exchanges that characterize the progress of the possible scenarios for performing a program task. It may further specify the properties to be checked and the monitoring statements to be executed at various points of time when the scenarios are being monitored. We extend the UML notation for sequence diagrams to specify such a behavior view.

Figure 1 shows a behavior view diagram specified using the extended notation (more details of the notation are explained in Section 3). In the diagram, an object is represented with a vertical line; a message exchange (in this case, a method call or return message) is represented with an arrow going from one vertical line to another. Some arrows are zigzag because they may represent indirect method calls/returns. The behavior view diagram specifies how the login task is expected to be performed in a GUI application. Three of the objects under monitoring are provided as parameters (`uBox`, `pBox`, and `auth`) to the diagram. The other object of `A` is discovered during execution monitoring.

According to the diagram, the login task is expected to start when method `login()` is called on an object of class `A`. During the invocation of this method, `getText()` is

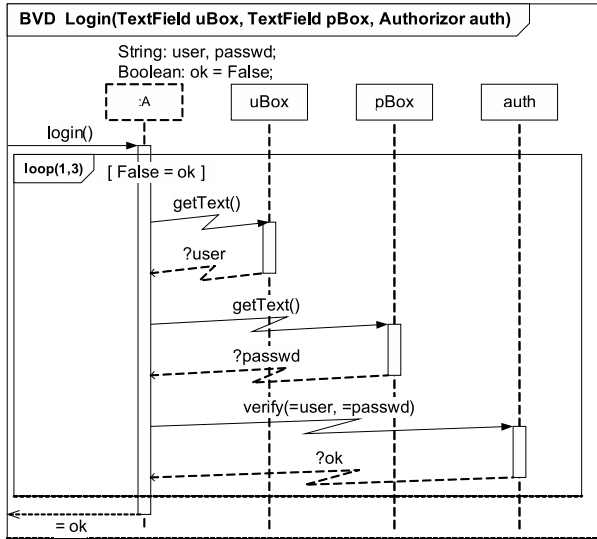


Figure 1. A behavior view diagram.

expected to be called (directly or indirectly) on the GUI text box objects `uBox` and `pBox`, respectively, to get the user id and the password. After that, `verify()` is expected to be called (directly or indirectly) on the authorizing object to see whether the input user id and password are valid. If they are invalid, then these steps can repeat up to three times. Otherwise, `login()` terminates.

The behavior view diagram also contains instructions for the debugger to check whether various values have been computed and used correctly at each step during the progress of the login task. Some of these instructions are specified in the labels of the arrows. The labels associated with the arrows representing the returns of `uBox.getText()`, `pBox.getTex()`, and `auth.verify()` instruct the debugger to store the return values into monitoring variables `user`, `passwd`, and `ok`, respectively. The label associated with the arrow for method call `auth.verify()` instructs the debugger to check whether the values of the two parameters for this method invocation equal to the values of `user` and of `passwd`, respectively. The label associated with the arrow representing the return of `A.login()` instruct the debugger to check whether the return value of this method equals to the value of `ok`. The checking ensures that the values input through the text boxes have been correctly used, and that login succeeds only if `auth.verify()` returns “true”.

Note that, although a behavior view diagram is used to check the behaviors of an implementation, it is specified independent of the code, and can be specified even before the code is constructed. It does not need to reflect the actual structure of the code. For example, in Figure 1, we use a loop to specify the three login attempts. However, in the actual implementation, this may be done with three consecu-

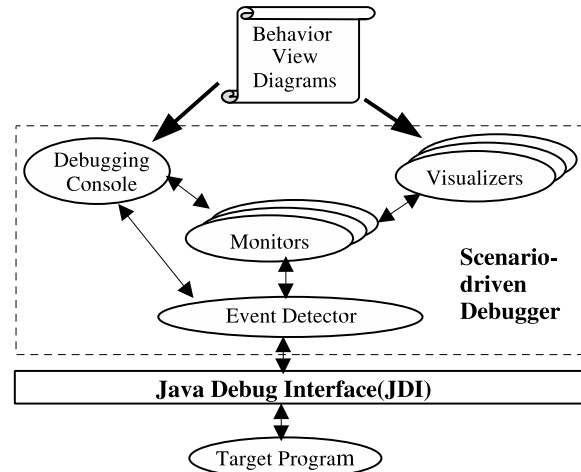


Figure 2. The Architecture for the Debugger.

tive blocks, or even with a recursion. In addition, a behavior view diagram will not need to specify every object involved in the scenarios or every message exchange that may occur among the objects represented by the life-lines. In other words, a behavior view diagram is a behavioral abstraction. This capability of abstracting away some implementation details is one of the main advantages of using behavior view diagrams for debugging.

