A Symbolic Java Virtual Machine for Test Case Generation

Roger A. Müller, Christoph Lembeck, and Herbert Kuchen
Department of Information Systems, University of Münster
Leonardo Campus 3, 48149 Münster, Germany
email: {piromu, pichle4, kuchen}@wi.uni-muenster.de

ABSTRACT
Quality management is becoming a more and more important part of the software development process. As software testing is currently understood as the core function of the quality managment, developers start using software testing tools to facilitate their work. However most existing tools just manage given sets of test cases and check them against pre-defined testing criteria. The necessary test case discovery is usually up to user, who has to generate the test cases in an ad-hoc approach with only minimal support from the software. GlassTT, the tool we present in this paper, faces this problem by creating the needed test cases for a given criterion for a Java class file. It uses a symbolic Java virtual machine to generate constraints representing the conditions for the control flow under consideration. The system can employ a parameterizeable test criterion such as data-flow coverage. The constraint solvers embedded in the tool are capable of solving linear and non-linear constraints, which is sufficient for almost all constraints encountered in the execution of Java programs. They are encapsulated in an incremental constraint solver manager, which dynamically chooses an appropriate constraint solver and enables the efficient communication between constraint solvers and symbolical virtual machine.

KEY WORDS
Software Testing, Structural Testing, Symbolic Virtual Machine, Java

1 Introduction and Related Work

Recent studies show that quality management in general and software testing in particular have a big share on the total development cost for software. Manual testing, which has been employed by generations of programmers and testers, has proven time consuming, costly and sometimes imprecise. Thus developers increasingly rely on software testing tools, but unfortunately the test utility market has traditionally been dominated by tools that either manage test cases (i.e. regression test suites) or check existing test cases against a pre-defined coverage criterion, both glass- or black-box. If a coverage criterion is not met, it is usually up to the user to figure out which new test cases are required to gain a better coverage. These test criteria are split up into glass- and black-box criteria. A black-box criterion derives the test cases from the specification, whereas glass-box testing is based on the analysis of the code.

Our test tool generates test cases for glass-box testing automatically. Based on a user defined criterion a symbolic execution of the Java byte code [LY99] of a method is performed. In this context symbolic essentially means that the content of a variable is an expression w.r.t. input parameters (e.g. the parameters of the considered method) rather than a value. The symbolic execution produces a system of equations on the fly, which is formed by the conditions to execute the taken path. This is because of branching instructions such as `ifgt`, where a constraint solver and a testing criterion are used for the decision process to determine which branch to take. If two (or more) alternatives still persist after the decision process, the *symbolic Java virtual machine* (SJVM) uses a backtracking mechanism similar to that of the Babel Abstract Machine (LBAM) [MK90] or the Warren Abstract Machine (WAM) [Wa83] in order to try them gradually. At the end of the symbolic computation, one particular solution of the generated system of constraints will be determined. This solution represents the input parameters, which should be used to run the method fulfilling the coverage criterion selected. If the method has a return value, the symbolic execution results in a test case in the sense that the considered byte code has to produce the computed result, if executed with the appropriate values for the input parameter. Due to backtracking, alternative computation paths and the corresponding test cases will also be determined.

If the symbolic computation would closely follow the actual concrete computation, this would be too expensive and unnecessarily precise. Thus, in our approach the symbolic computation is guided by a user-specified coverage criterion. In the present paper, we will focus on the well-known def-use-chain criterion [Be90], but we would like to point out that our approach can be a adapted to other criteria as well. A def-use chain is roughly a sequence of instructions beginning with a definition of a value and an instruction using this value, where the value is not destroyed between both instructions. If all definitions and uses in a certain piece of code have been reached, it (often) needs not be considered further and the search space explored by our backtracking mechanism can be cut. In practice, few

iterations of each loop are mostly sufficient to cover all def-use-chains.

If the symbolic execution is performed for a given procedure or method, the generated system of equations should be solvable for suitable input parameters. The solutions represent the input(s) required for a set of test cases that satisfy the given test criterion.

The symbolic Java virtual machine (SJVM) described in this paper has been implemented in a prototype, GlassTT, which has itself been implemented in Java and is thus executed in a Java virtual machine (JVM). The SJVM is currently only capable of handling a single thread, the multi-threaded properties of Java are therefore out of scope of this paper.

