Technology of Test Case Generation

Levi Lúcio* Marko Samer†

April 4, 2004

1 Symbolic Execution

Symbolic execution is a program verification technique born in the 1970s. One of the first papers in the area by King [5] describes the technique as being somewhere between the informal and the formal approaches. The informal approach may be described as follows: the developer creates test cases which are sets of input values to be provided to the application; these test cases are ran against the application which will output the results; the test results are tested for correctness against the expected results. In what concerns the formal approach, it means describing the application by means of a specification language and then using a proof procedure to prove that the program will execute as expected. While the informal approach involves actual execution of the application, the formal one can be applied even before a prototype for the system exists.

Symbolic execution was invented to fill the gap between the two above mentioned techniques. While the informal approach completely disregards input values that are not taken into consideration in the test cases, the formal one requires an exhaustive mathematical description of the application which is not easy to produce.

*The present work is part of the VeDiSS project which is partially funded by HaslerStiftung, DICS initiative.

†This author was supported by the European Community Research Training Network "Games and Automata for Synthesis and Validation" (GAMES) and by the Austrian Science Fund Project Z29-N04.

The first goal of symbolic execution is to explore the possible execution paths of an application. The difference between symbolic execution and informal testing with sample input values is that the inputs in symbolic execution are symbols representing classes of values. For example, if a numeric value is expected by the application, a generic x representing the whole set of numerical values is passed. Obviously, the output of the execution will be produced as a function of the introduced input symbols.

Given that symbolic execution is done over non-defined values, the control paths that are covered have to be defined either by heuristics or by humans at run-time. In particular, the symbolic execution of conditional structures is of great interest: when a symbolic condition is evaluated, the result may be true, false or not decidable. In case of true or false, it is clear which control path should be followed. If the symbolic execution environment is not able to decide unambiguously which branch of the condition should be taken, then both control paths can be followed and the symbolic execution of the program splits. From the above, one can imagine that to each possible program control path corresponds a conjunction of conditions accumulated by the decisions taken during the execution. The set of conditions that defines a control path is called its path condition.

It is now possible to talk about test case generation. As for the other two techniques mentioned in this section (*model checking* and *theorem proving*), test cases can be generated as by-products of symbolic execution. The main goal of symbolic execution is to analyze the control structure of a program and possibly discover errors in it. However, by finding a solutions to the equations that describe control paths it is possible to extract values that can be used as test cases. These values will clearly force the application to follow the control path that defines that path condition.

From the above it can be understood that symbolic execution was invented mainly for white-box testing. Despite, nothing prevents from applying the same techniques starting from an abstract specification such as a state machine. Symbolically searching a state space helps coping with state space explosion since it reduces the number of possible paths by associating classes of inputs. Several authors [9, 6, 7] provide interesting examples of the usage of symbolic execution for generating test cases from an abstract model. From here on in this text we will use the term model to mean both program and abstract specification.

In this section we will provide an account of the above described topics. In particular in Sect. 1.1 we will go through the technique of symbolic execution, showing both how it works and what problems it raises. In Sect. 1.2 the topic of test case generation from symbolic execution is discussed. Since in our days test case generation is mainly done using a conjunction of techniques, we will discuss several test case generation methods where symbolic execution plays a significative role.

1.1 The Technique

As already discussed in the introduction, symbolic execution started to be as a technique to help debugging programs by instantiating input variables with symbols. Each symbol represents the whole range of values a given input variable may assume. As an illustration, consider the code in Fig. 1 which is a C translation of an example that can be found in [2]:

```
    int foo(int a,int b) {
1       a++;
2       if (a > b)
3           a = a − b;
4       else
5           a = b − a;
6       if (a <= −1)
7           a = −a;
8       return a
    }
```

Figure 1: foo C code

In order to execute this piece of code symbolically, we start by assuming the instantiation of the input variables a and b of the *foo* routine by the symbols $\alpha 1$ and $\alpha 2$.

The instruction labelled *1* is an assignment which increments the value of $\alpha 1$. In this case it is simple to see that after this statement is executed we have $\alpha 1 = \alpha 1 + 1$.

The symbolic execution of a conditional statement is however more complicated. If we take the statement labelled *2* from the *foo* routine, there are two cases to consider:

- $\alpha 1 > \alpha 2$: the next instruction is the one labelled *3*;

- $\alpha 1 \leq \alpha 2$: the next instruction is the one labelled *5*.