2.2 A Scenario-Driven Debugger

We have extended the scenario-driven debugger presented in our previous work [7] to use the behavior view diagrams for monitoring. Figure 2 shows the architecture of the debugger. The debugger is built on top of the Java Platform Debugger Architecture (JPDA).² The debugging client and the target program are running on separate Java Virtual Machines (JVMs). The client can interact with the target JVM through the Java Debug 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, and an event detector. The debugging console provides an interface for entering debugging commands. The monitors are created from the behavior view diagrams for monitoring the execution of a program. A monitor may be connected to one or more visualizers that accept the information output from the monitor and present the information to the user with various visualization techniques. The event detector accepts event descriptions from the debugging console and the monitors, and interacts with JDI to detect the events of interest.

The debugging console can accept various kind of user

²See <http://java.sun.com/products/jpda/index.jsp>.

commands. The following sequence of debugging commands illustrates an interactive debugging session for using the behavior view diagram defined in Figure 1 to monitor the login task in the program.

```
1> load program A;
   Class A loaded.
2> load diagram Login;
   Login diagram loaded.
3> on reach A.15: { $uBox = @textbox; };
4> on reach A.20: { $pBox = @textbox; };
5> on enter Authorizer.<init>(): \
   { $auth=@this; };
6> on reach A.40: { stop; };
7> continue;
   Hit breakpoint at A.40.
8> create $X Login($uBox,$pBox,$auth);
9> on catch $X.error: \
   { $X.display(); stop; };
10> continue;
```

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 invokes `A.main()` and stops the execution at the entry of this method. The user then loads the Login diagram (command 2), and enters the observing commands (commands 3, 4, 5) for obtaining the references to the objects whose behaviors will be monitored. An observing command specifies an execution event to be monitored and a monitoring action to be performed when the event is detected. For example, command 3 specifies that, when the program execution reaches line 15 in A, the value of local variable `textbox` will be extracted (`@` is the value extracting operator) and assigned to the monitoring variable `$uBox`. The user then enters another observing command to set a breakpoint at line 40 in A (command 6). The user enters “continue” to resume the execution of the target program (command 7). The target program stops when it hits line 40 in A. The user enters a command to create a monitor using the Login diagram (command 8). The user then enters an observing command to catch the error signal raised by the monitor (command 9): when such a signal is caught, the debugging console can display the status of the monitor and transfer the control to the user interface for further investigation. The user resumes the execution of the target program (command 10). From now on, the monitor created from the Login diagram will monitor the login task. Details on the formats of the debugging commands can be found in [7].

3 Extending UML Notation for Specifying Behavior Views

This section presents several extensions to the syntax and the semantics of the UML notation for sequence diagrams to specify behavior view diagrams.

3.1 Specifying Message Exchanges with Interaction Fragments

In UML 2.0, sequences of message exchanges are specified with interaction fragments. We extend the notation for interaction fragments for specifying the message-based behavior views. As shown in Figure 1, an interaction fragment is surrounded by a frame. The top-left corner of the frame has a pentagon area (the *descriptor*) for describing the content inside the frame. An interaction fragment may also declare local attribute variables. These variables can be used for the monitoring actions to maintain information. We also refer to these variables as *monitoring* variables.

As shown in Figure 1, the sequences of message exchanges are specified with a two-dimensional chart within an interaction fragment. In the chart, the x-axis shows the roles that the individual objects would play in the interaction, and the y-axis (from top down) shows the progress of time. Each role in the diagram is represented with a *life-line* that consists of a rectangle *head* followed by a vertical line. The head can specify a role name and the type for the object that may play such a role. A message occurrence is represented in the diagram as an arrow from the life-line of the sender object to the life-line of the receiver object.

In this paper, an interaction fragment is used to monitor one single thread in an object-oriented program. Therefore, we consider only messages for the method calls, the method returns, and the object creations. Each of these messages can be seen as the result of the execution of a statement during a method invocation that is caused by another method call message. From this perspective, an interaction fragment can be viewed as specifying what method invocations are interesting to the software developers and how each of these method invocations is expected to progress. For example, Figure 1 shows the four kinds of method invocations that are interesting to us; it also shows that each invocation of `A.login()` should call `uBox.getText()`, `pBox.getText()`, and `auth.verify()` at most three times in this order. We refer to the method invocation that sends a message as the *sending* invocation of the message.

