Constraint-Based Techniques for Software Testing

Nikolai Kosmatov
CEA LIST, Software Reliability Laboratory
91191 Gif-sur-Yvette France
Nikolay.Kosmatov@cea.fr

Abstract

In this chapter, we discuss some innovative applications of artificial intelligence techniques to software engineering, in particular, to automatic test generation. Automatic testing tools translate the program under test, or its model, and the test criterion, or the test objective, into constraints. Constraint solving allows then to find a solution of the constraint solving problem and to obtain test data.

We focus on two particular applications: model-based testing as an example of black-box testing, and all-paths test generation for C programs as a white-box testing strategy. Each application is illustrated by a running example showing how constraint-based methods allow to automatically generate test data for each strategy. We also give an overview of the main difficulties of constraint-based software testing and outline some directions for future research.

Keywords

constraint programming, software testing, automatic test generation

Introduction

Artificial intelligence (AI) techniques are successfully applied in various phases of software development life cycle. One of the most significant and innovative AI applications is using constraint-based techniques for automation of software testing.

Testing is nowadays the primary way to improve the reliability of software. Software testing accounts for about 50% of the total cost of software development (Ramler & Wolfmaier, 2006). Automated testing is aimed at reducing this cost. The increasing demand has motivated much research on automated software testing. Constraint-solving techniques are commonly used in software testing since 1990's. They were applied in the development of several automatic test generation tools.

The underlying idea of constraint-based test generators is to translate the program under test, or its model, and the test criterion, or the test objective, into constraints. Constraint solving allows then to find a solution of the constraint solving problem and to obtain test data. The constraint representation of the program, interaction with a constraint solver and the algorithm may be different in each particular tool and depend on its objectives and test coverage criteria.

While learning about constraint-based techniques for the first time, we are often surprised to see that one constraint solver can so efficiently solve so different problems. For example, such as the famous SEND + MORE = MONEY puzzle (Apt, 2003), SUDOKU puzzles, systems of linear equations and many others, where a human would use quite different and sometimes very tricky methods. The intelligence of modern constraint solvers is not only in their ability to solve problems, but also in their ability to solve quite different kinds of problems. Of course, some solvers may be more adapted for specific kinds of problems.

In this chapter, we will discuss some innovative applications of constraint-based techniques to software engineering, in particular, to automatic test generation. We will focus on two particular applications: model-based testing as an example of black-box testing, and all-paths test generation for C programs as a white-box testing strategy. Each application will be illustrated by a running example showing how constraint-based methods allow to automatically generate test data for each strategy. We will also mention the main difficulties of constraint-based software testing and outline some directions for future research.

Organization of the Chapter

The chapter is organized as follows. We start by a short background section on software testing and describe the most popular test coverage criteria. The section on model-based testing contains an overview of the approach, an example of formal model and application of AI techniques to this example. Next, the section on all-paths test generation presents the generation method, its advantages and possible applications. We finish by a brief description of future research directions and a conclusion.

Background

The classical book *The Art of Software Testing* by G. J. Myers defines software testing as "the process of executing a program with the intent of finding errors" (Myers, 1979, p.5). In modern software engineering, various testing strategies may be applied depending on the software development process, software requirements and test objectives. In *black-box testing* strategies, the software under test is considered as a black box, that is, test data are derived without any knowledge of the code or internal structure of the program. On the other hand, in *white-box testing*, the implementation code is examined for designing tests. Different testing strategies may be used together for improved software development. For example, one may first use black-box testing techniques for *functional testing* aimed at finding errors in the functionality of the software. Second, white-box testing may be applied to measure

the test coverage of the implementation code by the executed tests, and to improve it by adding more tests for non-covered parts.

Significant progress in software testing was done by applications of artificial intelligence techniques. Manual testing being very laborious and expensive, automation of software testing was the focus of much research since 1970's. Symbolic execution was first used in software testing in 1976 by L. A. Clarke (Clarke, 1976) and J. C. King (King, 1976). Automatic constraint-based testing was proposed by R. A. DeMillo and A. J. Offutt (DeMillo & Offutt, 1991). Since then, constraint-based techniques were applied for development of many automatic test generation tools. Like in manual testing, various *test coverage (test selection) criteria* may be used to control automatic test generation and to evaluate the coverage of a given set of test cases. The possibility to express such criteria in constraints is the key property which allows to apply AI techniques and to automatically generate test cases. Let us briefly discuss coverage criteria used in model-based testing, where the criteria are applied to a formal model of the software under test. They can be classified into several families.

