Assertions in model-based debugging*

Wolfgang Mayer Dominik Wieland
Technische Universität Wien,
Institut für Informationssysteme
Favoritenstraße 9–11, A-1040 Wien,
Email: {mayer,wieland}@dbai.tuwien.ac.at.

September 17, 2001

1 Introduction

This document describes the design and implementation of an assertion language that is used within the Jade debugging environment.

The Jade assertion language helps to specify assertions and expected values of variables and expressions of Java programs. In particular, the language provides means to embed assertions either directly in the Java source code or to store them in separate files. The debugging environment is modified such that the assertions replace the traditional observation specification mechanisms of the debugger. Consequently, it is now possible to run test cases unattended. Also, multiple test cases can be applied concurrently to improve diagnosis quality.

2 Syntax

The syntax of the assertion language is chosen in a way that it can be easily integrated in existing Java programs. An important feature of the language is that the resulting program is still a valid Java program and can be parsed and compiled by existing Java tools. This is achieved by embedding the assertions within Java comments, similar to Java documentation comments (`/** ... */`). To distinguish assertions from ordinary Java comments and to allow for simpler parsing, the assertions must be delimited by the strings `/*@` and `@*/`.

Besides the delimiting tokens, an embedded assertion may consist of several parts. First, an assertion may be prefixed with the name of a test case the assertion belongs to. This allows the specification of assertions of several test cases in a single file. If this part is not present, it is assumed that the assertion is valid for all test cases. The main

* This work was partially supported by the Austrian Science Fund project P12344-INF and project N Z29-INF.

part of an assertion consists of a boolean expression that must evaluate to true. Here, any valid Java expression can be used, provided that the semantics of the program is not altered.

The above mechanisms to specify assertions is sufficiently general to specify almost any reasonable assertion used in the Jade software debugging environment. However, as the number of assertions and test cases increases, the program soon becomes cluttered with assertions. To avoid this, the assertions can be put in an extra file and be associated with parts of the program. This is done by introducing labels which denote parts of the program after which the assertions are inserted. Labels are specified similar to assertions, but without the expression part. In the separate file, the assertions are prefixed by the label to which the assertion belongs.

The assertions and assertion labels can be inserted into the program at several places: First, pre- and postconditions of methods are specified by inserting assertion comments immediately before or after the method's body. Further, assertions can be inserted within methods after each statement. Note that assertions on the expression level cannot be specified in the current implementation. However, the specification of assertions on the statement level is sufficient for our purposes.

As an example, consider the following program:

```
class Adder {
    public static void test (int a, int b, int c)
    /*@a0b0c0: (a==0)@*/ /*@a0b0c0: (b==0)@*/ /*@a0b0c0: (c==0)@*/
    /*@((a ==0) || ( a==1))@*/ /*@((b ==0) || ( b==1))@*/ /*@((c ==0) || ( c==1))@*/
    /*@test_entry@*/
    {
        int s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13;
        int z, cn;

        s1 = and(b,c);
        s2 = and(a,c);
        s3 = and(a,b);
        s4 = or(s1,s2);
        cn = or(s3,s4);
        /*@a0b0c0: (cn==0)@*/

        s5 = not(a);
        s6 = not(b);  // and(b,s5)
        s7 = not(b);
        s8 = and(a,s7);
        s9 = or(s6,s8);

        s10 = not(s9);
        s11 = and(c,s10);
        s12 = not(c);
        s13 = and(s9,s12);
        z = or(s11,s13);
        /*@a0b0c0: (z==0)@*/
    }
    /*@((cn ==0) || ( cn==1))@*/ /*@((z ==0) || ( z==1))@*/
```

```
31      /*@test_exit@*/
32
33      static int not(int x) { return 1 − x; }
34      static int and(int x, int y) { return x * y; }
35      static int or(int x, int y) { return x + y − (x * y);}
36  }
```

In the example several assertions are specified: First, a set of preconditions is specified for a test cases named *a0b0c0*. The three assertions require the values of the variables a, b and c to be 0. In the next line, further preconditions are given which require the values of the parameters to be either zero or one. These assertions do not include the name of a test case and are thus associated with all test cases. In line 5 of the program, an assertion label named $test\_entry$ is declared that can be used to specify assertions in a separate file (this method is illustrated below). The postconditions of the method are specified in a similar way (again, there are assertions limited to a single test case as well as assertions that apply to all test cases and an assertion label). Note that the assertions for test case *a0b0c0* are not formulated as postconditions, but are embedded within the method's body. Therefore, they are checked as soon as the execution reaches the assertion.[1]