An invocation of a method is represented by an execution specification in an interaction fragment. This execution specification can be explicitly represented as a thin vertical box that covers a part of the life line representing the receiver object. The messages sent or received during this invocation will be associated with this box. In many cases, the messages associated with an execution specification can be easily inferred by inspecting the relative positions of the arrows that represent these message on a life-line. Therefore, the box representing the execution specification is often not explicitly shown in the diagram. To simplify the discussion in this paper, we assume that the box always exists for the execution specification of a method invocation.

We extend the notation for interaction fragments to include *indirect* messages that are represented as zigzag arrows (see Figure 1). An indirect method call message indicates that the target method may be called indirectly by the sending invocation of this message. An indirect method return message indicates the return of the control to the sending invocation of the corresponding indirect method call message. The indirect messages can be used by software developers to avoid specifying uninteresting objects on the call path between the sender and the receiver.

An interaction fragment can be specified as a standalone diagram. Such a fragment can specify a name with an optional list of parameters and an optional return value type. By using this name, another interaction fragment may call into being the message exchanges represented by this interaction fragment. This composition mechanism enables software developers to modularize the specification of a large number of message exchanges among objects for complex scenarios. Therefore, they can employ the divide-and-conquer approach for specifying these message exchanges. We mark a standalone interaction fragment with BVD to indicate that it is a behavior view diagram.

An interaction fragment may also be embedded inside another interaction fragment. This kind of fragment is called *combined* fragment. Each combined fragment has an operator that indicates how the message exchanges specified by this fragment will be interpreted under the context of the containing fragment. This operator is specified in the frame descriptor of a combined fragment. A fragment with `ref` operator represents an *interaction use* that calls another sequence diagram.³ A fragment with `loop`, `opt`, or `alt` operator represents repeated, optional, or alternative message exchange groups respectively. A combined fragment with `alt` operator will be divided into several horizontal lanes (or the *interaction operands*), each of which represents a choice. There are yet several other types of combined fragments in UML 2.0. We do not consider these types of combined fragments in this paper either because they are not relevant to sequential programs or because their meanings are too complicated and/or not well defined. A combined fragment typically shares the life-lines of the containing interaction fragment; it may also have its own local life-lines that are only used within its frame.

3.2 Binding Life-Lines

Message exchanges specified in an interaction fragment occur among objects. Therefore, to monitor these message exchanges, a debugger must be able to identify the runtime objects that play the roles identified by the life-lines in the diagram. That is, the debugger must be able to *bind* each

³To ease the discussion in this paper, we treat an interaction use as a combined fragment.

life-line in an interaction fragment to exactly one runtime object. We introduce five different mechanisms for specifying such bindings. Two of these mechanisms use existing syntax of the interaction fragments. The other three mechanisms introduce new syntax. Note that, for each life-line, only one binding can be specified.

The first two kinds of bindings are the *parameter-based* bindings and the *position-based* bindings. Both bindings are used to specify how the life-lines within a stand-alone interaction fragment can be bound when this fragment is called by an interaction use. With the parameter-based binding, a life-line in a stand-alone interaction fragment is bound to an object referred to by a parameter of the fragment. This parameter is bound with the value of the corresponding argument specified in the calling interaction use. To specify such a binding, the role name specified inside the head of the life-line should be the same as the parameter name (see the three life-lines on the right in Figure 1). With the position-based binding, a life-line in a stand-alone interaction fragment is matched with and bound to a life-line covered by a calling interaction use based on the relative positions of both life-lines. To specify such a binding, the head of the life-line will be drawn with thick-lines in the fragment (see the left life-line in Figure 3).

The next two kinds of bindings are the *creation-implied* bindings and the *invocation-implied* bindings. These bindings are the result of message exchanges. In a creation-implied binding, a life-line is bound to the resultant object of a creation message. In this case, the arrow representing the creation message will point to the head of the life-line to be bound. In an invocation-implied binding, a life-line is bound to the target object of a specific method call. This method call is specified by the first arrow that points to this life-line. Before the detection of this call, the life-line remains unbound. For example, in Figure 1, the left-most life line will be bound to the target object of the call to `A.login()`. To distinguish an invocation-implied binding from the other kinds of bindings, the head of the life-line to be bound in this way will be specified as a dashed box.

