Testing Scenario Implementation with Behavior Contracts

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

Abstract

This paper presents behavior contracts as a new assertion mechanism and a tool that uses such contracts to support the testing of Object-Oriented (OO) systems. A behavior contract models how the scenarios for performing a designated task are expected to be implemented. Based on this contract, our tool can automatically monitor the program execution for checking important properties related to these scenarios. This capability can help testers to determine whether the scenarios have been implemented correctly. Our tool can also collect test coverage information w.r.t. the scenarios modeled by a behavior contract. Such information can be used to direct the testing efforts towards the less-covered scenarios, and to determine whether the implementation of these scenarios has been adequately tested. Therefore, using this tool should improve both the efficiency and effectiveness of testing OO systems.

1 Introduction

Software testing is about exercising the implementation under test (IUT) appropriately in order to detect bugs in the implementation. Ideally, the software developers should be able to completely specify, with some sort of formal notations, the functionalities of the IUT. This formal specification can then be used to check the output of each test run to determine whether a test passes or fails. Unfortunately, for a reasonably complex implementation, it is often impractical to come up with such a specification. To deal with this situation in practice, software developers often rely on assertions that are evaluated at various points of time during execution for checking the correctness of the IUT [2]. A test is considered passed if none of these assertions are violated.

Validating test runs with assertions is a sampling technique. The effectiveness of this technique depends on what properties will be asserted and when the assertions will be evaluated. Design-by-contract [11] is proposed as a systematic approach for identifying and placing assertions. This approach encourages software developers to identify and specify *class contracts* in the forms of method pre-/post-conditions and class invariants that can be checked during program execution. These assertions document the assumptions that the developers make at the boundaries of different modules. Thus, they can be effective in detecting errors caused by misunderstandings across module boundaries.

The class contracts used in Design-by-Contract have limitations in supporting the testing of the implementation for behavior scenarios. This kind of testing is typically required at the subsystem and the application levels [2]. In a modern OO software development methodology [6], the expected functionalities of a system are often identified as scenarios; and these scenarios will be implemented by a set of interacting objects. To test the implementation for these scenarios, it is important to track the progress of the object interactions for each specific scenario and check that (a) the right objects have been participating in the scenario, (b) the actions performed by these objects at various steps are in the right order, and (c) these actions have the expected effect on the program states under the context of this scenario. Because class contracts are specified within class contexts, it would be difficult to use such contracts for specifying properties specific to individual scenarios; it would also be difficult to use such contracts for specifying properties related to several steps of object interactions. Therefore, it may be challenging to use class contracts for asserting how the scenarios are expected to progress.

In this paper, we propose a new form of contracts, the *software behavior contracts* (or behavior contracts in short), that facilitate the testing of scenario implementation in an OO system. A behavior contract specifies how various scenarios are expected to progress for performing a particular task during the program execution. This contract specifies the milestones that mark the progress of these scenarios and the expected sequencing order of these milestones. The contract also specifies the objects that are expected to participate in the scenarios. The contract further specifies the important properties that must hold at these milestones. With a tool that we develop, such a contract can be used to compare with the actual behaviors observed at runtime for detecting bugs in the implementation for these scenarios.

With our tool, such a contract can also be used to determine whether the important scenarios and properties have been adequately exercised by a test suite.

In the rest of the paper, Section 2 discusses the behavior contracts. Section 3 presents our notation for specifying such contracts. Section 4 discusses a testing strategy with behavior contracts, and section 5 presents a tool that supports this testing strategy. Section 6 discusses related work. Section 7 concludes and discusses future work.

2 Software Behavior Contracts

Design-by-contract [11] takes the metaphor of a contract in our society and applies it in the software engineering world. The contracts advocated by the Design-by-contract typically involves a supplier (or server) and its clients. A contract is typically written from a server's perspective. The contract includes pre-conditions that characterize the proper state must be set up by a client before the client can invoke a method of a server. The contract also includes post-conditions that specify the properties that must hold when the method returns. The contract may further include class invariants that specify properties that must be true within the scope of the class. These contracts are often referred to as *class contracts* because they must be followed by all the instances of a class.