In addition to the above assertions, an additional test case is specified in a separate file `Adder.ass`:

```
test_entry : [t011 : ( a==0)]
test_entry : [t011 : ( b==1)]
test_entry : [t011 : ( c==1)]
test_exit  : [t011 : ( z==0)]
```

Here, a test case named *t011* is defined that requires the values of the parameters a, b and c to be 0, 1 and 1, respectively. Further, the value of the variable z is required to be zero after the execution of the method. Note that no constraint is specified for the variable cn. However, the assertions applicable to all test cases are added to the test case. Therefore, additional constraints requiring that the values of cn and z are either zero or one are added to the test case (the constraint that the value of z is 0 or 1 is redundant).

2.1 Grammar

This section lists the detailed grammar rules that define the language (EBNF notation).

AssertionComments	::=	{ AssertionComment }
AssertionFile	::=	{ AssertionLabel : [Assertion] }
AssertionComment	::=	AST (AssertionLabel | Assertion) AET
AST	::=	/*@
AET	::=	@*/
AssertionLabel	::=	Name
Assertion	::=	[Name :] (Expression)

[1] In this case this fact is irrelevant, as each variable is assigned only once. However, if a variable holds multiple values during the execution of the program this mechanism is able to check every version of the variable, whereas the postcondition is limited to the last assigned value.

The non-terminals $Name$ and $Expression$ are defined as in the Java Language Specification [1] and denote simple identifiers and Java expressions.

The integration of the above language definition is done through two modifications of the Java parser, resulting in the following production rules (changes are emphasized):

MethodBody	::=	*AssertionComments* Block *AssertionComments*
BlockStatement	::=	(LocalVariableDeclarationStatement
		| ClassOrInterfaceDeclaration
		| [Identifier :] Statement) *AssertionComments*

The separate assertion file is parsed according as specified above (non-terminal *AssertionFile*) using a modified Java parser.

2.2 Semantic restrictions

Although the grammar given in the previous section allows any valid Java expression to be used as assertion expressions, some semantic restrictions have to be followed. Note that these restrictions are not checked by the system. It is assumed that all assertions are well-formed according to these rules.

- First, any expression used as assertion must be executable at the point of specification. Pre- and postconditions must be executable before the first and after the last statement of the method, respectively.[2]
- The expression must not alter the behavior of the program. In particular, no values may be assigned to variables and no methods with side-effects may be called (although the creation of new objects is allowed if they are not reachable from any variable of the program.)
- The expression must evaluate to a boolean value and it must not throw an exception.
- The names of test cases and assertion labels must be globally unique. They are not local to a method, a class or a package.

2.2.1 Additional restrictions (FD model)

For the model based on functional dependencies (FDM), some more restrictions have to be followed.

- The syntax of expressions is restricted such that only one variable of the method is used. Otherwise, the assertion's associated variable cannot be determined automatically.
- Variables and fields may be used in expressions as far as they are used in the enclosing method. Unused variables and fields may not be included in assertions.

[2] Postconditions must be executable immediately after the evaluation of the return-expression of each return statement of the method.

- The form of assertion expressions used in preconditions of methods is limited to $V == C$, where V denotes a parameter or global variable and C represents a constant expression.
- All input parameters and used global variables must be assigned a value for each test case. Otherwise, the program cannot execute and the assertions cannot be compared with the computed values.

Assertions that do not respect these restrictions are assumed to be correct and are ignored.

2.2.2 Additional functions (VBM)

To be able to specify object structures that are not created by the program (for example, specify an object that is passed through a parameter), an additional function symbol `_id(/`$_2$`)` can be used to create an object identifier for the object. The function expects the name of the type the object is of as first argument and a string containing the name of the object as second argument. The name of the object may be chosen arbitrarily (the empty string is not allowed) and must be unique per test case.

