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 06-024

A Monitoring Profile for UML Sequence Diagrams

Kai Xu and Donglin Liang

July 24, 2006

A Monitoring Profile for UML Sequence Diagrams

Kai Xu and Donglin Liang

University of Minnesota, MN 55455, USA,
{kai,dliang}@cs.umn.edu

Abstract. UML sequence diagrams are widely used during requirements analysis and design for specifying the expected behaviors of a system. In this paper, we present a monitoring profile that extends sequence diagrams to facilitate the comparison between the actual behaviors and expected behaviors of a system. With the profile, the developers can precisely specify the runtime objects to be monitored, the expected sequences of message exchanges among these objects, and the monitoring actions to be performed when the message exchanges are observed. Supported by the tools that we develop, the monitoring actions can inspect program states, verify assertions, record coverage information, and visualize the computation during the progress of various scenarios. These tools allow software developers to effectively use their design knowledge to detect and localize bugs during testing and debugging.

1 Introduction

Software that we develop nowadays must implement increasingly complex behaviors. During the development of such a complex system, software developers often make mistakes in the requirement specification, design, and implementation. These mistakes introduce software bugs that often need to be detected through testing, and be identified and removed during debugging. Testing and debugging require software developers to exercise the system with appropriate inputs, monitor the program execution, and compare the observed behaviors of the software with the expected behaviors. A discrepancy between the observed and expected behaviors can indicate the existence of a software bug in the components of the system.

In modern software development methodologies (e.g., [5]), the expected behaviors for a system or a subsystem are often identified and documented as *scenarios* during requirements analysis and design. Therefore, testing and debugging at the system or subsystem level should focus on monitoring program actions and verifying properties relevant to the progress of these scenarios. This *scenario-based* monitoring 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 testing and debugging techniques provide inadequate support for the scenario-based monitoring. Assertions have been widely used in testing and

debugging to check whether the program behaves as intended (e.g., [2, 16]). However, because the assertions are often specified independent of the execution history, they are not suitable for specifying properties that are specific to a particular scenario, or properties that are related to several steps of object interactions for implementing the scenario. Existing debugging techniques provide various supports for execution monitoring. Source level debugging mechanisms, such as breakpoints, allow software developers to interactively inspect the program states when the program control reaches specific code locations. Event-based debugging techniques (e.g., [1, 3, 10]), on the other hand, allow the software developers to specify the inspections to be performed automatically when specific execution events occur. However, because these techniques do not emphasize on correlating the monitoring of the program actions at different points of time during execution, these techniques provide inadequate support for the observation and inspection of the progress of scenarios.

The goal of our research is to develop better techniques to support scenario-based execution monitoring. To achieve this goal, we propose behavior view diagrams to facilitate the monitoring and visualization of the scenario progress during the execution of an object-oriented system. A *behavior view diagram* (BVD) precisely specifies the set of runtime objects that may participate in the scenarios. It also specifies the sequences of message exchanges that characterize the progress of these scenarios. It may further specify the properties to be checked and the monitoring statements to be executed for inspecting the program states during the progress of the scenarios. Thus, it is a powerful mechanism for the software developers to compare their expectations of these scenarios with the actual behaviors of the system.

In previous work [8], we have presented an extension to UML 2.0 sequence diagram notation [11] for specifying BVDs. We have also investigated the use of BVDs for testing and debugging [8, 9]. In this paper, we present a UML monitoring profile that formally defines this extension. Defining our extension with a profile would allow BVD editing to be done with any editor that supports UML 2.0 sequence diagrams and profiles. We will also present the design of a monitoring engine and a compiler that takes an XMI (XML Metadata Interchange) representation of a BVD and compiles it into a representation that can be used by a monitoring engine. The compiler and the monitoring engine are the core components for a tool suite that supports the scenario-based execution monitoring during testing and debugging.

In the rest of the paper, Section 2 gives an overview of testing and debugging with BVDs. Section 3 presents the monitoring profile. Section 4 presents the monitoring engine and the BVD compiler. Section 5 discusses the related work. Section 6 concludes the paper and discusses future work.

2 Testing and Debugging with Behavior View Diagrams

This section briefly introduces the use of behavior view diagrams to facilitate testing and debugging.

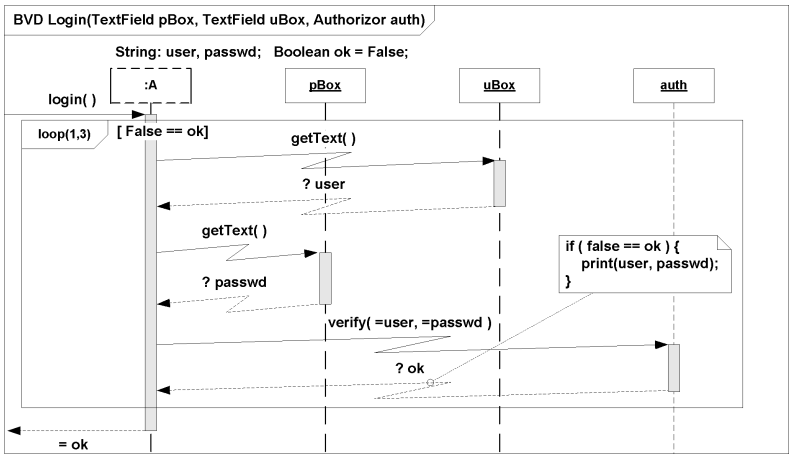

Fig. 1. A behavior view diagram.