Class contracts have limitations for asserting the properties about the expected behaviors that will be implemented by a system. In modern software development methodologies (e.g., [6]), the expected software behaviors are typically identified and specified as scenarios at the requirements and design levels. A *scenario* is identified as a sequence of interactions among the components of the system for performing a particular task.[1] Such a scenario is typically implemented by a sequence of interactions among various objects. To ensure the implementation is correct, it is important to ensure that the interactions for this scenario progress as intended. However, because the pre-/post-conditions and class invariants are specified from a class perspective, properties related to such a progression may be difficult to express using these mechanisms.

We propose *software behavior contracts* as a new abstraction for specifying properties specific to scenarios. In our society, a *behavior contract* is often used to curb the behavior problems of students in a school environment [8]. Such a contract often states how the students should behave and interact with one another and the teachers in various situations in order to maintain a good learning environment.[2] In analogy, a *software behavior contract* is used to deal with the behavior problems of objects in a software system. It specifies how objects should behave and interact with one another under the contexts of individual scenarios in order to correctly perform a designated task.

Any behavioral artifact (e.g., a UML sequence diagram) generated during analysis and design phases can be viewed as a form of behavior contract. These artifacts can guide the programmers in implementing the objects in the system. However, in their current forms, these artifacts in general cannot be automatically checked at runtime to determine whether the contracts have been followed. First, the notation for specifying these artifacts often does not have a formal semantics. Second, these artifacts may not contain enough information for mapping the elements in the model to the runtime entities. Thus, they cannot be directly used to compare with the actual behaviors observed at runtime.

We propose to extend existing behavior modeling notations for writing *runtime-checkable* behavior contracts. We have extended UML 2.0 sequence diagram notation for this purpose. A sequence diagram is a graphical notation for specifying the possible sequences of message exchanges among a set of objects for achieving a certain goal. It has been widely adopted in practice for modeling behavior scenarios. We introduced a monitoring semantics for this notation. We also extended this notation with constructs that provide information for identifying the runtime objects whose interactions are being modeled by a diagram. We further extended this notation with constructs for specifying important properties that must hold at various points of time during the progress of the object interactions. We refer to a diagram specified with the extended notation as a *behavior view diagram* (BVD).

Given a BVD, an execution monitor can be created at runtime to track the progress of the modeled scenarios. The monitor can identify the runtime objects based on the information provided by the BVD, detect the occurrences of the message exchanges specified in the BVD, check the sequencing order among these occurrences, and check the properties that are associated with these messages. Therefore, it can detect bugs in the implementation of the modeled interactions.

Example. Figure 1 shows a BVD that defines how the scenarios for a login task are expected to progress. The login task is expected to start when method `login()` is called on any object of class `A`. During the invocation of this method, `getText()` is 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

[1] In literature, a scenario sometimes is only used to refer to a sequence of interactions between a software system and its users identified during requirements analysis. Our definition extends this notation to include the object interactions identified during design.

[2] Similar contracts (e.g., code of conduct) are in place in many organizations for regulating the behaviors of its members.

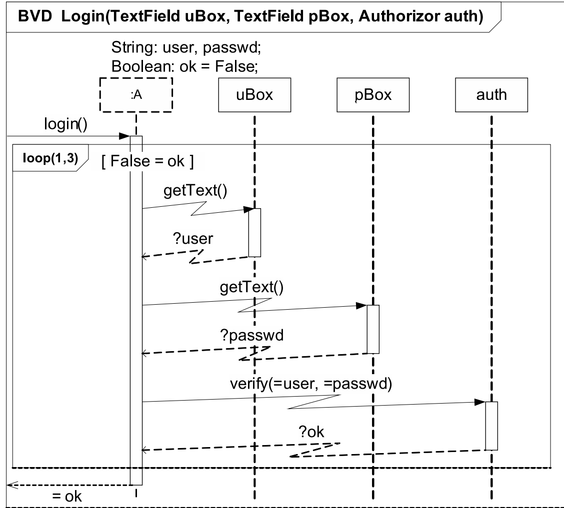

Figure 1. A behavior view diagram.

are valid. If they are invalid, then these steps can repeat up to three times. Otherwise, `login()` terminates.