This paper is organized as follows: The following section introduces the running example in this paper. Section 3 covers the details of the SJVM, section 4 reflects the implementation. Section 5 gives an overview over the related work, and in the last section we conclude the paper by an overview and by a summary and give an insight into future work.

2 Running Example

As a running example for this paper we have chosen the Java method `checkValues` that performs some simple computations:

```
public long checkValues(int x, int y,
    int z) throws Exception {
long result=x;
int  partresult=(y-2)+z;
try{
  if (((double)((x+5)+partresult))<=0)
      throw new Exception();
  else result=(double)((y-2)+z)+(x+5);
} catch (Exception e) {result=-1;}
return (result);
}
```

Using a sufficiently optimizing compiler this will result in the byte code as seen in Fig. 1.

```
Method double checkValues(int, int, int)
  00-12 <build exp> 30-40 <rebuild exp>
  13 iload 5        41 dstore_3
  15 iadd           42 goto 51
  16 i2d            45 astore 6
  17 dconst_0       47 ldc2_w #4
  18 dcmp           50 dstore_3
  19 ifgt 30        51 dload_3
  20-28 <Exception> 52 dreturn
  29 athrow
```

Figure 1. Example 1 (shortened)

3 Symbolic Java Virtual Machine

Java compilers usually produce intermediate byte code as an output, which is usually executed on a virtual machine. The symbolic execution in combination with constraint solving has been explored before as a technique to produce test cases, however no feasible tool has yet been developed (please refer to the related work section for details).

In this section we will show how these two techniques can be combined. The key component of Java is the Java Virtual Machine, namely its execution engine. Starting from this component we will explain how our machine works.

3.1 Execution Engine

Both the JVM and the SJVM execute byte code, though the SJVM executes the code symbolically. This means that expressions are assigned to a variable on the heap, where an expression can be any arithmetic or Boolean combination of Java primitive types. Computation on the fly is only done for the simplification of expressions, e.g. $a = x + 1 + 1$ is simplified to $a = x + 2$. All variables are expressed w.r.t. constant values and the input variables of the methods. Please note that no code outside a method exists in Java. The same symbolic representation is chosen for the elements on the stack. Technically the expressions are put neither on the stack nor the heap of the SJVM, both only contain references. The actual expression is an object tree, which is situated in the heap of the hosting virtual machine. From the point of view of the implementation this has the advantage that the management and disposal of the expressions is executed efficiently by the JVM and is kept transparent to the SJVM. Figure 2 depicts pushing of a variable on the heap, in this case an `iload 5` instruction (see line 13, Fig. 1), which puts the integer value of the fifth local variable on the stack.

When either the heap or the stack are accessed for an arithmetic operation such as `iinc` or `iadd`, the arithmetic operation and its operands are added to the object tree in the host's JVM. The operation becomes the new top of the tree with the operation's operands, which itself may be expression trees, as arguments. Other variables or stack elements pointing to expression trees used in the operation will not be affected. This situation is shown in Fig. 3 for `iadd` (Fig. 1, line 15).

The JVM specification employs trustful typing, i.e. the compiler takes care that the typing is correct whilst the virtual machine expects that nearly all type checking is done prior to run time (see Chapter 3.2 in [LY99]). The constraint system of our SJVM on the other hand keeps track of the type information itself - both to ensure the domains of the types are not violated and to know whether and when a variable has to be integer or floating-point. The information is saved in a simple attribute in the SJVM, in the figures we will depict integer values in the heap with a square box and floating point numbers with a circle. As an