Since both $\alpha 1$ and $\alpha 2$ are symbolic and represent the whole range of numeric values the input variables to the program may assume, it is impossible to decide whether the program should follow label *3* or label *5*. It is thus necessary to follow both of them and to split the execution in two separate control paths. Each of these control paths will however have a condition attached to it: the control path associated to the fact that $a > b$ is *true* has the $a > b$ condition attached to it; the control path associated with the fact that $a \leq b$ is *true* has the $a \leq b$ condition attached to it. These conditions are called path conditions. If we generalize, a path condition can be seen as a set of arbitrary contraints on input variables.

In Fig. 2 it is possible to observe the partial symbolic execution of *foo* by means of a directed graph. The nodes of the graph correspond to the state of the input variables and of the path condition while the edges correspond to the next statement in line for execution.

From *foo's* symbolic execution much information can be retrieved:

- At each state of the symbolic execution three data are known: the input variable's symbolic value, the path condition's symbolic value and the next statement to be executed;

- Each leaf in the symbolic execution tree corresponds to the end of a control path. The path condition on each leaf is the conjunction of all the assumption made about the input values as the program executes. At the tree's leaves, the path condition fully documents the followed control path;

- This method enables the detection of control paths that are never executed in a model. For example, in *foo* the control path *(1-3,6-8)* is associated with the path condition $(\alpha 1 - \alpha 2 > -1) \wedge (\alpha 1 - \alpha 2 \leq -2)$. There is no solution for these equations so no input values will ever make the program follow this control path.

The example from Fig. 2 only deals with assignments and conditional statements, no loop statements are included. We will not go into the details of how to symbolically execute a loop statement since the algorithm can be extrapolated from the one for symbolically executing a conditional statement:

- if from the path condition it can be deduced that the loop condition is

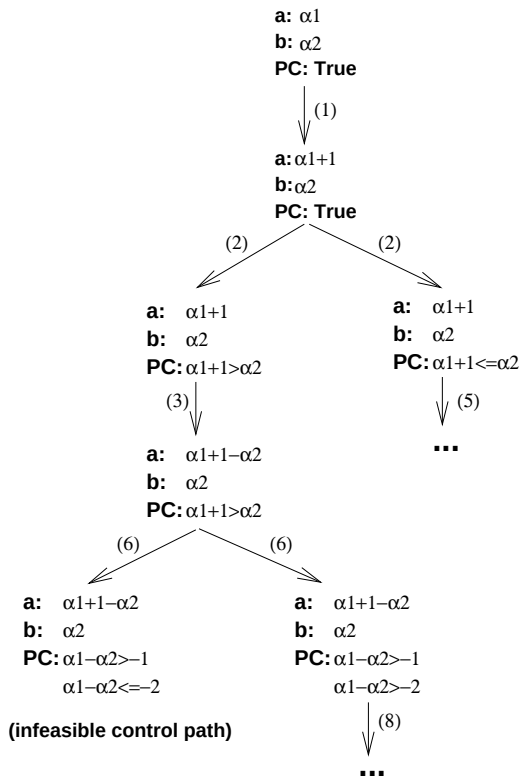

Figure 2: Symbolic execution tree of foo

- *true* then the control path is directed to the beginning of the loop statements;
- *false* then the control path is directed to the first statement after the loop.
- both *true* and *false*, then the control path is split in two as described in the first two bullets.

An interesting case is when the condition expression of a conditional statement involves a subroutine or method call. As an example, imagine instruction *2* of the *foo* routine has the following condition: $f(a) > b$, where f is a function or a method defined elsewhere in the application. In that case two different strategies may be used: consider the return value of the function call as one or multiple symbolic expressions (resulting from the symbolic exe-

cution of the *f* subroutine); consider the *f(a)* expression as another symbolic variable over the possible return values of *f*.

1.1.1 Proving program correctness

It is possible to extract relevant information only from symbolically executing an application model. For example, infeasible control paths may be identified or errors in the code can be detected by looking at the path conditions for each control path. However, using symbolic execution it is possible to go further than that into the domain of proving program correctness.

King [5] discusses the similarities between proving program correctness and symbolic execution. In order to prove that a program is correct it is necessary to state a precondition that constraints input variables and a post-condition that will be required to be true after program execution. It is possible to perform these proofs with symbolic execution since:

- the precondition can simply be conjunct with the path condition at the beginning of the execution;
- the postcondition can again be conjunct with the path condition at the end of the execution of each control path. If there is no solution for the equations posed by the conjunction of a control path's path condition and the postcondition, then that path should not exist.