In addition to the sequence of message exchanges, the BVD in Figure 1 also contains three monitoring variables, `ok`, `user`, and `passwd`. Variables `user` and `passwd` get their values at the returns of the calls to `getText()` on the two text-boxes, respectively. These variables are used to assert that the two parameters passed into `verify()` are indeed the user id and the password that the user had put into the two text-boxes. Variable `ok` gets it value at the return of the call to `verify()`. This value is used to assert that (a) the loop repeats only if `verify()` returns false, and (b) the returned value of `login()` must match the value returned by the last call to `verify()`.

Discussion. As shown in Figure 1, a behavior contract specifies properties with respect to the interaction of objects. These properties closely reflect the designer's expectations of the implementation for a set of scenarios. These properties cannot be easily mapped to class contracts in the forms of method pre-/post-conditions or class invariants. For example, it is not straightforwards for writing class contracts to check that `verify()` is indeed invoked with the strings input through the two text-boxes. Even if the mapping is possible, the class contracts may look very different from what the designers have in mind. Therefore, it may be difficult for the software designers to determine whether the class contracts correctly specify the expected behaviors.

As mentioned in the introduction, validating test runs with assertions is a sampling technique. Therefore, the behavior contracts alone may not be able to detect all faults. Further investigation is needed to study how behavior contracts can be used together with class contracts and other forms of assertions to achieve a better result.

3 Behavior View Diagrams

The behavior view diagram (BVD) notation was first introduced in our previous work [9] for assisting the debugging of object-oriented programs. In this paper, we adopt this notation for specifying behavior contracts. The BVD notation extends UML 2.0 notation for sequence diagrams. In a sequence diagram, an object is represented by a *life-line* that is covered by a sequence of vertical thin boxes representing the *execution specifications* of method invocations. Each execution specification is associated with a sequence of message arrows that represent message exchanges among objects. The current version of BVD specifies object interactions that occur within a single thread. Therefore, a BVD contains only object-creation, method call, and method return messages. We extend UML 2.0 with indirect messages (e.g., the zigzagged arrows in Figure 1) so that the software developers can avoid specifying uninteresting objects on the call path between the sender and the receiver.

A message arrow in a sequence diagram is typically marked with a label that describes the represented messages. We extend this notation to assert the expected values for the parameters or the return value of a method call. For example, label "`verify(=user,=passwd)`" in Figure 1 specifies that the values of the two parameters of `verify` are expected to equal to the values of the monitoring variables `user` and `password`, respectively. We also extend the label with notations for extracting the values from the parameters or the return value of a method call into monitoring variables. For example, in Figure 1, label "`?ok`" specifies that the return value of the call to `verify()` should be stored in `ok`.

A BVD extends a sequence diagram with binding information that specifies how the runtime object represented by a life-line should be identified. In Figure 1, the objects represented by the three life-lines on the right are provided as parameters (`uBox`, `pBox`, and `auth`) to the diagram. This kind of binding is referred to as the *parameter-based* binding. The object represented by the first life-line is discovered as the receiver object of `login()`. This kind of binding is referred to as the *invocation-implied* binding. There are three other kinds of bindings. A *position-based* binding is a special type of parameter-based binding. A *creation-implied* binding is derived from a creation message: the target life-line of this message is bound to the created object. A *value-based* binding is specified explicitly as a dashed arrow from one life-line to the head of another life-line (see Figure 2). This arrow is labelled with a *binding* expression whose value yields a reference to the object that will be bound to the target life-line of the arrow. These binding mechanisms provide powerful tools for identifying runtime objects to be monitored.

Inherited from the UML 2.0 notation, our notation for

BVD allows sequences of messages to be grouped into interaction fragments that enable modular specification of complex object interactions. Each interaction fragment is contained in a rectangle frame with a descriptor at the top-left corner. The area within the rectangle frame may be divided into swimming lanes (called *operands*). An operand defines a scope for local life-lines and local *monitoring* variables (e.g. `user` and `passwd` in Figure 1).