The fifth kind of binding is the *value-based* binding. In this case, a life-line is bound to the object identified by the evaluation result of an expression of reference type (see the two short life-lines in Figure 3). Such a binding is specified with a dashed arrow pointing to the head of the life-line to be bound. This dashed arrow is labelled with `bind(expr)`, where *expr* is the expression whose value will be used for binding. The arrow must start from a bound life-line, which is referred to as the *originating* life-line. We refer to the starting point of the arrow on the originating life-line as the *originating* point. The originating point indicates when the binding should occur. If there is a message exchange associated with the originating life-line right before the originating point, then the binding should occur right after the

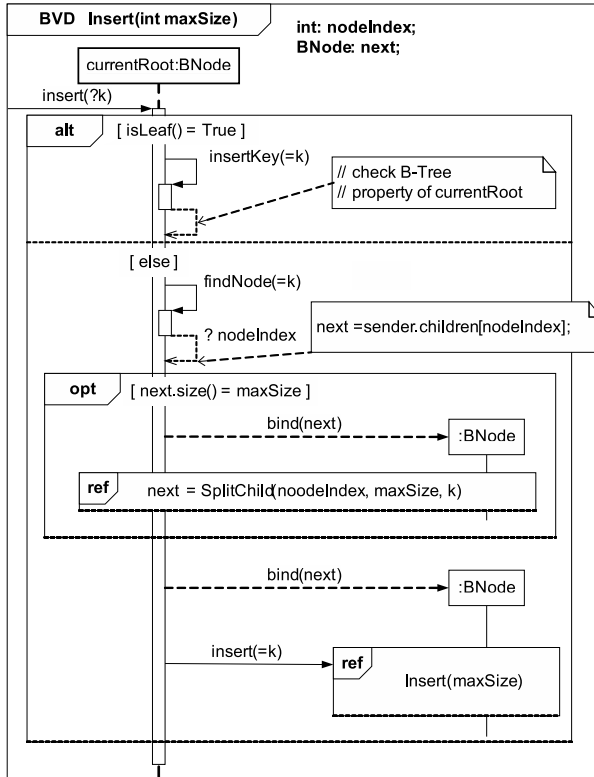


Figure 3. Monitoring the insertion to a B-tree.

message exchange is detected. If there is no message exchange associated with the originating life-line before the originating point, then the binding should occur right after the originating life-line is being bound.

Value-based binding is a powerful mechanism for dynamically determining the objects to be monitored. It is necessary in many monitoring situations. Figure 3 illustrates a behavior view diagram that can be used to monitor the scenarios for inserting a key into a B-tree. To perform this functionality, method `BNode.insert()` will be called on the nodes of a B-tree to find an appropriate leaf for inserting the key. The diagram in Figure 3 specifies that, when `BNode.insert()` is invoked on the current node, if the current node is already a leaf, then `insertKey()` is expected to be invoked to insert the key to this node. However, if the current node is not a leaf, then `findNode()` is expected to be invoked to find the index of the child where the key should be inserted. In this case, if the size (the number of children) of the identified child equals to the maximum allowed number, then the identified child is expected to be split into two nodes. This splitting is described in another behavior view diagram, `SplitChild`, which is not described here due to space limitation. After that, `insert()` is expected to be invoked on the identified child. To monitor this invocation, an interaction use to

the diagram `Insert()` is specified to cover the life-line that is bound to the identified child. From the figure, we can see that, because the interaction use need to be applied to a life-line that represents an object whose identity is not known until certain message exchange occurs, it would be impossible to specify the monitoring that involves this kind of object without using the value-based binding.

3.3 Specifying Properties

UML 2.0 provides several constructs that can be used to specify restrictions that must be satisfied by a correct implementation. These constructs can also be used for specifying properties that will be checked at various points of time during the execution monitoring. Of course, because these constructs were not intended for monitoring, new semantics may need to be defined for the constructs accordingly.

The UML notation allows to specify state invariants on a life-line in a sequence diagram. A state invariant is represented as round box that contains a logical assertion. This assertion must be true when the next event on the life-line occurs. We can use a state invariant to specify a property that must be checked at a particular point of time during program execution. During the monitoring, if a state invariant is evaluated false, then an error will be reported.