As an example, assume the variable x denotes a formal parameter of type $SomeType$. Assume the member $value$ of x should be set to '123' for the current test case tc. This can be accomplished by

```
/*@tc:  (x == _id(SomeType, ``obj1''))@*/
/*@tc:  (x.value == 123)@*/.
```

Here, the name of the object is set to '*obj1*' and the value of its member $value$ is set to 123.

Note that the implementation of the above mechanism is limited to the plain VBM and cannot be used with the model based on storage shape graphs (SGVBM) [2]. Consequently, no observations for parameter variables of reference-type can be specified for the SGVBM.[3]

3 Implementation

This section describes the current implementation of the assertion mechanism and its integration into the debugger. As the implementations for the FDM and the VBM differ significantly, they are described separately.

Common to both models is the implementation of the parsing mechanism and the association between assertions, assertion labels and nodes of the parse tree. When the parser recognizes an assertion comment, its contents is parsed and added to the current node of the parse tree (representing a statement or a method declaration). When separate assertion files are parsed later-on, the assertions therein are added to the node

[3] In order to allow for specifying input parameters of reference type, the assertion language would have to be extended such that the declaration of aliasing properties between objects is possible.

of the parse tree that contains the same assertion label as the assertion. Note that the current implementation requires the assertion labels to be globally unique.

A further requirement is that every test case must be present in the method's precondition in order to be applicable to the method. Otherwise the system does not include the test case in the list of defined test cases. In case the method has formal parameters, this is not considered a restriction, as every test case must fully specify the values of the formal parameters. If no parameter variables are present, an empty precondition (containing the constant *true* as condition) can be inserted.

3.1 Implementation using the FDM

The FD model separates the model building process of the program from the modeling of the assertion expressions. The overall process can be summarized as follows:

1. Build a model of the program as before, ignoring all assertions.
2. Generate an instrumented version of the program, including all assertion expressions. The assertions are converted into equivalent Java code and their result is written to a log file for later use. Note that this step can be fairly complex for return statements and constructor invocations, as the expressions have to be duplicated and wrapped in function calls. Preconditions of methods are handled specially, as they are converted into statements assigning the specified value to the corresponding parameter variable.
3. Read the log and extract all values computed by the program and all values resulting from assertion components. For all assertions that are associated with exactly one variable, compare the computed value with the value of the assertion expression and set an observation accordingly. The connection representing the occurrence of the variable is observed as *not ok* iff the values are different. For assertions that evaluated to true no observations are specified.[4]
4. Perform diagnosis process as before.

3.2 Implementation using the VBM

In the VBM, assertions are treated as ordinary elements of the program, except that they are assumed to be always correct. Therefore, the assertion expressions are integrated into the modeling process right from the start. When building the model of a program, all assertions of all test cases specified so far are included in the model. The output connection of a component representing an assertion expressions is connected to an auxiliary component that is responsible for enabling or disabling the assertion, depending on the currently active set of test cases. When the assertion is active, the component propagates the value *true* to the output of the assertion component. Otherwise, the value remains unknown. By the properties of the model, there is no need for

[4]This is impossible for general expressions. Consider the assertion $x > 5$. If the assertion evaluates to true it is not guaranteed that the value of x is correct.

special handling of preconditions, as the assertion components can also compute backwards for most expressions and in this way determine the values of variables. Note that this is not possible in general, as back-propagation is not possible if more than one value of the expression's input connections are unknown. However, this cannot occur for parameter variables. The diagnosis process itself can be applied unchanged.

The advantage of this implementation is its simplicity. However, for programs with a large number of observations the number of components required to represent the observations can cause the required computational effort to increase substantially. This is especially true for methods that are embedded within loops or are called multiple times. In these cases a similar approach to the FDM could be used, but is currently not implemented.

Another drawback of this implementation is the fact that the model of the program has to be rebuilt whenever the set of assertions changes.