A stand-alone interaction fragment defines a BVD. A BVD may be called by another BVD through an *interaction use* construct (a rectangle tagged with **ref**) that contains the name and a list of actual parameters for the called BVD. An interaction use construct is a placeholder for the message sequences defined by the called BVD. The construct may cover one or more life-lines of the containing BVD. The bindings of the covered life-lines will be passed as implicit parameters to the called BVD. These implicit parameters will be bound to the life-lines in the called BVD based on their relative positions (*position-based* bindings).

An interaction fragment may be nested within another fragment to define message sequences that can be composed into the nesting fragment.[3] The composition is controlled by the descriptor of the fragment. In a BVD, the descriptor is limited to `loop`, `opt`, or `alt`. An operand in a nested fragment can be associated with a *guard condition* (specified as a predicate enclosed within a pair of brackets). Such a condition controls whether the sequence of messages defined by the operand is expected to occur.

A BVD also extends a sequence diagram with *monitoring blocks* for specifying more sophisticated assertions. A monitoring-block is a special UML comment construct associated with a message. This block contains a block of Java statements that can access monitoring variables, and may access program states through a reflection mechanism. These statements can perform computation necessary for checking complex properties. When a BVD is used for monitoring the object interactions during execution, the statements in a monitoring block will be executed when an occurrence of the associated message is detected.

Figure 2 shows a BVD for a more sophisticated behavior: the insertion to a B-tree. To perform the insertion, method `BNode.insert()` will be called on the nodes of a B-tree to find an appropriate leaf for inserting the key. The BVD specifies that, when `BNode.insert()` encounters a leaf node, `insertKey()` is expected to be invoked to insert the key to this node. However, when `BNode.insert()` encounters a non-leaf node, `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 this child is expected to be split into two nodes. This splitting is described in another behavior

[3] UML 2.0 refers to nested fragments as combined fragments.

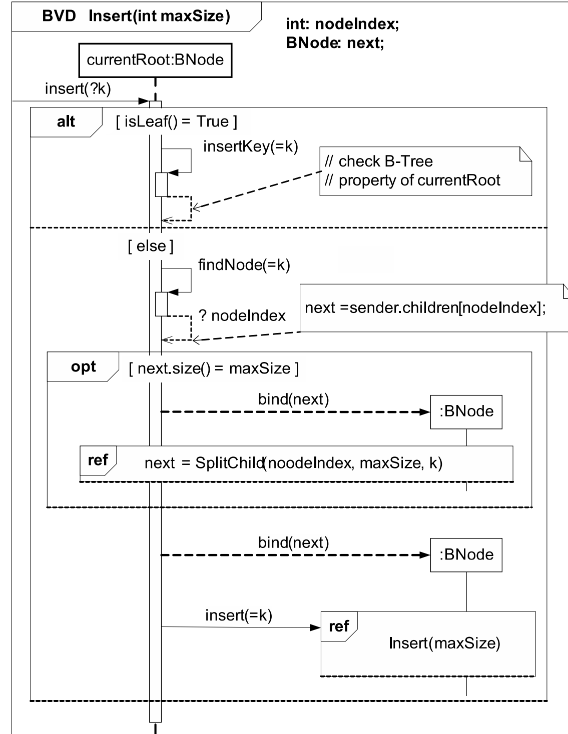

Figure 2. BVD for the insertion to a B-tree.

view diagram, `SplitChild`, which is not described here due to space limitation. After that, `BNode.insert()` is expected to be invoked on the identified child. This behavior is specified by a recursive reference use. In this example, the value-based binding enables us to identify runtime objects that must be discovered as the scenario progresses. A recursive reference use to the BVD itself enables us to specify properties over the interactions among a varied set of objects. These capabilities are important for practical use.

4 Testing with Behavior Contracts

This section discusses a methodology for using behavior contracts in testing Object-Oriented systems.