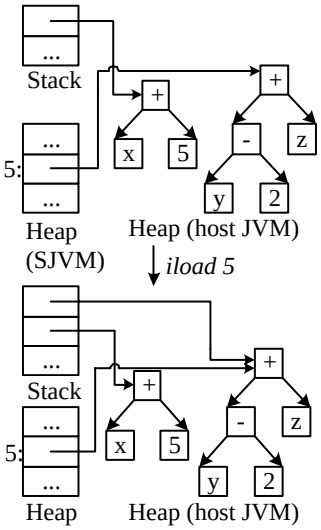

Figure 2. Transformations on `iload 5`

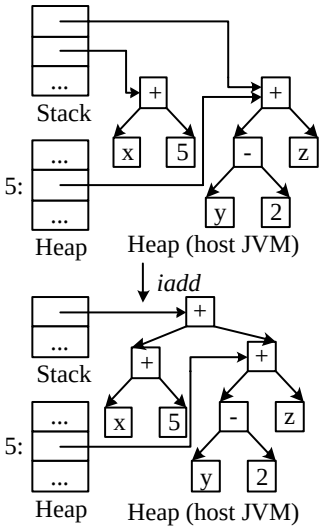

Figure 3. Transformations on `iadd`

example take the byte code type cast operation `i2d`, which casts the top of the stack from `integer` to `double` as seen in Fig. 4. Note that all computations before the type-cast still appear as integer, this represents the fact that at the point of computation these values were integer numbers and not floating point numbers. Some parts of the original JVM specification can be used without bigger modifications, namely unconditional jumps and genuine stack operations like `pop` or `swap`. Due to a lack of space we will not discuss those in detail.

3.2 Backtracking

The biggest change to the SJVM in comparison to the JVM lies in its ability to backtrack. The SJVM itself needs this capability, as the execution engine will need to execute more than one path, if the testing criterion requires it to do so. If one or more branches of a conditional jump are available, the virtual machine takes a branch based on a configurable testing criterion as described in subsection 3.4. If this testing criterion requires the execution of more than one branch, the virtual machine will have to backtrack when the execution of the previously taken path (and all its subpaths) is finished. The *backtracking* mechanism used for the SJVM is similar to the one in the LBAM [MK90] and related to the one in the WAM [Wa83].

Two things need to be added to the SJVM to allow backtracking: a *trail* and a *choice point* stack. A choice point is a set of pointers to the stack and the trail as well as a program counter, and it is instantiated every time a conditional jump is reached and subsequent backtracking will be required. The trail on the other hand is used to track the former state of variables or stack elements, when they are changed for the first time (or deleted) after a choice point is instantiated. Prior to changing a variable, a link to the tree containing the current expression in the SJVM's host JVM is created (this is depicted by a small 1 or 0 in the figures). Each time a new choice point is created all variables become tagged to indicate they have not changed since the last choice point was created. Stack elements are only put onto the trail, if an element is popped from the stack, which was situated lower than the element the stack pointer in the current choice point points to. All entries on the trail contain their origin on the heap or stack.

The transactions for a variable change w.r.t. the trail and the choice point stack are depicted in Fig. 5 for the instruction `ifgt 30` (for the creation of a new choice point, Fig. 1, line 19) and in Fig. 6 `dstore_3` (for the handling of the trail, Fig. 1, line 41).

As mentioned above the host JVM will not dispose the old expression tree as long as a link on it still exists.

3.3 Exceptions

A Java exception may be thrown due to an abnormal execution condition, because of a `throw` statement (i.e. `athrow` in byte code) or because of an asynchronous exception. As the latter is out of scope of this paper we will only discuss the first two options. For both we have to distinguish between the casting and the catching of the exception. The catching of the exception is done by using an *exception table*. This table, which is generated at compile time, contains two line numbers per row, which form the upper and lower bound of the `try-catch` statement speaking byte-code wise. A jump target for the case the exception is thrown is also available, as well as the type of exception that is affected by the catch.

When an exception occurs, the Java virtual machine interrupts the current execution of the program and checks the exception table of the current frame. If it contains an en-

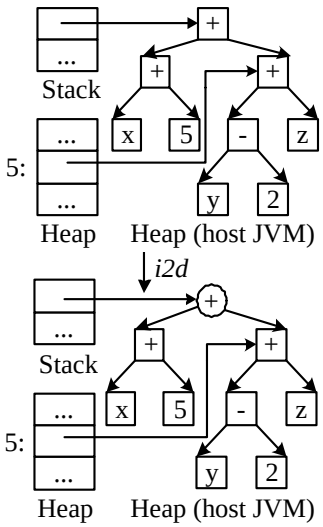

Figure 4. Transformations on `i2d`

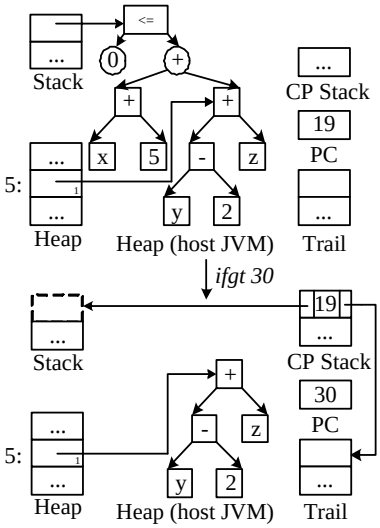

Figure 5. Transformations on `ifgt 30`

try for the line the exception originated from and the type of the exception matches the one from the exception table, the virtual machine continues the execution at the line number indicated in the exception table. Whenever no match for the exception is found, the current frame is destructed and the context is restored to the stack it was called from, and the exception is immediately thrown into that context and dealt with as described above.