2.1 An Overview of Behavior View Diagrams

In the previous work [8], we proposed an extension to UML 2.0 sequence diagram notations for specifying how the progress of a program task should be monitored. A sequence diagram uses a graphical notation to represent the possible sequences of message exchanges that may occur under various scenarios for performing a task.[1] This notation can also be used for specifying the message exchanges to be monitored during testing and debugging. We extend the syntax and the semantics of this notation to precisely specify the runtime objects whose behaviors will be monitored, and the monitoring actions to be performed when the message exchanges are detected. We refer to a diagram specified with this extended notation as a *behavior view* diagram (BVD).

Figure 1 shows a BVD that defines how a login task is expected to be performed in a GUI application. This BVD illustrates a subset of features that we introduce. In a BVD, life-lines have been extended to include information that specifies the runtime objects represented by these life-lines. In Figure 1, the first life-line is specified to represent the receiver object of method call `login()`; the other three life-lines are specified to represent the objects whose references are passed in through the three parameters `uBox`, `pBox`, and `auth`. The BVD also contains the declarations of three *monitoring* variables `user`, `passwd`, and `ok`. These monitoring variables can be used for storing information extracted from the program states during the monitoring.

In a BVD, the notations for messages have also been extended for supporting monitoring. A zigzag arrow in a BVD represents a possibly-indirect method call or return. This notation can be used to avoid specifying uninteresting objects on the call path between the sender and the receiver. In addition, the label of a message is extended with notations for asserting the expected values for the

[1] In literature, a scenario often refers to a sequence of interactions between a software system and its users. We extends this notion to include interactions among objects.

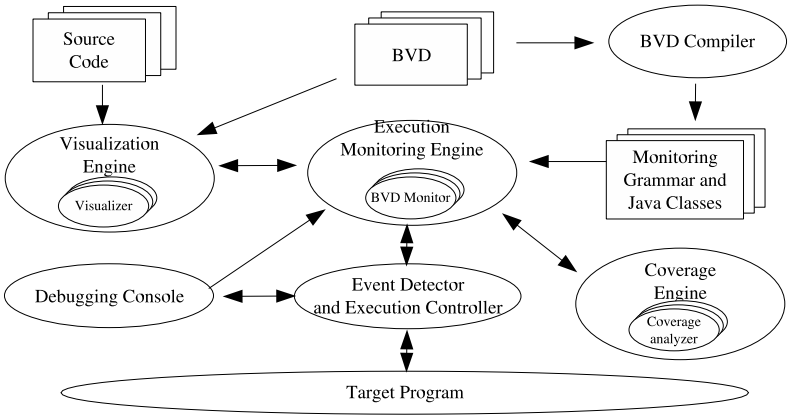

Fig. 2. A BVD-based testing and debugging tool suite.

parameters/return-values or extracting values from them. For example, label "`verify(=user,=passwd)`" in the figure specifies that the values of the two parameters for `verify()` are expected to be equal to the values of monitoring variables "user" and "passwd". For another example, label `?user` specifies that the return value of the first "getText()" call should be extracted and stored in "user". A message may also be associated with a *monitoring block* that contains regular Java statements describing the monitoring actions to be performed when an occurrence of this message is detected. For example, in Figure 1, the return message for `verify()` is associated with an `if` statement that checks the return value of `verify()` and provides visual feedback. By interacting with appropriate tools, a monitoring block can inspect program states, visualize the progress of the scenarios, or even record the scenario-based coverage for a test suites.

2.2 Tool Support for Testing and Debugging with BVDs

We have developed a tool suite to support the use of BVDs for testing and debugging Java programs. Figure 2 illustrates the architecture of our tool suite. The tool suite has been implemented on top of the Java Platform Debugger Architecture (JPDA). The core component of this tool suite is a monitoring engine that monitors the program execution based on BVDs. The monitoring engine interacts with the target program execution through an event-detector/execution-controller. To support this monitoring, the BVD must be compiled into a representation that can be understood by the monitoring engine.

The tool suite also includes other components to support interactive debugging and scenario-based test coverage analysis. It contains a debugging console that provides a powerful command language for managing the monitoring activities. It also contains a visualization engine to allow the software developers to control the program execution and to visualize the progress of the scenarios based on the graphical presentation of the BVD. For example, the software developers may set breakpoints on a particular message or to single-step through the

message exchanges specified on the BVD. This capability supports the design-level debugging approach that advocates "driving and monitoring the debugging process from a design model viewpoint" [13]. The tool suite further includes a coverage engine that can interact with the BVD monitor for measuring BVD-based coverage of a test suite. Such coverage can measure the thoroughness of a test suite in exercising the specified scenarios [9].