4.1 Result-Oriented Testing Strategy

"Effective testing must show that the component parts of the implementation under test (IUT) are sufficiently operable and must exercise the observable behavior of IUT." [2]. Based on this perspective, Binder (Chapter nine in [2]) proposed a result-oriented strategy for testing object-oriented software. This testing strategy orchestrates responsibility-based testing, integration testing, and implementation-based testing techniques for effectiveness. The testing activities will be organized based on the method, class, cluster, subsystem, and application system scopes.

The result-oriented testing strategy requires testing to be performed incrementally from the lower-level scopes to higher-level scopes. To perform testing at a certain scope, the parts of this scope must have been adequately tested at a previous step. Therefore, testing at the current scope can focus on the responsibilities of the IUT as a whole, not on its parts. Before applying any responsibility-based testing technique on the IUT, integration testing techniques must be applied on the parts to demonstrate a minimal interoperability among these parts. After sufficient responsibility-based testing has been applied to the IUT, implementation-based testing technique will be used to evaluate the coverage of the part interfaces and to enhance the existing test cases for better coverage.

The result-oriented testing strategy claims at least the following practical benefits. First, the three different kinds of testing techniques have been assigned clear roles to better leverage the relative strength of the different techniques for effective testing. Second, focusing on the responsibilities of the IUT during testing makes it easier for deriving test inputs and for interpreting the execution results of the test run. Third, incrementally applying testing to the scopes brings management advantages. In this case, testing of a code module (e.g., a method, a class) can be started once the implementation of this module is completed; and bugs that are specific to a particular scope may be caught during the testing at this scope. Therefore, bugs can be detected and fixed as early as possible.

4.2 Behavior Contracts as Responsibility Specifications

Responsibility-based testing is the major focus of the result-oriented testing strategy. To perform responsibility-based testing, the testers must first identify the responsibilities implemented by the IUT. The testers must then derive and run test cases for exercising these responsibilities. The testers also must determine whether the test runs are successful. The testers must finally determine whether the IUT has been adequately tested.

Both class contracts and behavior contracts can be viewed as responsibility specifications. These contracts can be used to guide the testers for deriving responsibility-based test cases. Because these contracts are runtime-checkable, they can be used to determine whether the test runs are successful. Pre-/post-conditions and class invariants specify the responsibilities of the methods in a class. Therefore, they are useful for testing at the method scope and at the class scope. Behavior contracts, in contrast, specify the responsibilities in terms of object interactions. Therefore, they may be more suitable for testing at the subsystem scope and the application scope.

A behavior contract often models more than one scenario. In this case, the testers may require these scenarios to be exercised appropriately by the test cases. Therefore, a behavior contract can be used as a coverage model for measuring the adequacy of a test suite. For example, based on the behavior contract specified in Figure 2, we may require the system to be tested such that 0, 1, or 2 nodes have been split in individual test runs. A contract-based coverage criterion ensures that the important responsibilities of the IUT have been adequately demonstrated during testing. Such an assurance cannot be obtained by test suites that satisfy only implementation-based coverage criteria. Therefore, contract-based coverage criteria complement implementation-based coverage criteria for determining the adequacy of test suites.

5 Tool Support for Using Behavior Contracts

This section briefly introduces a tool that we develop to support the use of behavior contracts for validating test runs and for measuring test coverage for Java programs.

5.1 Deploying Behavior Contracts

The behavior contract specified by a BVD should be deployed under the contexts in which the scenarios that it models are expected. Our tool supports two approaches for specifying the deployment of a behavior contract: an embedded approach and an Aspect-Oriented approach. In the embedded approach, the deployment is specified by a *launch block* placed in the source code as special comments. These special comments will be translated by our tool into Java code that will be inserted into the program to support testing. The following shows a launch block that deploys the BVD in Figure 2 for monitoring each invocation of `BTreeContainer.insertKey()`.

```
class BTreeContainer {
   ... //other fields and methods
   final int MAXSIZE = 100;
   private BNode root;
   /*@launch {{
    *    deploy Insert(MAXSIZE,root);
    * }} */
   public BNode insertKey(int key) {
      ...
   }
} // end of BTreeContainer
```