The proof can also be done in a compositional fashion by making pre and postconditions cover relevant segments of the program. Proving the program is correct corresponds in this case to proving all the specified segments. This technique is used in [5] to show that EFFIGY (one of the first symbolic executors) could be used to prove program correctness.

The authors of [4] use a similar technique for verifying object oriented concurrent programs with complex data structures. Their approach consists of annotating source code with method pre and postconditions and performing symbolic execution using a model checker. Each time the model checker fails to verify one of the preconditions the search backtracks (the path is infeasible) and an alternative execution path is tried out.

1.1.2 Issues related to the approach

Several difficulties arise when trying to execute a model symbolically. As with other verification techniques, the main problem is linked to the fact that the state space for control path verification is usually infinite, as well as the range of values of each input. Despite the advantage offered by symbolic execution of abstracting sets of input values into symbols, solving the *path condition* equations is still necessary in order find which input values yield a given control path. The following bullets discuss these problems and how some authors approached them.

- *Dealing with infinite control paths* If we consider models which comprise loops - which means all the programming languages and virtually all state spaces generated by abstract specifications - there is an infinite number of control paths with infinite states. When a loop depends on symbolic input values it is a difficult problem to automatically understand when the loop execution should stop. Several solutions may be envisaged:

  - simply prompt the user at each iteration of the loop for directions;
  - establish an upper limit on the amount of iterations to be performed on each loop (automatically or by human intervention). This limit is an heuristic and will have an impact on the quality of the generated control paths;
  - Try to automatically find a fixed point to the loop. This is however not trivial and may require human assistance.

  The usual approach implemented in symbolic executors is to provide an upper limit on the number of symbolic executions to be performed (e.g. CASEGEN [10], DISSECT [3]). In DISSECT another approach to controlling control path length is to provide an upper limit for the total control path length, as well as for the total amount of generated paths.

- *Solving path condition equations* This is crucial both for the symbolic execution itself and for test case generation. During symbolic execution it will be necessary to constantly evaluate the path condition equations of in order to decide whether the control path being explored is feasible

or not. If there is no solution to the equations at some moment, the path is infeasible.

In what concerns test case generation, for each feasible control path the *path condition* provides the relation between input variables that will direct execution through that particular path. If it is possible to generate values that satisfy that relation, then it is possible to extract a test case.

- The *path condition* holds a general system of equalities and/or inequalities, for which any algorithm will not be complete. Clark [2] presents a linear programming algorithm that can be applied in the case where the equations are linear constraints. Ramamoorthy et al [10] deal with non-linear equations using a systematic trial and error procedure. This procedure assigns random values to input variables until a solution is found (which is not always possible). Much more recently in [9], random trial and error is also used, in conjunction with limit analysis.

In this subsection we have discussed the fundamentals of symbolic execution. This knowledge provides the basis for understanding the next section - an overview on test case generation using symbolic execution.

1.2 Test case generation using symbolic execution

While going through the available literature on test case generation using symbolic execution we found that the models used to specify the application vary wildly. We have thus opted by describing in this section three examples that have their starting point in three different abstract models: B, AUTOFOCUS and CO-OPN.

More than that, we also found that some interesting techniques that make use of symbolic execution for test case generation don't use abstract models but rather start directly from code. We explore in this section also one of these techniques coming from white-box testing. We find that the example described enriches this survey since it can also eventually be used in test case generation from abstract models.

Another axis where we based this survey on are the synergies between symbolic execution and other program verification techniques in the context of test cases generation. As we have shown in Sect. 1.1, symbolic execution

started out as a pure white-box testing technique during the 1970s. Later, it has been recycled to help reducing state-space explosion problems associated with formal verification techniques such as *model checking* or *theorem proving*.

The examples that follow encompass the application of several techniques for program verification, including obviously symbolic execution. For each of the examples the several techniques employed for test case generation are identified, so that the synergies between them are exposed.

We start with three frameworks for model based test case generation where symbolic execution is heavily used. We then pass onto one example of code based test case generation which we find particularly interesting since it uses model checking to perform symbolic execution.