Structural coverage criteria exploit the structure of the model and contain several sub-families. *Control-flow-oriented coverage criteria* focus on the control-flow aspects, such as the nodes and the edges, the conditional statements or the paths. Some examples for this family include the *all-statements criterion* (every reachable statement must be executed by some test case), the *all-branches criterion* (every reachable edge must be executed) and the *all-paths criterion* (every feasible path must be executed). *Transition-based coverage criteria* focus on transitions and include, for example, the *all-transition-pairs* and the *all-loop-free-paths coverage criteria*. *Data-flow-oriented coverage criteria* concentrate on the data-flow information such as *definitions* of variables (where a value is assigned to a variable) and their *uses* (i.e. expressions in which this value is used). So, the *all-definitions criterion* requires to test at least one possible path from each definition of a variable to one of its possible uses. The *all-uses criterion* states that at least one path from each definition of a variable to each of its possible uses must be tested. The *all-def-use-paths criterion* requires to test each possible path from each definition of a variable to each of its possible uses.

The second big family of criteria contains *data coverage criteria* used to choose a few test cases from a large data space. *Statistical data coverage* requires some statistical properties for the chosen test cases (e.g. respecting a given statistical distribution), whereas *boundary coverage* looks for test cases situated on the boundary of the data space. Several *boundary coverage criteria* were formalized in (Kosmatov, Legeard, Peureux, & Utting, 2004). Some of them may be formulated as optimization problems (e.g. minimization of a cost function on a given data space), and are suitable for constraint solvers. A data coverage criterion may be applied in combination with a structural coverage criterion as follows: for each test target required by the structural criterion (e.g. statement, branch or path), the choice of test cases covering this target must satisfy the data coverage criterion.

Fault-based coverage criteria are aimed at detecting certain types of frequent faults in the SUT. For example, *mutation coverage* checks if test cases efficiently detect errors in *model mutants*, that is, erroneous versions of the original model obtained from it by introducing

mutations. Some examples of mutations are confusions between $>, \geq, <, \leq, =$ and $\neq$, or between operations $+$ and $-$, which are considered as frequent faults.

Similar criteria may be used in white-box testing strategies, where they are applied directly to the program code rather than to its model. The reader will find a good general overview of test coverage criteria in (Zhu, Hall, & May, 1997). A. P. Mathur's book (Mathur, 2008) describes modern techniques of software testing. An introduction to constraint programming techniques may be found in K. Apt's book (Apt, 2003).

Model-Based Testing

Overview and Benefits of the Approach

Model-based testing is a variant of testing that relies on an explicit model describing the desired behavior of the system under test (SUT). The model is based on the requirements or existing specification documents for the SUT. It is also called *a formal model* or *an abstract model* since it formalizes an informal specification of the SUT and usually focuses on its most important aspects without describing all implementation details.

Various notations may be used to write models. *State-based or Pre/Post notations* (such as B, Z, VDM, JML) model the SUT by a collection of state variables, which define the current state of the system, and some operations, modifying the variables. The operations are usually described by a precondition, which must be verified before executing the operation, and a postcondition, defining the result of its execution.

Transition-based notations include finite-state machines, labeled transition systems, statecharts, UML State Machines, etc. They focus on the transitions between different states of the SUT. They are based on graphical representation, where the nodes represent the states and the edges represent the operations.

Operational notations are used to describe distributed systems and communication protocols. They include process algebras and Petri nets. *Data-flow notations*, such as Lustre, focus on the data rather than on the control flow. A complete classification of model notations can be found in (van Lamsweerde, 2000).

After writing and validating the model and choosing a test coverage criterion, an automatic model-based testing tool is used to generate test cases. *A test case* includes input data and expected behavior of the system, also called *an oracle*. In model-based testing, as its name suggests, test cases are derived from the model, without any knowledge of the implementation's source code, so this approach is part of *black-box testing.* Next, test cases are transformed into executable tests, executed on the SUT, and the results are analyzed. The reader will find more on automated test execution in the book (Mosley & Posey, 2002).