In the example, the launch block is defined as a special comment "@launch {{ ... }}". The block contains a deployment statement (starting with keyword `deploy`) that extracts value from field `MAXSIZE` for the formal parameter of BVD `Insert`, and extracts value from field `root` to provide the position-based binding for life-line `currentRoot` in BVD `Insert`. Note that, in general, a launch block can also contain regular Java statements for

defining more sophisticated deployment strategy. The statements in a launch block can access program variables visible at the location where the launch block is placed. These statements can also access object methods and object fields. Therefore, these statements can examine the current program state to determine whether a particular behavior contract should be deployed.

A launch block defines *launchers* that manage the deployed BVDs. The life-time of a launcher depends on where the defining launch block is placed. A launch block can be placed within a method body. In this case, a launcher will be created when the control-flow reaches the beginning of this launch block. The created launcher is like a local variable: it will be destroyed when the control-flow reaches the end of the Java block that immediately contains the launch block. A launch block can also be placed immediately before a method head. This is equivalent to placing the block at the beginning of the body of the method. In the rest of the paper, we will not distinguish these two cases.

The statements in the launch block will be executed when a launcher is being instantiated. When a deployment statement is encountered, a monitor will be created and the created monitor will be able to track the progress of object interactions based on the deployed BVD. This monitor can be destroyed when the method invocation represented by the root execution specification of the deployed BVD terminates. The monitor can also be destroyed when the launcher is destroyed unless the monitor is created by a *global* deployment statement that starts with keyword `::deploy`. When a monitor is destroyed by a launcher, if the monitor is still pending on some execution event, then the scenarios under monitoring have not reached an expected end. An error will be reported in this case.

In the Aspect-Oriented approach, the deployment scripts are written as aspects in an Aspect-Oriented Programming (AOP) language. The following example shows an aspect for deploying the BVD in Figure 2.

```
aspect LaunchBVDInsert {
  private Deployer deployer;
  pointcut insertPC(BTreeContainer bt) : this(bt) &&
    (execution(public * BTreeContainer.insertKey(..));
  before(BTreeContainer bt): insertPC(bt) {
    Object[] arguments = new Object[2];
    arguments[0] = new Integer(bt.getMaxSize());
    arguments[1] = bt.getRootNode();
    deployer = new Deployer("Insert",arguments);
  }
  after(BTreeContainer bt): insertPC(bt) {
    deployer.destroy();
    deployer=null;
  }
}
```

The aspect is written in AspectJ, an AOP extension to Java. It defines a *pointcut* `insertPC()` that identifies the invocation of `BTreeContainer.insertKey()`. It contains a *before advice* that will be invoked before the body of `insertKey()` is executed. The before advice extracts values from the fields of the receiver object of this invocation and creates a `Deployer` object that contains an execution monitor for BVD `Insert`. The aspect also contains an *after advice* that will be invoked after the body of `insertKey()` terminates. The after advice destroys the monitor created for the BVD. This aspect achieves the same effect as the launch block inserted before `insertKey()` in the previous example, under the assumption that this method cannot lead to a recursive call to itself.

The two deployment approaches provide different tradeoffs. The embedded approach allows the deployment to be inserted at almost any location in the method body; it also allows the use of the block structures in the method body for controlling the life-time of the monitors. However, this approach is intrusive; it requires that the source code is available at least for the classes where the deployment will be placed. In contrast, the AOP approach is not intrusive; it does not require the source code to be available. However, it can only deploy the behavior contracts at the well-defined places like method entry or method exit. Thus, it is less flexible than the embedded approach.

5.2 Monitoring with Behavior Contracts

After a BVD is deployed, an execution monitor must be created by our tool for monitoring the interactions of objects according to the BVD. Based on the BVD, the monitor automatically identifies and detects relevant events during the program execution, performs appropriate monitoring actions at the designated points of time, and properly maintains the data environment required for identifying events and for executing the monitoring actions. We have developed an algorithm for performing these tasks [15].