1.2.1 Abstract Model-based test case generation

The three following examples are relatively similar in the way they approach the problem of test case generation. They all start from an abstract specification and perform searches through the execution state space of the specified application by using a constraint logic programming language or simply Prolog. This search is done in a symbolic fashion in the sense that each state of the model corresponds not to a single concrete state but rather to a set of constrained model input variables. The constraints for the model input variables at a given state are calculated by symbolically executing the path until that state - the same way we have shown in Sect. 1.1.

At this point it seems important to also define what a constraint logic programming (CLP) language is. CLP languages are declarative logic languages, such as Prolog, but particularly enabled to deal with arbitrary constraints. Examples of constraints could be for example $X > 0$ or $Y + Z < 15$. Intuitively, while Prolog's inference engine only understands syntactic unification, a CLP engine includes semantic knowledge about constraints while performing unification. This makes CLPs more efficient than Prolog for performing searches through variable constrained state spaces such as the ones we are considering in this text.

We will start by an example that builds test cases starting from a B

specification. This approach called BZ-TT (BZ-Testing-Tools) is described by Legeard and Peureaux in [6, 7] and consists essentially of three steps:

- *translating a B specification into their custom CLPS-B* constraint logic programming language: B is a specification language related to Z that supports the development of C code from specifications. In B a software application can be seen as a state machine in which each state is defined by a number of state variables and transitions are defined by the operations the state machine accepts for each state.

  The translation step generates CLPS-B prototypes of the operations described in B to allow the animation of the specification. With CLPS-B it is then possible to generate the execution state space for the specified application and to search it for traces that are interesting to be tested. The translation from B into CLPS-B can be seen as a first (abstract) prototyping of the system under test;

- *calculate the boundary states* for each state variable in the specification Boundary states consist of states of the specification execution which are considered to be particular hence should be tested. In order to find these states the test generation framework relies on symbolic execution. In what follows, please keep in mind that this approach is limited to finite enumerated domains.

  Boundary states are calculated in the following fashion: each state variable's[1] value domain is partitioned by symbolic execution of the B specification (by means of the CLPS-B translation). In fact, in a B specification properties of state variables are defined both in the preconditions and the body of operations by structures such as "SELECT...THEN", "IF...THEN...ELSE" or "ANY...WHERE". For each possible execution (each possible trace) of the specification there is a *path condition* associated. The difference with normal symbolic execution is that this time the interest is not on all the conditions posed on all the state variables, but rather on the conditions posed on one single state variable. For example, if we consider that a given execution trace implies the following conditions over state variable x with domain

[1] State variables are related to what we described in Sect. 1.1 as *input variables*

1,2,...,9,10:

$$x \in \{1, 2, 3, 4, 5\} \wedge x \neq 3$$

then x's value domain would be partitioned in the following way:

$$x \in \{1, 2, 3\} \cup \{3\} \cup \{3, 4, 5\} \cup \{5, ..., 10\}$$

From this union of sets called *P-Domain* it is possible to calculate an intermediate product called *boundary values*. These are the values belonging to the extremes of each of the subsets in the *P-Domain* set. If we take the example above, the *boundary values* for $x \in \{1, 2, 3\} \cup 3 \cup \{3, 4, 5\} \cup \{5, ..., 10\}$ would be $\{1, 3, 5, 10\}$. It should be said that we have not taken into consideration state variables over non-numeric sets, although the computation of boundary values for these sort of variables is relatively similar to the previous description.

It is now possible to calculate the *boundary states*. These correspond to states in the execution space of the specification where at least one of the state variables assumes a *boundary value*. Symbolic execution is again necessary at this stage given that in order to know the ranges of all the other state variables that define a *boundary state* it is necessary to know the *path condition* for that state.