As both the throwing of the exception and the handling of the exception may result in additional def-use chains, exceptions are taken into consideration for test case discovery. Dealing with exceptions is rather simple by means of the SJVM. W.r.t. to the type of the exception we can handle them just like jumps, though the jump might take the program execution back to a calling method. Please note that only a subset of the Java byte code instructions is capable of throwing exceptions. The ones that are capable of throwing exceptions will be handled as conditional jumps, which leaves the decision whether or not to invoke the exception up to the testing criterion. Details for the def-use criterion are described in the next subsection.

Some research has been done on the issue of exception based testing criteria, e.g. [SH99]. Though currently no testing criterion based solely on exceptions has found its way into our tool, we plan to implement some in the future.

3.4 Testing Criterion

As pointed out in the previous subsection the SJVM relies on a decision unit, which chooses a branch for execution depending on the environment of the SJVM and a static analysis of the byte code. The implementation of the criterion is held flexible, and several coverage criteria have been implemented, namely def-use chains, branch coverage and instruction coverage. For brevity reasons we will only discuss def-use chain coverage in detail. First of all, as def-use chains are defined non-uniformly in the literature (see e.g. [Be90]), we will give the definition we applied for our implementation:

$$def(S) := \{X | \text{ instruction } S \text{ (re)defines } X\}$$
$$use(S) := \{X | \text{ instruction } S \text{ uses } X\}$$

$[X, S, S']$ is a *def-use chain* for variable X, if $X \in def(S) \cap use(S')$ but $X \notin def(S'')$ for all instructions S'' passed between S and S'.

Taking this definition into consideration, we will explain how the static scan of byte code is performed to discover all existing def-use chains. The first step is to add an initial `store` for all input parameters, as the virtual machine assigns the parameters to the first local variables when the new method is called. As the second step, build a directed flow graph for the byte code, as this allows a depth first search for def-use chains. While traversing the flow-graph record all encountered definitions and uses of a variable and construct the corresponding def-use chains. If the same def-use chain is discovered twice break the traversion and backtrack, as this indicates a loop contains no new information. The same applies for the termination of the program.

A special issue for the generation of the flow-graph arises because of the implicit exceptions that can occur while executing certain byte code instruction. At that point, the probability has to be explored whether the exception results in the use of a variable, which is in scope of the current def-use chain analysis. This can be done by treating the instructions like conditional jumps, where the constraint is the condition to throw the exception. The byte code instruction `idiv` for example throws an exception, if a division by zero occurs. If the `idiv` instruction is embedded

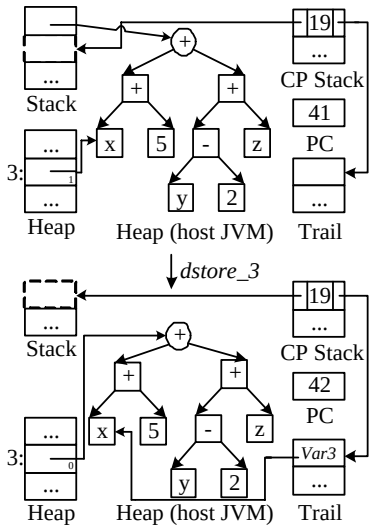

Figure 6. Transformations on `dstore_3`

in a try-catch clause, the exception could be caught and the result of the division be set differently like in the following example:

```
public static int div(int a, int b)
    throws ArithmeticException {
int result;
try{ result=a/b;}
catch (ArithmeticException e) {
    result=0;}
return result;
}
```

Thus the catch clause includes a new def-use chain.