The UML notation also allows to specify an interaction constraint that controls the iterations of a loop or the selection of a particular branch in a set of alternatives. An interaction constraint is a logical expression placed in a pair of square brackets. The constraint is typically placed on a life-line at the point right before the first message of the loop or the corresponding branch. In execution monitoring, we use such a constraint to specify a property that must be checked when the first event is detected in the iteration of the loop or in the corresponding branch. For example, in Figure 3, `[isLeaf() = True]` is placed before `insertKey()` is called. Therefore, when a call to `insertKey()` is detected, this constraint will be evaluated. Note that, with this monitoring semantics, the constraints for different branches in a set of alternatives do not need to be mutually-exclusive.

The UML notation further allows to specify, on the method call messages or the method return messages, expressions that define the expected values for the parameters or the expected return values, respectively, of the method invocations. When the messages are detected during execution monitoring, these expressions will be evaluated and compared with the actual values of the corresponding parameters or the actual values returned by the method invocations. If the expected values are different from the actual values, then an error is reported. To emphasize the monitoring semantics, we require an expression that defines an expected value to be preceded with '='. For example, in Figure 1, the method call message for `auth.verify()`

contains expressions `=user` and `=passwd` whose values are used to check the actual values of the two parameters when the call message occurs.

3.4 Specifying Monitoring Statements

We utilize the comment construct for specifying the monitoring statements to be executed in response to the occurrences of the messages specified in an interaction fragment. A comment is a rectangle that associates a block of text with an element in a UML diagram to annotate this element. We introduce a special kind of comment that associates monitoring blocks with various messages. A monitoring block is similar to a method body in a Java program. It can declare local variables. It can contain various kinds of regular Java statements. For example, in Figure 3, the comment associated with the return message of `findNode()` contains a Java statement to assign the reference of the identified child to local attribute variable `next`.

The statements in a monitoring block may use auxiliary objects to assist the monitoring task. For example, these statements may use a vector object for maintaining the information that it collects from the program execution. An auxiliary object can be referenced to by a local variable declared in a monitoring block or by an attribute variable declared in an interaction fragment. We require the name for the type of an auxiliary object and the name for a variable referring an auxiliary object to start with '\$'.

3.5 Accessing Values/Objects in the Target Program

To inspect the program state, a property specification or a monitoring statement may need to access the runtime values/objects in the target program. One way to obtain the runtime values is through the parameters to a behavior view diagram. These parameters will be initialized with runtime values by the debugging console when a monitor is being created from this diagram. Another way to obtain the runtime values from the target program is through the role names for the life-lines. By referring to the role name of a life-line, a reference to the object bound to the life-line can be obtained. The third way to obtain values from the target program is through the implicit variables "this", "sender", and "receiver". Variable "this" can be used by a state invariant or an interaction constraint to refer to the object represented by the life-line covered by the state invariant or the interaction constraint. Variables "sender" and "receiver" can be used by a statement in the monitoring block associated with a message to refer to the sender or the receiver object of the message, respectively.

We also introduce an explicit value extracting mechanism to extract the value of a parameter or the return value

of a method invocation and store this value into a local attribute variable declared in an interaction fragment. This extraction is specified by an extracting expression that contains a question mark operator followed by the name of a variable. To extract the value from a parameter for a method call, an extracting expression will be placed at the corresponding position in the argument list for the message that represents the method call. To extract the return value for a method call, the extracting expression will be placed on the message that represents the return of the method call. For example, in Figure 3, `?nodeIndex` is placed on the method return message of `findNode()` to extract the return value and assign it to local attribute variable `nodeIndex`.

3.6 Summary

In this section, we reuse many constructs provided by UML 2.0 for interaction fragments. However, because these constructs are not designed for execution monitoring, we have to clarify or redefine their semantics. We also introduce several new constructs for specifying information that is crucial for execution monitoring. With these extensions, the notation now becomes a visual programming language for specifying behavior views that can be interpreted by a debugger for execution monitoring.

4 Related Work

Our previous work [7] proposed the concept of behavior views and a textual language for describing these views. Compared to the textual language, the graphical notation for the behavior view diagrams presented in this paper is more direct and intuitive for specifying the expected interactions among objects. These interactions are difficult to debug because they involve actions performed by different objects. In addition, because a behavior view diagram can be derived from a sequence diagram, using behavior view diagrams for monitoring may allow software developers to partially reuse the design artifacts for debugging.