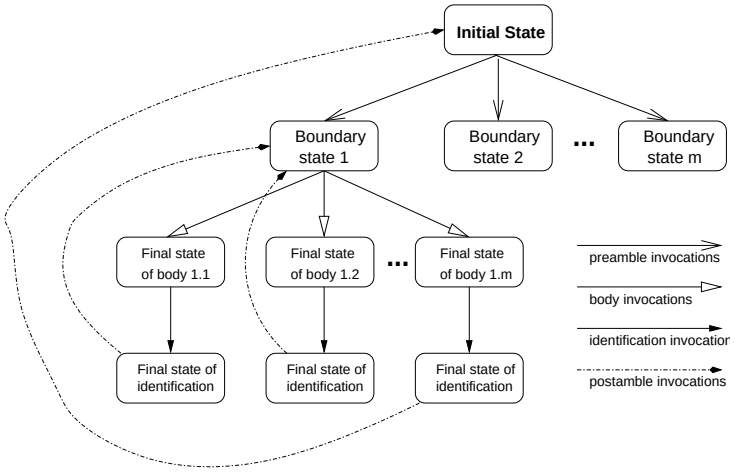

Figure 3: Trace construction for boundary testing

- *generate the test cases (traces through the state space)*: this activity may be resumed to the following:
  - *Calculate the preamble trace to the boundary state*: this consists of calculating the sequence of operations that leads the system to a *boundary state*. Given that the *path condition* for the boundary state is already known, this can be considered trivial;
  - *Calculate the "critical invocation" step*: the authors of the approach define *critical invocation* as the execution of the operations which are possible from the *boundary state*. The execution of these operations in is clearly sensitive since is implies the manipulation of a boundary value. For that reason the input parameters for the operations under analysis are decomposed into subdomains as was done in order to find the *boundary values*. We can say that the operations accessible from the *boundary state* are then symbolically executed over their entry parameters, yielding a subdivision of the *preamble trace* (see Fig. 3);
  - *Calculate the identification traces and the postamble after the "critical invocation"*: The identification trace consists of one or more operations to be executed in order to observe the behaviour of the system after the critical step. The postamble trace is a sequence of operations that resets the state machine to the initial state from where new test cases can be again searched for.

  It is then possible to concatenate the *preamble trace* with the *critical invocation* traces with the *identification traces*. The remaining symbolic parts of the traces are finally fully instantiated in order to generate real test case scripts that can be applied to a concrete implementation of the application.

While trying to discriminate the verification techniques used in this framework we can identify clearly the usage of *symbolic execution*, but also of *theorem proving* since a logic programming language (i.e. a *theorem prover*) is used to calculate the *boundary states*.

The second example on model-based test case generation we will discuss is presented by Pretschner *et al* in [9]. This approach also relies on a CLP tool to symbolically execute the abstract specification of the system. The application's state space is then searched for symbolic traces that can be instantiated to form interesting test cases.

The specification model used in this case in the one used by the AUTOFOCUS CASE tool - inspired from UML-RT (UML for Real-Time systems), especially directed towards the development of embedded systems. In this paradigm the system's structure is defined as a network of components. Each of the bottom level component's behaviour is described by a state machine. Composition of the bottom level state machines generates higher level state machines and so on until the full system's state machine is reached. As in the previous example, states are defined by state variables and the transitions are possible via commands (or operations) that are issued to the system.

Before describing how the test cases are generated, it is useful to mention that the authors of the approach consider different kinds of coverage of the execution state space. In [9] they describe three different coverage classes:

- *Functional coverage*: this sort of coverage implies generating test cases that exercise precise execution scenarios given in the specification. Both positive as well as negative test cases are interesting to validate the system;
- *Structural coverage*: structural criteria implies for example issuing sequences of commands that selectively test critical components of the system. Another example is the coverage of states that may be considered dangerous or unsafe;
- *Stochastic coverage*: using this approach random traces are generated through the execution state space of the application. Despite the search not being directed, this sort of coverage may still produce relevant test cases.

The generation of test cases is done by translating the AUTOFOCUS model into a CLP language so that it can be symbolically executed by a constraint logic engine. The idea is that a bottom level transition of a component K is modelled into a formula of the following type:

$$step^K(\overrightarrow{\sigma}_{src}, \overrightarrow{\iota}, \overrightarrow{o}, \overrightarrow{\sigma}_{dst}) \Leftarrow guard(\overrightarrow{\iota}, \overrightarrow{\sigma}_{src}) \wedge assgmt(\overrightarrow{o}, \overrightarrow{\sigma}_{dst})$$

This means that upon reception of input $\overrightarrow{\iota}$, component K can evolve from control and data state $\overrightarrow{\sigma}_{src}$ to $\overrightarrow{\sigma}_{dst}$ while outputting $\overrightarrow{o}$. In order for this to happen however the transition's guard has to hold. Also, the data state of the component after transition is determined by the assignment function $assgmt$. If we consider a component K that is not a bottom level one, then a transition of component K shall be composed of a set of lower-level transitions of K's subcomponents.

The CLP program representing the application's specification is then executed to calculate interesting traces through the execution's state space. We can say symbolic execution is used here since the traces are built not with actual input values for each transition - rather with constraints over variables representing input values. One of the interesting feature of this particular framework is the possibility of annotating the abstract AUTOFOCUS specification with coverage criteria. These annotations are also translated into the CLP model of the application in order to allow heuristics for trace construction.