Although creating an explicit model of the system under test is an additional task unnecessary for manual testing, the effort is recompensed by the benefits of model-based testing. Its main advantage is the possibility of automation. A model-based testing tool uses constraint-based techniques to transform the test generation problem for a given model with a chosen coverage criterion into a number of constraint problems (test targets), and calls a constraint

solver to find a solution for each problem. Each solution provides one test input and is completed by a test preamble and an oracle to obtain a test case. Some test targets may be unreachable: their constraints are incompatible and describe a situation that can never occur according to the model. It can happen because the chosen coverage criterion does not take into account the properties of a particular model. For example, the all-transition-pairs coverage criterion requires to activate every pair of adjacent transitions at least once, but the constraint solver will not find a solution for a pair of transitions (t_i, t_j) if t_j can never be executed after t_i in the model (e.g. when the precondition of t_j is incompatible with the postcondition of t_i). The example in the next section will illustrate how constraint-based techniques are used in model-based testing tools for automatic test generation.

To finish our overview of model-based testing, let us cite other benefits of this approach. Systematic and repeatable automated model-based testing reduces testing cost and time. It makes it easier to handle requirements evolution by modifying the model and regenerating test cases. Unlike in manual testing, the quality of generated test cases is not so much dependent on the competence and experience of the test engineer. The engineer may interact with the model-based testing tool to focus the generation on some particular part of the model, specify some special kinds of test cases, control their number and the desired coverage criteria. Writing a model of the SUT also allows to detect possible errors in the requirements and encourages early modeling. We refer the reader to the book (Utting & Legeard, 2006) for a practical guide to the process of model-based testing, its benefits and several applications.

An Example: Formal Model of a Process Scheduler

To show how constraint-based techniques are used in model-based testing tools we consider an example studied in (Legeard, Peureux, & Utting, 2002). Figure 1 shows a formal model written in B notation, also called *a B abstract machine*. It contains state variables, operations and invariant properties that should be satisfied at each moment. For convenience of the reader, specific B notation for set operations and relations is replaced here by the common mathematical notation.

This model describes a simplified uniprocessor scheduler. At every moment, at most one process may be executed by the processor. When several processes are ready to be executed, they are scheduled in turn one after another. The constant PID is the set of all possible process identifiers. The variable *active* represents the set containing the active process, or the empty set when no process is executed (on a real processor, the system idle process is executed). The variable *ready* represents the set of processes ready to be scheduled. The variable *waiting* contains the set of processes not ready to be scheduled (e.g. waiting for a disk read request, an input, an authorization to continue or some other event). The sets *active*, *waiting* and *ready* must be always disjoint and are initially empty. Their union is the set of currently existing processes in the system. The last line in the invariant states that the system may have no active process only if no process is ready to be executed. In this example, the system idle process is not modeled, and only four processes with three operations are considered. The operation **new** adds a process not yet existing in the system into *waiting*. The operation **ready** is executed when some waiting process gets ready, or is

MACHINE
  SCHEDULER
SETS
  $PID = \{p1, p2, p3, p4\}$
VARIABLES
  $active, waiting, ready$
INVARIANT
  $active \subseteq PID \wedge$
  $waiting \subseteq PID \wedge$
  $ready \subseteq PID \wedge$
  $\text{card}(active) \leq 1 \wedge$
  $active \cap waiting = \emptyset \wedge$
  $waiting \cap ready = \emptyset \wedge$
  $active \cap ready = \emptyset \wedge$
  $(active = \emptyset) \implies (ready = \emptyset)$

INITIALIZATION
  $active, waiting, ready := \emptyset, \emptyset, \emptyset$

OPERATIONS
  new(pp)=
    PRE
      $pp \in PID \wedge$
      $pp \notin (active \cup waiting \cup ready)$
    THEN
      $waiting := (waiting \cup \{pp\})$
    END;

  ready(pp)=
    PRE
      $pp \in waiting$
    THEN
      $waiting := (waiting \setminus \{pp\})$ ||
      IF $(active = \emptyset)$ THEN
        $active := \{pp\}$
      ELSE
        $ready := ready \cup \{pp\}$
      END
    END;
  swap=
    PRE
      $active \neq \emptyset$
    THEN
      $waiting := waiting \cup active$ ||
      IF $(ready = \emptyset)$ THEN
        $active := \emptyset$
      ELSE
        ANY pp
        WHERE $pp \in ready$ THEN
          $active := \{pp\}$
          $ready := ready \setminus \{pp\}$
        END
      END
    END
END;

Figure 1: B model of a process scheduler

allowed, to continue execution: this process is moved into *active* if no process is active at this moment, or into *ready* otherwise. To ensure that all ready processes are executed in turn, the operation **swap** is called regularly to move the active process into *waiting* and, if there are some ready processes, to move one of them into *active*. A schema of possible process moves and operations is shown in Figure 2. This model was validated using AtelierB, the well-formedness conditions (e.g. respecting the invariant) were generated and proved automatically.