One advantage of our tool suite is that it allows the software developers to use BVDs for building a layer of high-level abstractions for behavior verification. These abstractions hide the details for interpreting the meanings of the low-level program constructs, and present direct interfaces for examining the progress of various scenarios. This enables developers to effectively use their design knowledge to guide testing and debugging. Another advantage is that it enables the reuse of design artifacts for testing and debugging. Sequence diagrams specified during design can be turned into BVDs by enhancing them with monitoring specific information. Using the monitoring profile, "monitor-able" sequence diagrams can be produced during design. This approach not only improve the efficiency of testing and debugging, but also add incentives for a software team to produce and maintain the design artifacts.

3 A Monitoring Profile for Sequence Diagram

This section presents a profile that formally defines our extension to UML 2.0 sequence diagrams for specifying BVDs. We refer to such a profile as a *monitoring* profile because its main goal is to support execution monitoring.

3.1 A Metamodel for Sequence Diagrams

A metamodel for sequence diagrams defines the syntax and the semantics of the elements in sequence diagrams. Figure 3 shows an important subset of these elements and their inter-relationship according to the UML 2.0 [11]. As shown in the figure, a sequence diagram is an `Interaction` that contains a set of life-lines, a set of messages, and an ordered set of interaction fragments. This ordered set of interaction fragments defines the sequences of message exchanges modelled by this diagram. An interaction fragment can be one of the following types. A message occurrence specification represents either the sending event or the receiving event of a message. A state invariant is a boolean expression the characterizes the state that an object must be in at a particular point of time. An execution specification is the thin box that specifies the duration of a method invocation. A combined fragment has one or more interaction operands, each of which contains a sequence of interaction fragments. Each operand has a guard that specifies the condition under which the sequences defined by the operand can be used to refine the combined fragment. An interaction use refers to another sequence diagram. It stands as a place holder for the sequences defined by the referred diagram. An interaction use may contain an ordered set of actual gates that are corresponding to the formal gates defined in the referred interaction.

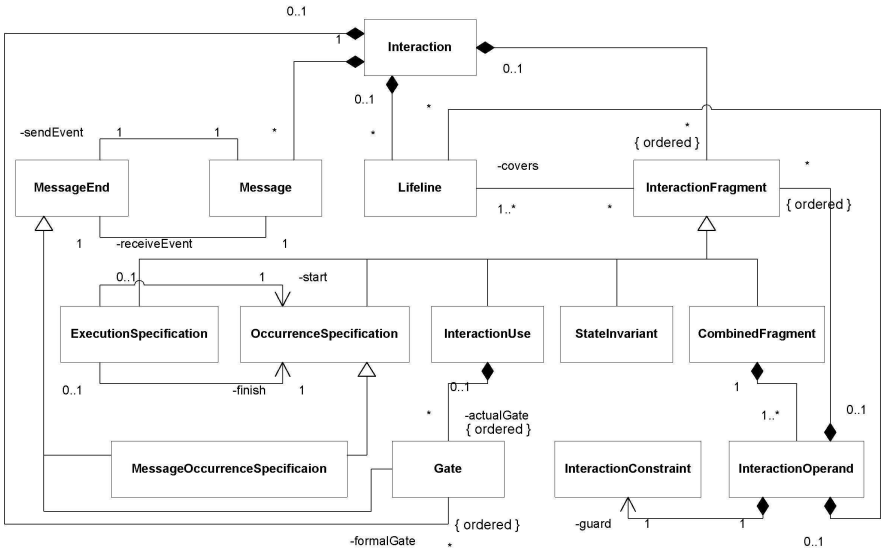

Fig. 3. A metamodel for sequence diagrams.

These gates serve as step-stones for connecting the sending/receiving events specified in different interactions.

Note that, in UML 2.0, all life-lines are owned by an interaction. However, this arrangement is not ideal: if a life-line is covered only by the interaction fragments within an interaction operand, this life-line should be local to the scope of this operand. This scoping can reduce the complexity of the diagram. Thus, in Figure 3, we extend the metamodel specified in UML 2.0 so that an interaction operand can contain local life-lines.

3.2 The Monitoring Profile

A profile defines how the elements for a diagram can be extended to support modelling in a specific domain. Figure 4 shows the monitoring profile that extends the elements of sequence diagrams for specifying BVDs. We have extended both `Interaction` and `InteractionOperand` with stereotypes for adding an extra property `localDec` that contains the declarations of local monitoring variables. We have extended `Message` with stereotype `Monitorable` for adding an extra property `Mblock` that specifies the monitoring block associated with the message. We have also extended `Message` with stereotype `Indirect` for specifying indirect messages. We have further defined three stereotypes for `LifeLine` to indicate the approaches for identifying the runtime object that will be represented by a life-line. We refer to these approaches as the *binding* approaches.