Clearly, at the end of the search the symbolic traces need to be instantiated in order to build real test cases that can be used to verify a concrete implementation of the system. This instantiation is done either at random or by limit analysis.

As in the previous example, we can clearly identify in this approach the presence of the *symbolic execution* and the *theorem proving* verification techniques.

The final example on test case generation from an abstract model we describe in this text is presented by Peraire, Barbey and Buchs in [1]. The starting point for the framework is a formal specification language called CO-OPN (Concurrent Object Oriented Petri Nets), also developed by the same group. CO-OPN uses algebraic structures to define data types and the Petri Net formalism to handle concurrency. From a specification in this language an axiomatization in Prolog is produced automatically. The role of this axiomatization is dual:

- it allows the generation of test cases by composition of operations of the SUT's interface. Since there is an infinite random amount of these compositions, the test engineer can apply hypotheses on the behaviour

of the system in order to reduce the initial number of tests. This is done using a special purpose language;

- on the other hand the axiomatization of the specification in Prolog also makes it executable (at a level which is necessarily more abstract than the SUT). This high level prototype makes it possible to validate the generated tests, i.e. checking whether the transitions between the operations in the test sequence are possible. If they are not, then that sequence of operations should not be applicable to the implementation - this type of tests are negative but also relevant to verify the correctness of the SUT.

The next step in the approach is to define a set of hypotheses that will direct the symbolic execution of the axiomatized prototype. Unlike other frameworks described in this section, this one relies on human intuition during test case selection. Despite the fact that some of the possible automation during this step is lost, the high-level language used to describe hypotheses about interesting test cases provides a basis to generate tests which are semantically meaningful.

The test engineer can express two types of hypotheses concerning the tests that will be generated:

- *Regularity hypotheses*: this type of hypotheses stipulates that if a test containing a variable v if valid for a subset of v satisfying an arbitrary complexity criteria, then it is valid for all of v's domain of greater complexity. The notion of variable in a test is very generic, including not only input variables but also constraints on the shape of the sequence of operations that form the test. This is however a complex topic and the reader is referred to [8] for details;

- *Uniformity hypotheses*: the uniformity hypotheses state that if a test containing a variable v is valid for one value of v, then it is valid for all of v's domain.

After introduction in the system of hypotheses by the test engineer, the prolog adapted engine (the resolution is not pure SLD[2]) symbolically executes the uninstatiated tests against the axiomatic definition of the application.

[2]SLD is the standard mechanism used in logic programming languages in order to compute goal solutions

The idea behind the approach is to extract from path condition of a given test the constraints on the variables corresponding to the input values of the operations present in that test. Given this knowledge it becomes possible to calculate the subdomains of the uninstatiated input variables and apply uniformity hypotheses in a way that the operation behaviours described in the specification are taken into consideration. This activity is somehow equivalent to what is performed by Legeard and Peureaux in [6] (see Sect. 1.2.2) while calculating the *P-Domains*.

Again, as with the previously described approaches, both theorem proving and symbolic execution techniques are used in this test generation framework.

1.2.2 Code-based test case generation

In the last example of this section we will be describing a framework by Khurshid *et al* [4] that generates test cases not from an abstract model as the ones described in 1.2.2, but from Java code directly. We chose to take this detour from the main topic of this section in order to discuss a technique that:

- generates test cases from "real code" in a modern programming language;
- takes advantage of a model checker (Java PathFinder) to overcome some of the difficulties of symbolic execution;
- takes advantage of symbolic execution to overcome some of the difficulties of model checking.

Java PathFinder is a model checker built specifically for Java. As all model checkers, it allows verifying that a model of an application (or, in this case the application itself) satisfies a set of logic formulas specifying given properties of the application. An interesting property to be verified with Java PathFinder is for example that no exception is left unhandled in a given method. As a result of the model checking we can obtain either execution trace witnesses of the validity of the formulas or execution trace

counter-examples if the formulas do not hold. Clearly, witnesses are positive test cases and counter examples are negative ones.

There is a fundamental difference between this approach and the ones described before. In fact, all the previous frameworks were based on the fact that a model of the application, assumed correct, existed. The implementation could then be verified against that model. In the present case, the model does not exist explicitly: it is provided implicitly with the temporal logic formulas. The expected correct and incorrect behaviours of the implementation are described by the test engineer using temporal logic. The simple fact that the witnesses or counterexamples to these formulas exist already provides information about the correctness of the implementation.