Constraint Solving in Model-Based Testing

Let us now illustrate on the model of Figure 1 how constraint-based techniques are used in model-based testing tools, such as the academic tool BZTT (Ambert et al., 2002) or its industrial version for UML models Test Designer (Smartesting, 2008). First, some test

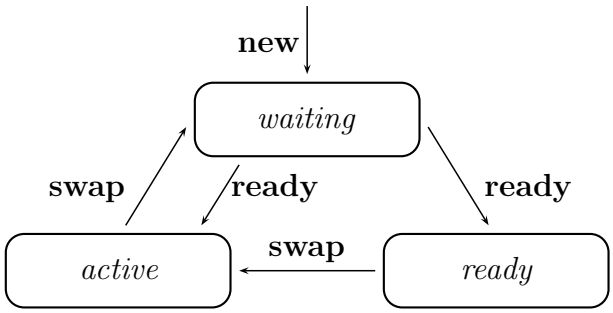

Figure 2: Schema of process moves in the scheduler model

coverage criteria must be selected for test generation. For our example, we choose the all-branches criterion of structural coverage combined with the boundary coverage criterion which requires for each test target to choose two boundary test cases using the min/max heuristic with the cost function f

$$f = \text{card}(active) + \text{card}(waiting) + \text{card}(ready).$$

In other words, the cost function should achieve its maximum and minimum on some test cases for each target. So, two test cases are in general necessary for a reachable test target (or just one if $\max(f)$=$\min(f)$ for the target).

The test generation session proceeds as follows.

> (Step 1) First, the model is parsed and the operations are translated into a set of *Before-After predicates* (also called *Pre/Post, or effect predicates*). Each operation may be decomposed into several simpler predicates, for example, using the disjunctive normal form of the model. The degree of decomposition depends on the desired coverage.

Let $\mathfrak{I}$ denote the invariant of the model. Figure 3 shows the generated Before/After predicates. Their Before parts form a partition of the original operations' preconditions. The After part defines the new values of state variables, where the new value of a variable v after the operation is denoted by v'. The operation **new** is expressed by P_1, **ready** is decomposed into P_2 and P_3, and **swap** into P_4 and P_5.

> (Step 2) Next, each test target that must be covered according to the selected criteria is expressed as a constraint problem. A constraint solver is called to find a solution for the problem.

Test targets and the way the constraint solver is applied to solve them depend on the coverage criteria. The all-branches criterion requires to test each edge in the operations. In our example, we need to cover the test targets $TG_i = Before_i$ $(1 \leq i \leq 5)$. Moreover, the combination with the chosen boundary coverage criterion requires to find for each TG_i one

P_i	$Before_i$	$After_i$
P_1	$\mathcal{I} \wedge pp \in PID \wedge$ $pp \notin (active \cup waiting \cup ready)$	$active' := active \wedge$ $waiting' := waiting \cup \{pp\} \wedge$ $ready' := ready$
P_2	$\mathcal{I} \wedge pp \in waiting \wedge active = \emptyset$	$active' := \{pp\} \wedge$ $waiting' := waiting \setminus \{pp\} \wedge$ $ready' := ready$
P_3	$\mathcal{I} \wedge pp \in waiting \wedge active \neq \emptyset$	$active' := active \wedge$ $waiting' := waiting \setminus \{pp\} \wedge$ $ready' := ready \cup \{pp\}$
P_4	$\mathcal{I} \wedge active \neq \emptyset \wedge ready = \emptyset$	$active' := \emptyset \wedge$ $waiting' := waiting \cup active \wedge$ $ready' := ready$
P_5	$\mathcal{I} \wedge active \neq \emptyset \wedge pp \in ready$	$active' := \{pp\} \wedge$ $waiting' := waiting \cup active \wedge$ $ready' := ready \setminus \{pp\}$

Figure 3: Before/After predicates for the scheduler model

solution to minimize f and another one to maximize f, so the five test targets TG_i should be split into ten more specific test targets:

$$TG_i^{min} = Before_i \wedge (f \rightarrow min) \quad \text{and} \quad TG_i^{max} = Before_i \wedge (f \rightarrow max).$$

Figure 4 shows possible solutions that might be generated for these targets by a constraint solver with sets like that of the BZTT tool. The figure contains the system state S_k from which the operation B_k (*the test body*) must be executed. The expected result of the execution (*the oracle*) obviously follows from the Before/After predicates of Figure 3 and is not shown here.