Stereotypes `PositionBased` and `ParameterBased` are used to annotate the life-lines whose bindings are determined using the values of formal parameters. There are two kinds of formal parameters in a BVD. *Explicit* parameters are declared following the name in the descriptor of the BVD. A life-line with

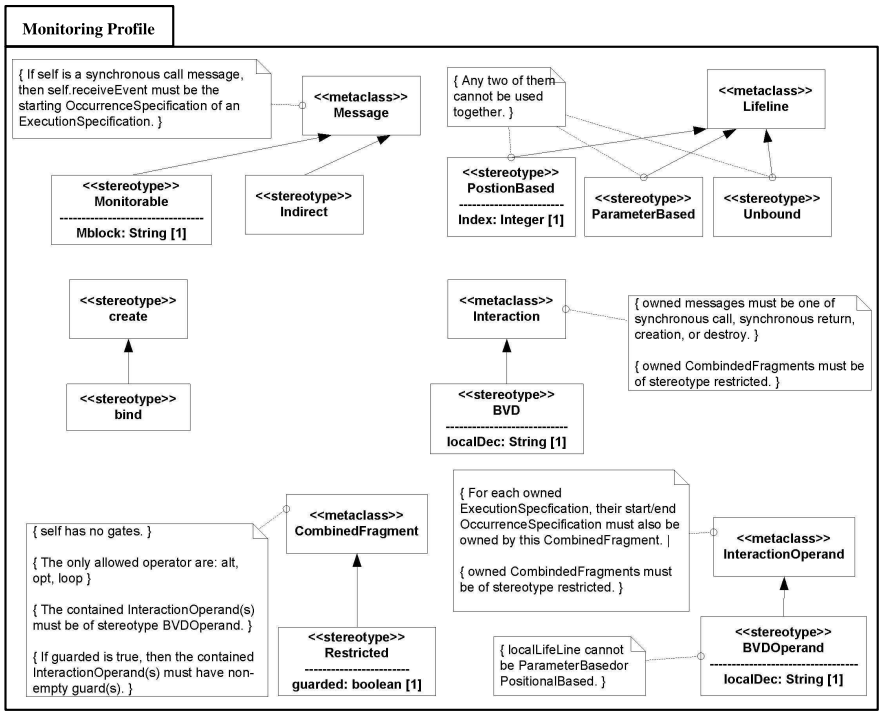

Fig. 4. The monitoring profile.

`ParameterBased` stereotype uses the value of an explicit parameter to determine the object that it represents. In this case, the name of the life-line must match the name of the parameter. *Implicit* parameters, however, are not explicitly declared. They are mainly used for passing the objects represented by the life-lines covered by an interaction use into the referred BVD. Within the referred BVD, these objects must be represented by life-lines with `PositionBased` stereotype. In this case, the bindings between the objects and the life-lines are based on the relative positions of the life-lines.

Stereotype `Unbound` is used to annotate life-lines whose bindings will be determined during the progress of the monitoring. The object represented by such a life-line is determined when the first message pointing to this life-line is detected. The first message can be a creation message or a call message. In both cases, the target object of the message will be bound to the life-line. To allow more flexible bindings, we also introduce a a special kind of message, the `binding` message, that is marked with stereotype `bind`. The label of a binding message has a binding expression whose value will be used as a reference to find the object to be bound to the target life-line. Syntactically, we consider a binding message as a special creation message.

The monitoring profile also specifies important constraints that have been imposed on an interaction and its elements to make it a BVD. Currently, a BVD is used for monitoring the behavior of a single thread specified in a Java-

like language. Therefore, a BVD contains only method call, return, or creation (including binding) messages; and a BVD contains only one *root* execution specification from which other execution specifications can be reached by tracing call message arrows. Furthermore, a combined fragment in a BVD must have stereotype `Restricted`. The operator for such a combined fragment can only be `alt`, `opt`, or `loop`. Other types of operators are excluded because they are either used for concurrency (e.g., `par`) or not well-defined (e.g., `neg`). A combined fragment in a BVD must be either *guarded* (i.e., all guards of the operands are not empty) or *unguarded* (i.e., all guards of the operands are empty). Unguarded combined fragments allow software developers to avoid modelling the conditions that control the program execution. The monitoring profile further specifies constraints for defining well-formed BVDs. For example, a combined fragment should not contain gates. The well-formedness is critical for the BVD compiler to work correctly.

3.3 An Operational Semantics for the Monitoring Profile

The major tasks of monitoring the execution of a program based on a BVD can be broken down as the following: compute the bindings for the life-lines, detect the occurrences of relevant events, match each event occurrence with a message, and perform the monitoring action associated with this message. These tasks are performed based on the progress of the execution and the instructions provided by the BVD. We will give an overview of an operational semantics for the BVD constructs by explaining how a monitoring engine can use these constructs.[2]

To allow a BVD to be used simultaneously for monitoring several situations, the monitoring engine creates a separate *BVD monitor* that captures the data structure for each use of the BVD. During the monitoring, a BVD monitor maintains a *data-frame record* for the BVD and each interaction operand under consideration. The data-frame record contains a slot for each local life-line or monitoring variable declared in the corresponding BVD/operand. A BVD monitor is created when a BVD is *deployed* by the debugging console [8] or a launch block instrumented in the target program [9]. In both cases, arguments must be provided for both the explicit parameters and the implicit parameters of the deployed BVD.