The design-level debugging advocates "driving and monitoring the debugging process from a design model viewpoint" [12]. The scenario-driven debugging approach that we propose is an example of design-level debugging. Rhapsody [12] and Fujaba [5] can support design-level debugging. Both debuggers rely on code generators that generate codes from UML or UML-like diagrams. They use the model/code associativity derived during code generation to map the statement under execution to the elements in the UML diagrams, and thus, would allow the user to visualize and control the execution based on the design diagrams. Our debugger is a new design-level debugger that targets the application domains where automatic code generation may

be impractical. Using the behavior view diagrams, our debugger provides strong support for inspecting the program state during the progresses of the scenarios. No comparable support has been discussed in [5, 12].

In spirit, a behavior view diagram are quite similar to a visual constraint diagram [13]: both types of diagrams extend a form of UML diagram for specifying constraints to be checked during runtime. However, they are designed for different purposes. A visual constraint diagram is used for detecting the object configuration anomalies. It extends a UML object diagram. In contrast, a behavior view diagram is used for detecting the interaction anomalies. It extends a UML sequence diagram. It should be possible to use them together for more effective debugging.

Event-based behavioral abstractions were first used in a debugger for distributed system [2]. The ideas had been used to build debuggers for sequential programs (e.g., [2, 1, 4, 9]) and explored for automatic monitoring of software requirements (e.g., [3, 11]). Our behavior view diagrams focus on one important type of execution event in an OO system: the message exchanges among objects. They use scenarios as a higher level abstraction for organizing the related message exchanges and the monitoring actions to be performed when a message occurs. In addition, they are specified in graphical notation. Thus, these diagrams may be more suitable than other event-based abstractions for monitoring the execution of OO programs.

5 Conclusion

This paper presents behavior view diagrams as a new abstraction for debugging. A behavior view diagram is specified using our extended version of the UML 2.0 sequence diagram notation. It allows software developers to specify their expectations of the scenarios in which a certain program task will be performed. Automatically checking these expectations at runtime allows software developers to investigate software bugs based on scenarios, and thus, enables a scenario-driven debugging approach.

A behavior view diagram can be drawn using tools that support UML 2.0 sequence diagrams. In fact, the diagrams shown in this paper are drawn using Microsoft Visio with the UML 2.0 stencil. We have extended our implementation of the scenario-driven debugger to use the information translated from a behavior view diagram for execution monitoring (see [8] for the monitoring algorithm). In our future work, we will develop visualizers that can provide visual feedback when a behavior view diagram is used for execution monitoring. We will also perform case studies to evaluate the usefulness and the usability of our debugging technique.

References

- [1] M. Auguston, C. Jeffery, and S. Underwood. A framework for automatic debugging. Technical Report TR-CS-004/2002, New Mexico State University, 2002.
- [2] 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.
- [3] D. Cohen, M. Feather, K. Narayanaswamy, and S. Fickas. Automatic monitoring of software requirements. In *ICSE'97*, 1997.
- [4] M. Ducasse. Coca: An automated debugger for C. In *ICSE'99*, pages 504–513, May 1999.
- [5] L. Geiger and A. Zundorf. Graph based debugging with fujaba. *Electronic Notes on Theoretical Computer Science*, 72(2), 2002.
- [6] I. Jacobson, G.Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
- [7] D. Liang and K. Xu. Debugging object-oriented programs with behavior views. In *Sixth International Symposium on Automated and Analysis-Driven Debugging (to appear)*, 2005.
- [8] D. Liang and K. Xu. Monitoring with behavior view diagrams for scenario-driven debugging. Technical Report 05-025, Univ. of Minnesota, 2005.
- [9] 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.
- [10] OMG. UML 2.0 superstructure. <http://www.omg.org/cgi-bin/doc?ptc/2004-10-02>.
- [11] G. Spanoudakis and K. Mahub. Requirements monitoring for service-based systems: Towards a framework based on event calculus. In *19th IEEE International Conference on Automated Software Engineering*, pages 379–384, 2004.
- [12] J. Stanglewicz. Design-level debugging. *Real-Time Magazine*, (1):68–72, 1999.
- [13] C. J. Turner, T. N. Graham, C. Wolfe, J. Ball, D. Holman, H. D. Stewart, and A. G. Ryman. Visual constraint diagrams: Runtime conformance checking of UML object models versus implementations. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 271–276, 2003.