After all the def-use chains have been obtained, the SJVM starts the symbolic execution of the program. When a conditional jump, which in Java byte code also includes if-clauses and loops, is reached, the virtual machine decides based on the information gathered by the pre-scan and the constraint solver, whether to continue with the true- or false-branch of the conditional jump. A constraint solver, whose properties are discussed in the next subsection, is used to check whether branches become unreachable: an additional equation is added to the constraint system describing the current program flow and the constraint solver can check the ongoing solvability of the constraint system. If e.g. the condition is "`i<z-y`", "`i<z-y`" will be added in case the condition is assumed true, or "`i>=z-y`" if the condition is assumed false.

3.5 Constraint Solver and Constraint Solver Manager

As noted above the SJVM relies on constraint solvers to resolve both the input variables and the feasibility of paths. To ensure flexibility and extensibility of the constraint solving process we have employed a two stage mechanism for the constraint solving. The "upper" level is made up by the *constraint solver manager*. It adds three functions to the constraint solving. First of all it acts as an adaptor for the constraints. While most of the solvers are implemented using the same interface, this enables the usage of an external computer algebra system as Mathematica, which provides additional flexibility for the solving process. Additionally, the solver manager is enabled to decide which solver to use for the properties of the current constraint system. The criterion for the selection is usually the speed the constraints are computed with, e.g. linear constraints can be solved with a truly linear solver, as a non-linear solver will usually compute those much slower.

Secondly, the constraint solver traces the addition or removal of constraints and therefore allows an incremental solution of a constraint system, where the constraint solver can make use of previous (partial) results. For complex computations w.r.t. the structure of the problem this may result in performance improvement.

As a last point, if parts of the constraint system aggravated in the constraint solver manager are non-dependant, the parts can be solved separately, thus allowing simpler and more efficient computations.

The current implementation of GlassTT has a boolean solver, (mixed) integer linear solvers and a non-linear solver, which have proven sufficient for all problems encountered so far. An optional bridge to Mathematica as a solver is also implemented and available.

For further information about the constraint solving system please refer to [ML03].

4 Implementation

As mentioned earlier the concepts discussed in this paper have been implemented in a prototype, GlassTT. This has enabled us to gather feedback from real users, experiment with the technology and thus check the feasibility of our approach.

GlassTT was implemented in Java, an overall structure of the system can be seen in Fig.7.

We have also added some of the functions each Java virtual machine possesses and needs for proper operation, e.g. a class loader and object management. To facilitate the contact to the real user we have also implemented a front-end, i.e. a NetBeans plug-in, which allowed comfortable use (see Fig.8 for a screenshot).

The SJVM itself communicates its results in a XML format to the outside world. For the NetBeans plug-in we have decided to present the test cases in a graphical tree, and optionally in the form of JUnit tests, i.e. Java (source-code) classes that implement the test cases using JUnit. The tool also has a reporting component which can log anything from errors to warnings or simply debugging information, which can be viewed by any XHTML compatible browser.

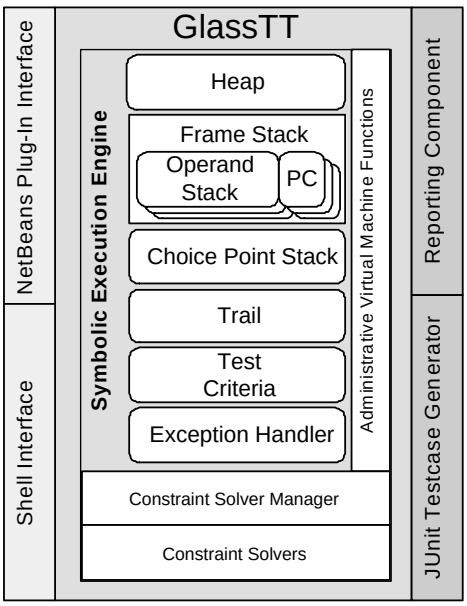

Figure 7. Conceptual View of the Test Tool

Figure 8. GlassTT embedded in NetBeans

5 Related Work

Test data generation has been discussed in many papers (see e.g. [Ed99]), but only a few can directly be related to our approach.