A BVD monitor iteratively performs the following four steps: (a) to search in the BVD for the description of the next possible event, (b) to wait for the occurrence of the next event, (c) to match a detected event with a message in the BVD, and (d) to perform the monitoring actions associated with this message. The ordered set of interaction fragments contained within a BVD defines the sequences of events that are expected to occur during program execution. Therefore, steps (a) and (c) are guided by this set. For a synchronous message, the sending and receiving events are often presented as a pair next to each other in the ordered set. Thus, the BVD monitor will consider this pair as one element.

[2] The semantics presented here is different from the standard semantics of UML 2.0 interactions (e.g., [11, 14]): the latter only defines the valid and invalid sequences of events described by an interaction.

The BVD monitor goes through the ordered set of interaction fragments during monitoring. It takes different actions depending on the type of the current element under consideration. If the current element is a state invariant, then the expression specified by this construct will be evaluated. If the expression is evaluated false, a fail signal will be raised. This signal will be handled appropriately depending on the current monitoring mode. If the current element is a pair of events for a binding message, then the binding expression will be evaluated, and the result will be used to find the object bound to the target life-line of the message. If the current element is a pair of events for a message of another type, then there are two cases. If the monitor is performing step (a), then it has found the next expected event and enters step (b). However, if the monitor is performing step (c), then the current element will be used to match the detected event. If the matching succeeds, then the monitor consumes the event and enters (d). If the matching fails, then a fail signal will be raised.

The monitoring becomes more sophisticated in the presence of interaction uses and combined fragments. If the current element under consideration is an interaction use, then the arguments specified on the interaction use will be evaluated, and a child BVD monitor will be created for the referred BVD and start monitoring the program execution. If the interaction use has actual gates, then the two BVD monitors will be coordinated appropriately for monitoring messages connected to these gates. If the current element under consideration is a guarded combined fragment, then the guards of its operands will be evaluated one by one.[3] If the guard of an operand is evaluated true, then the BVD monitor will continue on processing the ordered set of interaction fragments in this operand. If the current element under consideration is an unbounded combined fragment, then there are two cases. If the BVD monitor is performing step (a), it stops searching for the next possible event description because insufficient information is available at this time for identifying the target branch. In this case, it enters step (b). However, if the BVD monitor is performing step (c), it temporarily saves its state and enters an exploration mode. In this mode, the BVD monitor explores the operands one by one to see whether it can find a matched message specification. If the BVD monitor finds a successful match, it consumes the event and enters step (d). If the BVD monitor receives a fail signal during the exploration, it will try another operand. If no match is found after exploring all operands, it ignores the event and restores its state.

4 Execution monitoring with BVD

Directly dealing with the UML elements during monitoring is unnecessarily tedious. In our project, we follow a compilation approach to simplify the design of the monitoring engine: a BVD is first translated into an intermediate representation by a compiler; this representation is then used by the engine to perform the monitoring.

[3] To simplify the discussion, we assume that there is an extra empty operand for a combined fragment whose operator is `alt` or `loop`.

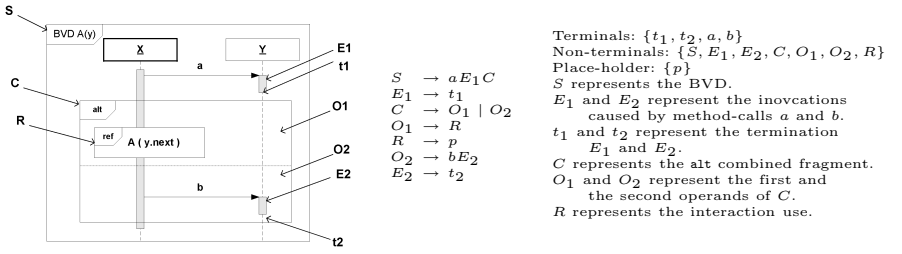

Fig. 5. An example monitoring grammar.

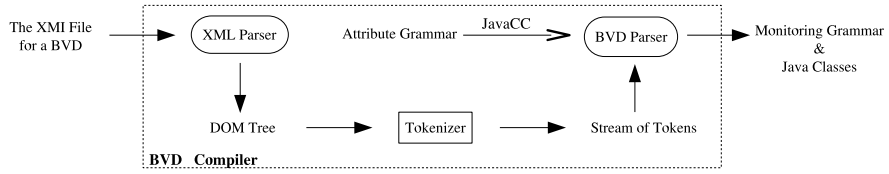

Fig. 6. The design of a BVD compiler.