The execution monitors created from BVDs are supported by an event-subscription subsystem. The event-subscription system takes event descriptions from the execution monitors, detects the occurrences of events that match these descriptions, intercepts the program execution, and transfers the control to the appropriate execution monitors. Our previous work [9] used the Java Platform Debugging Architecture (JPDA)[4] for implementing the event-subscription system to support the use of BVDs for interactive debugging. For testing, this approach of implementation may introduce too much overhead. To reduce such an overhead, our testing tool chooses to use aspects in AspectJ for detecting execution events. The same approach has been used in other projects for similar purposes (e.g., [13]).

5.3 Recording Contract-Based Coverage

Our tool allows the recording of coverage information so that a coverage tool can perform a post-mortem analysis

[4] See http://java.sun.com/products/jpda/.

to measure the adequacy of a test suite. Our tool currently supports two different categories of coverage: the structural coverage and the user-defined coverage. The tool supports two basic types of structural coverage: k-message string coverage and the fragment coverage. In the k-message string coverage, when an event that matches a message arrow is detected, this message arrow and the last $k - 1$ matched message arrows will be recorded. In the fragment coverage, the tool will record the operands of the fragments that have been used during monitoring. Note that, in our internal representation of a BVD, an opt fragment or a loop fragment is extended with an empty operand to represent the skipping of the sequence defined by the opt fragment or loop body [15]. Therefore, the fragment coverage is analog to the branch coverage in source code. More sophisticated structural coverage will be considered in the future.

A user-defined coverage is computed by statements specified in monitoring blocks. For example, by introducing additional parameters into the BVD in Figure 2, the users may introduce monitoring statements in the BVD to record the number of times the node splitting occurs when inserting a key to the B-tree. This kind of coverage can measure the thoroughness of a test suite from the semantics perspective. Therefore, it complements the structural coverage approach.

A BVD may be deployed at different locations in the program. A BVD may also be called by other BVDs through reference uses. Therefore, it is important to distinguish the coverage information recorded for a BVD under different contexts. To support this strategy, our tool allows specific context information to be defined at a deployment-site or a reference use construct. Such context information can be used to annotate the coverage information recorded for the deployed/called BVD under this specific context. We refer to such context information as the *inherited* context for the BVD. The default context information for a deployment-site is the unique ID of this site. The default context information for a reference use construct is the unique ID for this construct plus the inherited context for the containing BVD of this construct. These default values can be overridden by explicitly specifying the context information at the deployment site or at the reference use construct.

Coverage policies can be defined for a deployment-site, for a reference use construct, or for an operand of an interaction fragment. A policy will only take effect during the period of time when the associated entity is being used for execution monitoring. A coverage policy states the kind of coverage that will be recorded. The coverage policy also states whether the recorded coverage information should be annotated with the inherited context of the BVD. When the coverage policy is defined for an operand of an interaction fragment, the policy can be given a name. For a structural coverage policy, this name can be used within the scope for enabling and disabling the recording of the coverage information. For a user-defined coverage policy, this name can be used for outputting the user-defined coverage information. These capabilities provide flexible ways for recording the coverage information.

6 The Related Work

Testing cannot be fully automated without an *oracle* that automatically checks whether a test run passes or fails. Assertions have been used as a practical tool for detecting software faults (e.g., [12]). They have also been used in practice as an oracle for testing a software component [2, 14]. Traditional assertions are specified as boolean expressions. Recent works (e.g., [5, 7, 13]) allow temporal logic to be used in writing assertions. Temporal logic assertions can specify properties related to particular paths through the component under test (CUT). Theoretically, they may be used for specifying behavior contracts. However, unlike the BVD notation, a temporal logic formula does not directly match to the typical ways in which the scenarios are described. It should be possible to support some limited forms of temporal logic in BVD for improving its expressiveness.

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, presumably, one should be able to use LSCs for specifying behavior contracts and use the play-engine for checking these contracts. 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 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 testing tool (e.g., the value-based binding, monitoring blocks, the flexible deployment).