One of the main issues around model checking software applications is the state space explosion problem. In order to be model checked efficiently, an application needs to be bounded on its input variables. Symbolic execution may help in this point, by replacing explicitly valued states by symbolic states representing large domains.

On the other hand, model checking provides a number of built-in facilities that allows exploring a state space efficiently. In particular, goodies like the handling of loops, recursion or method invocation can be hidden from the symbolic execution part. The handling of infinite execution trees is handled by the model checker by exploring the state space using either iterative deepening depth first or breadth first techniques. Heuristic based search is also supported.

In what concerns the technique itself, it requires that the Java code passes through a first instrumentation phase. Since Java PathFinder takes in pure Java code, the model checking is done over all possible values of input variables of the system. In order for the model checker to be able to manipulate symbols rather than real values the code needs to be instrumented. This is done at three levels:

- Basic typed variables (e.g. integers) are replaced by objects of an *Expression* type that will be able to keep track of symbolic values. Objects that are static or dynamic can be seen as compositions of the basic types and can thus be represented symbolically by replacing the basic typed fields with *Expression* objects;
- Code instrumentation is also necessary in order to build the *path condition* for each of the traces the model checker explores. To do this a

PathCondition class is provided that allows modifying the conditional statements of the code so that the path condition may be built as the application is executed;

- Finally, code instrumentation is used to add method pre and postconditions. Method preconditions are used to constrain the method's input values. This is relevant in order to constrain the search space and avoid execution traces that will never exist.

For each of the types of instrumentation described above, the framework provides the necessary Java libraries.

The most interesting aspect of this approach is the symbolic execution algorithm that allows dealing with methods that take as inputs complex unbounded data structures. This algorithm uses what the authors of [4] call *lazy initialization* since it initializes data structures as the they are accessed. In Fig. 4 the algorithm for lazy initialization is described.

```
if (f is uninitialized) {
    if (f is a reference field of type T) {
        nondeterministically initialize f to
            1. null
            2. a new object of class T (with uninitialized field values)
            3. an object created during a prior initialization of a field of type T
        if (method precondition is violated)
            backtrack();
    }
}
```

Figure 4: Lazy initialization algorithm

The algorithm allows the construction of path conditions that take into consideration not only conditions over basic types, but also over complex data structures involving a dynamic number of objects. Figure 4 only shows how the algorithm deals with initializing references to objects, being that primitives types are given symbolic values. The *backtrack()* instruction in the algorithm points out the fact that since the initialization of a reference is non-deterministic, the algorithm backtracks when the selected initialization is not allowed by the precondition of the method. It can then continue searching for other solutions at the last decision point.

Finally, the test cases are obtained by running the Java PathFinder model checker over the instrumented code. For each of the criteria specified in logic formulas, witnesses or counter-examples traces are generated. As in the previous approaches, the path conditions for these traces may then be used the build the actual input values to test the concrete system.

References

[1] S. Barbey C. Peraire and D. Buchs. Test selection for object-oriented software based on formal specifications. In *Programming Concepts and Methods - Proceedings of PROCOMET 98*, pages 385–403. Chapman and Hall, 1998.

[2] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. on Software Engineering*, SE-2(3):215–222, September 1976.

[3] W. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Trans. on Software Engineering*, SE-3(4):266–278, July 1977.

[4] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[5] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[6] B. Legeard and F. Peureux. Generation de sequences de tests a partir d'une specification b en plc ensembliste. In *Proc. Approches Formelles dans l'Assistance au developpement de Logiciels*, pages 113–130, June 2001.

[7] B. Legeard, F. Peureux, and M. Utting. Automated Boundary Testing from Z and B. In *Proceedings of the International Conference on Formal Methods Europe (FME'02)*, volume 2391 of *LNCS*, pages 21–40, Copenhagen, Denmark, July 2002. Springer-Verlag.

[8] Cecile Peraire. *Formal testing of object-oriented software: from the method to the tool*. PhD thesis, EPFL - Switzerland, 1998.

[9] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Scholl. Model-based test case generation for smart cards. In *In Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, 2003. to appear.

[10] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the automated generation of program test data. In *Proceedings: 2nd International Conference on Software Engineering*, page 636. IEEE Computer Society Press, 1976.