In our previous work [17], we proposed an intermediate representation for BVD and a monitoring algorithm based on this representation. The execution monitoring is performed based on the sequence patterns of message exchanges defined by a BVD. We proposed to use a context free grammar to encode these patterns. Figure 5 illustrate an example of such a grammar. The grammar uses terminals to represent message exchanges defined by individual pairs of sending/receiving events. It uses non-terminals to represent the sequence patterns related to method invocations, combined fragments, and interaction operands. A production rule in the grammar specifies how the sequences represented by a non-terminal is composed from individual messages and the sub-sequences represented by other non-terminals. Given this representation, tracking the progress of the program execution can be viewed as a process of constructing, from the grammar, a derivation for the string of events that occur during the execution. We also proposed to represent the local-lines and monitoring variables of a BVD or an interaction operand as fields of a *data-frame* class, and to capture the monitoring actions defined by the monitoring blocks, state invariants, and guards contained in the BVD/operand as methods in this class. Given this representation, monitoring actions can be performed by invoking appropriate methods on a data-frame object.

This section presents the design of a BVD compiler that translates a BVD from its XMI representation into a monitoring grammar and a set of data-frame classes. A sequence diagram (and thus, a BVD) is often stored in an XMI (XML Metadata Interchange) [12] file. This structured text file serializes the information captured by the elements illustrated in the metamodel. As illustrated in Figure 6, we reuse existing tools to simplify the design of our compiler. The compiler first builds a DOM (Document Object Model) tree from the XMI file using an XML parser. The DOM tree is then converted into a stream of tokens

by a tokenizer that we developed. These tokens are parsed by a BVD parser generated by JavaCC [6] based on an attributed grammar that we define. The parser generates the monitoring grammar and the data-frame classes.

Our tokenizer performs a depth-first traversal on the DOM tree and converts each encountered XML element into a token that carries forwards the attributes defined in the element. For a leaf on the DOM tree, the tokenizer produces a token that captures the attributes of this leaf. For an internal node on the DOM tree, the tokenizer first produces a token that captures the attributes of this internal node. It then produces a **begin** token and starts traversing the children of this node. After all the children have been processed, the tokenizer produces an **end** token. This pair of **begin**/**end** delimits the tokens representing the inner elements of a composite UML element.

Tables 1 and 2 show an attributed grammar that we define for parsing the tokens generated by the tokenizer. The left side of Table 1 shows the set of productions for this grammar. In the grammar, we introduce non-terminals for recognizing the important chunks of information required for constructing the target grammar: we introduce non-terminals for recognizing the elements for individual message exchanges, non-terminal $Invocation$ for recognizing the UML elements related to a particular method invocation, and non-terminals for recognizing the elements related to combined fragments and their operands. Such an arrangement makes it easier to generate the monitoring grammar that organizes message exchanges specifically for each method invocation.

Generating monitoring grammar. The right side of Table 1 shows the semantic rules for generating the monitoring grammar from a BVD. These rules use a synthesized attribute "val" introduced for many non-terminals of the attributed grammar. The rules specify actions that generate the target grammar based on the following translation strategy. A message, a state invariant, an operand guard, and the end of an execution specification is translated into a unique terminal in the target grammar. An occurrence of $BVDSeq$ is translated into a start symbol for the target grammar. An occurrence of a $Invocation$, $Operand$, $CmbFrg$, or $interactionUse$ is translated into a unique non-terminal for the target grammar. An appropriate EBNF production rule is also generated for each of these non-terminals. Note that, the right side of the production rule for the non-terminal of $interactionUse$ is a place-holder symbol. When this place-holder is encountered during monitoring, it will be replaced by the starting symbol of monitoring grammar generated for the BVD referred to by the interaction use. We refer to this step as the *dynamic linking* step.

A terminal symbol created during the translation is also associated with a set of attributes that contain the necessary information for handling the symbol during monitoring. For example, a terminal for a call message is associated with information that characterizes the execution event that may match this terminal. This kind of associations is created by the "Crt*Terminal()" methods invoked by the semantics rules. A "Crt*Terminal()" method may need to access to the information of the life-lines and the messages stored in a parsing environment by the semantics rules for productions (3) and (5).