An early approach to test-data generation was the one by Ramamoorthy, Ho, and Chen [RH76], who transformed Fortran-code and tried to solve the resulting constraints describing the sought paths through the program by forward substitution. The work of Offutt and DeMillo [DO91] is based on their unusual mutation analysis test criterion which is utilized to gather constraints, but comprehends a rather simple constraint solver. Korel's [Ko96] approach included test data generation by performing a data dependence analysis on a Turbo Pascal program and using a minimization technique to discover suitable input values. Gotlieb, Botella and Rueher [GB98] have proposed a limited constraint-based approach for a sub-set of the C-language that can identify paths that cover every instruction in the program. Their approach featured neither an exchangeable testing criterion, nor did the test tool contain several constraint solvers chosen due to the structure of the constraints. Lapierre et al. [LM99] implemented a test tool that tries to solve the path-describing constraints by using mixed-integer linear programming for C programs. Pargas, Harrold and Peck [PH99] have suggested test base generation based on a genetic algorithm approach. Finally, Gupta, Mathur and Soffa [GM00] have proposed an approach that uses a branch selection algorithm that generates input data which exercises a selected branch in a program. However their approach features no virtual machine and is strictly numerical.

6 Conclusion and Future Work

We have presented a strategy for automatically generating test cases for Java programs. We have described the layout of a symbolic Java virtual machine, which is capable of discovering test cases using a definable structural coverage criterion. The byte code is executed symbolically, and the decision whether to enter a branch or throw an exception is based on the earlier constraints, a constraint solver and current testing criterion. The constraint solvers, which are also encapsulated in a constraint solver manager, can also help to decide whether a path is infeasible before entering it. If more than one branch of a decision remains feasible, a backtracking mechanism implemented in the virtual machine is used to explore further test cases. The SJVM has been implemented in a test tool called GlassTT, whose properties have been explained in this paper.

The feedback and experience we gained from the use of our prototype let us spot some issues we will improve and extend in the future. First of all we will add more coverage criteria, especially some that are related to method interaction and exceptions. We will also put further work into the graphical user interface, and integrate GlassTT with a regression test tool, so test cases can be reused and do not have to be created on the fly for continuous testing. We will also open our tool for further user interaction, e.g. create test cases for sub paths of a method the user enters into the system using a graphical representation of the flow chart of a given method. Finally we are currently considering adding a symbolic execution engine for the Microsoft Intermediate Language, which is rather similar to Java byte code, and add more constraint solvers to our system.

References

[Be90] B. Beizer. Software Testing Techniques. Van Nostrand Reinhold, 1990.

[DO91] R.A. DeMillo and A.J. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on SE, Vol.17, No.9*, 1991.

[Ed99] J. Edvardsson. A survey on Automatic Test Data Generation. *2nd ECSEL in Linkping*, 1999.

[GB98] A. Gotlieb, B. Botella, and M. Rueher. Automatic Test Data Generation using Constraint Solving Techniques. *ISSTA '98*, 1998.

[GM00] N. Gupta, A.P. Mathur, and M.L.Soffa. Generating Test Data for Branch Coverage. 15th IEEE ASE, 2000.

[MK90] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, M. Rodriguez-Artalejo. Graph-based Implementation of a Functional Logic Language, Procs. ESOP, LNCS 432, 1990.

[Ko96] B. Korel. Automated Test Data Generation for Programs with Procedures. *Software Engineering Notes Vol.21 No.3*, 1996.

[LM99] S. Lapierre, E. Merlo, G. Savard, G. Antoniol, R. Fiutem, and P. Tonella. Automatic Unit Test Data Generation Using Mixed-Integer Linear Programming and Execution Trees. ICSM 1999.

[LY99] T. Lindholm and F. Yellin. The Java(TM) Virtual Machine Specification, Second Edition. Addison-Wesley 1999.

[ML03] R.A. Müller, C. Lembeck, and H. Kuchen. GlassTT - a Symbolic Java Virtual Machine Using Constraint Solving Techniques for Glass-Box Test Case Generation, Technical Report 102, University of Münster, 2003.

[PH99] R.P. Pargas, M.J. Harrold, and R.R. Peck. Test-Data Generation Using Genetic Algorithms. *Software Testing, Verification and Reliability Vol. 9 No. 4*, 1999.

[RH76] C.V. Ramamoorthy, S.-B.F. Ho, and W.T. Chen. On the Automated Generation of Program Test Data. *IEEE Transactions on Software Engineering, Vol.SE-2, No.4*, December 1976.

[SH99] S. Sinha and M.J. Harrold. Criteria for Testing Exception-Handling Constructs in Java Programs. *Technical Report OSU-CISRC-6/99-TR16*, June 1999.

[Wa83] D.H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.