Given an executable model for the functionalities of a CUT, test cases may be automatically derived from this model; the model can be run in parallel with the CUT for checking the results of the test cases (e.g., [1, 3]). It is foreseeable that this testing strategy would be applicable on models written in any form of executable UML (e.g., [10]). However, an executable model typically requires executable definitions for the actions of the individual objects, which in turn may require the introduction of many design details. In contrast, a BVD is not an executable model for the CUT.

Instead, it focuses on specifying runtime-checkable properties for a set of scenarios; it relies on the CUT for defining how the actions will be performed. Thus, using BVDs may involve less specifying effort than using the executable models.

Voas and Kassab [14] argue that assertions should be placed where they are more desperately needed. A BVD defines scenario-specific contexts for placing assertions about the progress of the scenarios. Such assertions are essential for detecting bugs in the implementation of these scenarios. These assertions may be difficult to specify at code level.

7 Conclusions

This paper introduces the behavior contract as a new concept that supports the testing of object-oriented software. A software behavior contract is a metaphor borrowed from its counterpart in our society. It characterizes the proper interactions among objects for implementing the scenarios of performing a certain task. Specified with our behavior view diagram (BVD) notation, a behavior contract can be automatically checked by our testing tool during program execution to validate test runs. A BVD can also be enhanced with instructions for our testing tool to record contract-based coverage information. Such coverage information provides new ways for evaluating the adequacy of a test suite.

A major *benefit of using behavior contracts* is that it provides a practical approach for the software developers to better use their high-level knowledge of a system in testing. In many application domains, the expected functionalities of the system are often identified as scenarios, and specified informally and partially as use cases or user's stories [6]. These specifications are digested and refined by the software developers and turned into knowledge about objects and object interactions that can guide the implementation of the system. In testing, it is important to see whether the system behaves as expected according to such knowledge. Behavior contracts provide a runtime-checkable description of the object interactions. Using behavior contracts and our testing tool enables software developers to use their high-level knowledge obtained during analysis and design to detect bugs during testing.

A major *benefit of using our BVD notation* for writing behavior contracts is that it enables the reuse of design artifacts for testing. Sequence diagrams specified during design can be turned into BVDs by enhancing them with monitoring specific information. In fact, using the extensions that we propose, it may be possible to produce, during design, "monitor-able" sequence diagrams that can directly be used for testing. This reuse not only improves the efficiency of testing, but also adds incentives for a software team to produce and maintain the design artifacts.

Currently, we have built a prototype monitoring component that can automatically monitor the program execution based on our behavior contracts. We are still building the compiler for translating the launch blocks into Java statements, and the runtime support for extracting contract-based coverage information. Once the implementation for these components is completed, we will perform case studies to evaluate the usefulness and the usability of our tool. We will also continue developing more powerful notations for specifying behavior contracts.

References

[1] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Towards a tool environment for model-based testing with AsmL. In *Formal Approaches to Software Testing: Third International Workshop, FATES 2003*, pages 252–266, 2003.

[2] R. Binder. *Testing Object-Oriented Systems*. Addison-Wesley Professional, 1999.

[3] G. Devaraj, M. Heimdahl, and D. Liang. Coverage-directed test generation with model checkers: Challenges and opportunities. In *IEEE COMPSAC'05*, 2005.

[4] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

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

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

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

[8] S. W. Lee, editor. *Encyclopedia Of School Psychology*. Sage Publications Inc, 2005.

[9] D. Liang and K. Xu. Monitoring with behavior view diagrams for scenario-driven debugging. In *IEEE Asia-Pacific Software Engineering Conference*, Dec. 2005.

[10] S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.

[11] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[12] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.

[13] V. Stolz and E. Bodden. Temporal assertions using aspectj. In *Fifth Workshop on Runtime Verfication (RV'05), Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2005.

[14] J. Voas and L. Kassab. Using assertions to make untestable software more testable. *Software Quality Professional Journal*, 1(4), 1999.

[15] K. Xu and D. Liang. Supporting scenario-driven debugging with behavior view diagrams. Technical Report 06-002, Dept. of CSE, Univ. of Minnesota, 2006.