PRODUCTION	SEMANTICS RULES
1: BVD → **interaction begin** GtList LifelineList MsgList BVDSeq **end**	RegisterStartSymbol(BVD.id,BVDSeq.val)
2: LifelineList → ϵ	
3: LifelineList → **lifeline** LifelineList	StoreLifeline(lifeline)
4: MsgList → ϵ	
5: MsgList → **message** MsgList	StoreMessage(message)
6: BVDSeq → **startExec exec** FragList **endExec**	BVDSeq.val = S = CrtNonterminal() Output S → FragList.val
7: BVDSeq → **receiveEvt exec** FragList EndRootInv	BVDSeq.val = S = CrtNonterminal() m = CrtMsgTerminal(receiveEvt.message) X = CrtNonterminal() Output X → FragList.val EndRootInv.val Output S → m X
8: EndRootInv → **endExec**	EndRootInv.val = CrtEndTerminal(endExec)
9: EndRootInv → **sendRtMsg**	EndRootInv.val = CrtRtTerminal(sendRtMsg.message)
10: FragList → ϵ	FragList.val = ""
11: FragList → Fragment FragList	FragList_1.val = Fragment.val + FragList_2.val
12: Fragment → Msg	Fragment.val = Msg.val
13: Fragment → Msg Invocation	Fragment.val = Msg.val + Invocation.val
14: Fragment → CmbFrag	Fragment.val = CmbFrag.val
15: Fragment → **stateInv**	Fragment.val = CrtConstraintTerminal(stateInv)
16: Fragment → **interactionUse**	Fragment.val = X = CrtNonterminal() p = CrtPlaceHolder(interactionUse) Output X → p
17: Fragment → **sendEvt** GtIntUse	x = CrtMsgTerminal(sendEvt.message) Fragment.val = x + GtIntUse.val
18: CmbFrag → **alt begin** OperandList **end**	CmbFrag.val = X = CrtNonterminal() Output X → OperandList.val
19: CmbFrag → **opt begin** Operand **end**	CmbFrag.val = X = CrtNonterminal() Output X → Operand.val | ϵ
20: CmbFrag → **loop begin** Operand **end**	CmbFrag.val = X = CrtNonterminal() Output X → Operand.val *
21: OperandList → Operand	OperandList.val = Operand.val
22: OperandList → Operand OperandList	OperandList_1.val = Operand.val + "|" + OperandList_2.val
23: Operand → **operand begin guard** LifelineList FragList **end**	Operand.val = X = CrtNonterminal() g = CrtConstraintTerminal(guard) Output X → g FragList.val
24: Invocation → **exec** FragList EndInv	Invocation.val = X = CrtNonterminal() Output X → FragList.val EndInv.val
25: EndInv → **endExec**	EndInv.val = CrtEndTerminal(endExec)
26: EndInv → **sendRtMsg receiveEvt**	EndInv.val = CrtRtTerminal(sendRtMsg.message)
27: GtIntUse → **interactionUse begin** GtList **end** GtEnd	GtIntUse.val = X = CrtNonterminal() p = CrtPlaceHolder(interactionUse) Output X → p GtEnd.val
28: GtEnd → ϵ	GtEnd.val = ""
29: GtEnd → **receiveEvt**	GtEnd.val = CrtRtTerminal(receiveEvt.val)
30: Msg → **sendEvt receiveEvt**	Msg.val = CrtMsgTerminal(sendEvt.message)
31: GtList → ϵ	
32: GtList → **gate** GtList	

Table 1. An attributed grammar for generating the monitoring grammar.

The semantics rules generates one monitoring grammar for each BVD. This grammar is slightly different from the one that we proposed in [17] where separate monitoring grammars are created for different execution specifications. Since then, we have improved the monitoring engine so that it can perform the monitoring based on one unified monitoring grammar for a BVD.

Generating data-frame classes. The right side of Table 2 shows the important semantics rules that generate the data-frame classes for a BVD or the interaction

PRODUCTION	SEMANTICS RULES
1: BVD → **interaction begin** GtList LifelineList MsgList BVDSeq **end**	BVD.code = CrtClassDef(interaction.name, interaction.dec + LifelineList.code + BVDSeq.code)
3: LifelineList → **lifeline** LifelineList	LifelineList_1.code = CrtLflnDec(lifeline) + LifelineList_2.code
7: BVDSeq → **receiveEvt exec** FragList EndRootInv	BVDSeq.code = CrtMonitorMethod(receiveEvt.message)
9: EndRootInv → **sendRtMsg**	EndRootInv.code = CrtMonitorMethod(sendRtMsg.message)
15: Fragment → **stateInv**	Fragment.code = CrtBoolMethod(stateInv)
17: Fragment → **sendEvt** GtIntUse	Fragment.code = CrtMonitorMethod(sendEvt.message) + GtIntUse.code
23: Operand → **operand begin guard** LifelineList FragList **end**	gMethod = CrtBoolMethod(guard) Operand.code = CrtClassDef(operand.id, gMethod + opeand.dec + LifelineList.code + FragList.code)
26: EndInv → **sendRtMsg receiveEvt**	EndInv.code = CrtMonitorMethod(sendRtMsg.message)
29: GtEnd → **receiveEvt**	GtEnd.code = CrtMonitorMethod(receiveEvt.message)
30: Msg → **sendEvt receiveEvt**	Msg.code = CrtMonitorMethod(sendEvt.message)

Table 2. Generating data-frame classes.

operands contained in the BVD. These rules use a synthesized attribute "code" for each non-terminal in the attributed grammar. Note that some productions of the attributed grammar are not shown in the table. For such a production, the default semantics rule is to concatenate the values of the "code" attributes of the non-terminals at the right side and assign the result (or the empty string if the right side does not contain any non-terminal) to the "code" attribute of the non-terminal at the left side.

A data-frame class will be created for a BVD and each interaction operand contained in the BVD. Each class contains field declarations generated from the local life-lines and the local monitoring variables. The class also contains methods that are generated to capture the monitoring actions defined by a guard, a state invariant, the value checking/extraction in a message label, and the monitoring block associated with a message. To allow the monitoring actions specified in the operands of a nested combined fragment to access the life-lines and monitoring variables declared in a containing BVD or interaction operand, the data-frame classes for these operands are declared as inner classes to the class created for the containing BVD/operand. To allow a method that captures the monitoring action associated with a UML element to be invoked during the monitoring, the semantics rule will also generate a mapping between the terminal that represents this element in the monitoring grammar and the name of this method.