3.2.1 Test scripts

To execute test cases unattended a simple test driver is implemented. The following example illustrates its use:

```
| testsuite |

testsuite := Root.Jade.TestScriptRunner new.
testsuite loadSettingsFrom: '/home/mayer/jade/system/settings.sys'.

testsuite
        runAssertionTestOn: '/home/mayer/jade/testcases/Adder/Adder_f1.java'
        class: 'Adder'
        method: 'test(int, int, int)'
        usingAssertions: '/home/mayer/jade/testcases/Adder/adder.ass'
        testcases: #(t000)
        logFile: '/home/mayer/jade/testcases/Adder/results/adder_f1.t000.log'.

testsuite
        runAssertionTestOn: '/home/mayer/jade/testcases/Adder/Adder_f1.java'
        class: 'Adder'
        method: 'test(int, int, int)'
        usingAssertions: '/home/mayer/jade/testcases/Adder/adder.ass'
        testcases: #(t011)
        logFile: '/home/mayer/jade/testcases/Adder/results/adder_f1.t011.log'.

testsuite done.
```

First, a new instance of the the test driver is created and the test driver is initialized with the settings of the debugger (loaded from `settings.sys`). To run a test case, several parameters have to be provided:

- the name of the source file of the program,
- the name of the class that contains the method to be debugged,

- the name of the method to be debugged,
- the name of the assertion file to be loaded,
- a collection of the names of the test cases to be used when debugging, and
- the name of the log-file the diagnosis results are appended to. Note that the previous contents of the file is not deleted.

It is important that the names of the class, the method and the test cases are specified exactly as displayed by the debugger (i.e. without additional spaces, etc.). Otherwise, the test case fails and the test driver stops operation.

In the example above, two tests are run on the same method using different test cases.

Finally, the test driver is closed to destruct the debugger. Afterwards, the test driver cannot be used any further.

The log file created by the debugger contains several relevant informations obtained from the debugging process. As an example, the results of the first test case from above are presented below (comments and empty lines have been deleted).

```
Source File: /home/mayer/jade/testcases/Adder/Adder_f1.java
Assertion File: /home/mayer/jade/testcases/Adder/adder.ass
Method: Adder.test(int,int,int)
Test cases: #(#t000)
Time: April 30, 2001; 3:42:19 am

Performing syntactic parsing ...
Parsing file: /home/mayer/jade/testcases/Adder/Adder_f1.java
Performing semantic parsing pass 1 on internal parse tree ...
Semantic parsing pass 1 done (1ms).
Total number of warnings: 0

Performing semantic parsing pass 2 on internal parse tree ...
Semantic parsing pass 2 done (10ms).
Total number of warnings: 0

Performing semantic parsing pass 3 on internal parse tree ...
Semantic parsing pass 3 done (29ms).
Total number of warnings: 0

Parsing done (865ms).

Parsing file: /home/mayer/jade/testcases/Adder/adder.ass
Assertion file parsing done (1182ms).

Active testcases: t000

Generate VB-Model (651ms)
Size of component focus: 30

```

```
Source code:
...package: default
...class: Adder
...method: test(int,int,int)

Diagnosis system:
...number of components: 215
...number of connections: 218
...number of observations: 1
...maximum number of diagnoses: 100
...maximum size of diagnoses: 1

Method inputs:
none

Diagnosis time: 6117 [ms]
Number of diagnoses: ’10’
Diagnoses: Root.Modis.DiagnoseSet ([ ab(DC[s12=not(c)]) ]
[ab(DC[not(b)]) ] [ ab(DC[s13=and(s9,s12)]) ] [ ab(DC[not(c)]) ]
[ab(DC[or(s11,s13)]) ] [ ab(DC[s9=or(s6,s8)]) ]
[ ab(DC[z=or(s11,s13)])] [ ab(DC[or(s6,s8)]) ] [ ab(DC[and(s9,s12)]) ]
[ ab(DC[s6=not(b)]) ])
Number of observations: 1
Observations: Dictionary (’t000’->Dictionary (’_result[176]’->true
’_result[18]’->true ’_result[10]’->true ’_result[14]’->true
’_result[180]’->true ) )

Total:
=====
...interactions: 0
...queries: 0
...diagnosis time: 6117 [ms]
...measurement selection time: 0 [ms]
...diagnosis size sequence:
’10’
```

Note that some statistics are not meaningful for assertion debugging (i.e. measurement selection and related entries) but are included because the report is generated by the generic reporting facility of the debugger, which includes these entries.

References

[1] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Publishing Company, 1996.

[2] Wolfgang Mayer. Evaluating Value-based Models for Java Debugging. Internal documentation, 2001.