5 Related Work

In spirit, a BVD is quite similar to a visual constraint diagram [15]: 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 BVD is used for detecting the interaction anomalies. It extends a UML sequence diagram. It should be possible to use them together for more effective monitoring.

Live sequence charts (LSCs), an extended form of message sequence chart, can be executed by a play engine for simulating the behaviors of a reactive system [4]. This notation can also specify the message exchanges that can be observed during the execution of a system. Therefore, presumedly, one should be able to use LSCs for specifying and checking scenario-specific properties. However, there is a paradigm difference between LSCs and BVDs. LSCs follow a rule-based composition paradigm: different LSCs are specified as relatively independent rules, and are implicitly composed together by event production/consumption. In contrast, BVDs follow a call-based composition paradigm: they are composed explicitly through reference uses. This paradigm allows a hierarchical way for specifying complex scenarios. In addition, because testing and debugging manually-constructed OO programs is not the design goal of the play engine, it does not provide the same level of support for this task as our tool (e.g., the value-based binding, monitoring blocks, the flexible deployment).

Kiviluoma et. al. [7] have proposed to use UML sequence diagrams for specifying the architecturally significant behaviors. They have also proposed an approach that automatically generates monitoring code from this specification. Their framework is different from our tool suite in several ways. First, their framework can only check whether the actual method calls among different objects follow a predefined sequencing order. In contrast, our tool suite also allows to perform various monitoring actions when a message exchange is detected. This broader monitoring capability is important for detecting and locating bugs manifested by incorrect states. Second, their framework automatically applies the monitoring to all objects of the classes that match the role description. In contrast, our tool suite allows software developers to precisely specify the runtime objects to be monitored and the period of time in which the monitoring should be deployed. This precision is important for investigating a specific bug in a program. In addition, due to the lack of details, it is not clear how their framework can handle UML 2.0 features such as the guarded and unguarded combined fragments and (potentially recursive) reference uses.

6 Conclusion

This paper presents a monitoring profile for UML 2.0 sequence diagrams. This profile can be used by software developers to specify execution monitoring based on their understandings of the scenarios for performing various program tasks. The paper also presents a compiler that can translate a diagram specified with the profile into a representation that can be understood by a monitoring engine that we have developed. With this engine and the other supporting tools, software developers can effectively use their design knowledge to guide the detection and the investigation of bugs during program execution. Initial experience [17]

shows that this capability has the potential to improve both the effectiveness and the efficiency of testing and debugging.

UML 2.0 has introduced many features that increase both the expression power and the complexity of the sequence diagram notation. Our tool suite considers only a subset of these features. Such a subset should allow software developers to specify many realistic program behaviors. In our future work, we will consider other features. Our initial assessment shows that few existing UML tools can provide the desired support for editing BVDs. In the future, we will continue evaluating the UML tools. Meanwhile, we are developing an editor for editing BVDs. We are also integrating our tool suite into Eclipse and continue evaluating and improving its usability. Once the tool suite becomes stable, we will make it available to public.

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. R. Binder. *Testing Object-Oriented Systems.* Addison-Wesley Professional, 1999.
3. M. Ducasse. Coca: An automated debugger for C. In *ICSE'99*, pages 504–513, May 1999.
4. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Springer, 2003.
5. I. Jacobson, G.Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.
6. JavaCC. https://javacc.dev.java.net/.
7. K. Kiviluoma, J. Koskinen, and T. Mikkonen. Run-time monitoring of behavioral profiles with aspects. In *The 3rd Nordic Workshop on UML and Software Modeling*, 2005.
8. D. Liang and K. Xu. Monitoring with behavior view diagrams for scenario-driven debugging. In *IEEE Asia-Pacific Software Engineering Conference*, Dec. 2005.
9. D. Liang and K. Xu. Testing scenario implementation with behavior contracts. Technical report, Univ. of Minnesota, 2006.
10. 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.
11. OMG. UML 2.0 superstructure. http://www.omg.org/cgi-bin/doc?ptc/2004-10-02.
12. OMG. Mof 2.0 / xmi mapping specification, v2.1. http://www.omg.org/technology/documents/formal/xmi.htm, 2005.
13. J. Stanglewicz. Design-level debugging. *Real-Time Magazine*, (1):68–72, 1999.
14. H. Storrle. Semantics of interactions in uml 2.0. In *International Symp. Visual Languages and Formal Methods*, 2003.
15. 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.
16. J. Voas and L. Kassab. Using assertions to make untestable software more testable. *Software Quality Professional Journal*, 1(4), 1999.
17. K. Xu and D. Liang. Supporting scenario-driven debugging with behavior view diagrams. Technical Report 06-002, Dept. of CSE, Univ. of Minnesota, 2006.