(Step 3) Next, test preambles are generated and final test cases are created.

Notice that Figure 4 does not contain complete test cases. Indeed, to execute the operation B_k on the state S_k, we first have to get the SUT into the correct state. It is done by *a test preamble,* i.e. a sequence of operation calls prior to the test body execution. Again, constraint-based algorithms may be used to automatically find test preambles for generated test data. For example, the preamble for the state S_1 is empty since S_1 is the initial state of the SUT. A possible preamble for S_2 is $\mathbf{new}(p1), \mathbf{new}(p2), \mathbf{ready}(p1), \mathbf{new}(p3)$. Test preambles for the other S_k are computed similarly.

(Step 4) Finally, the test cases are executed on the SUT. The results are analyzed.

k	Target	System state S_k	Test body B_k
1	TG_1^{min}	$active = \emptyset,\ waiting = \emptyset,\ ready = \emptyset$	**new**$(p1)$
2	TG_1^{max}	$active = \{p1\},\ waiting = \{p2, p3\},\ ready = \emptyset$	**new**$(p4)$
3	TG_2^{min}	$active = \emptyset,\ waiting = \{p1\},\ ready = \emptyset$	**ready**$(p1)$
4	TG_2^{max}	$active = \emptyset,\ waiting = \{p1, p2, p3, p4\},\ ready = \emptyset$	**ready**$(p1)$
5	TG_3^{min}	$active = \{p1\},\ waiting = \{p2\},\ ready = \emptyset$	**ready**$(p2)$
6	TG_3^{max}	$active = \{p1\},\ waiting = \{p2, p3, p4\},\ ready = \emptyset$	**ready**$(p2)$
7	TG_4^{min}	$active = \{p1\},\ waiting = \emptyset,\ ready = \emptyset$	**swap**
8	TG_4^{max}	$active = \{p1\},\ waiting = \{p2, p3, p4\},\ ready = \emptyset$	**swap**
9	TG_5^{min}	$active = \{p1\},\ waiting = \emptyset,\ ready = \{p2\}$	**swap**
10	TG_5^{max}	$active = \{p1\},\ waiting = \{p2, p3\},\ ready = \{p4\}$	**swap**

Figure 4: Solutions for test targets TG_i^{min}, TG_i^{max} generated by constraint solving

In practice, a test case of a complete test suite may also contain an observation step (describing how to observe the result on the SUT) and a postamble (resetting the SUT to some state allowing the execution of the following test case). Constraint-based techniques may be also used for postamble computation. We do not model observation and reset operations in our example and do not detail these steps.

All-Paths Test Generation for C Programs

Description of the Method

In the previous section, we have seen how constraint-based techniques are used for test generation in model-based testing, a strategy of black-box testing. This section discusses an advanced technique of white-box testing, where an ingenious combination of constraint logic programming, constraint-based symbolic execution and concrete execution allows to generate test cases for a C function for the all-paths coverage criterion. This technique (with various modifications) was used in several testing tools: PathCrawler developed at CEA LIST (Williams, Marre, & Mouy, 2004; Williams, Marre, Mouy, & Roger, 2005), DART (Godefroid, Klarlund, & Sen, 2005), CUTE (Sen, Marinov, & Agha, 2005), SimC (Xu & Zhang, 2006) and EXE (Cadar, Ganesh, Pawlowski, Dill, & Engler, 2006).

In practice, all-paths coverage can be unrealistic for some programs. For example, if the number of iterations of some loop is determined by an input variable, the all-paths criterion requires exhaustive testing for this input variable. That is why path-oriented testing tools propose weaker versions of this criterion. For instance, given a parameter n, CUTE looks for different paths with a difference in the first n instructions, while PathCrawler can be restricted to paths in which every loop has at most n consecutive iterations. The generation methods for these weaker criteria are easily obtained by slight modifications of the all-paths test generation method.

We will use as a running example the C function shown in Figure 5. The function `min3`

```
1   //returns minimum in given three-element array a
2   int min3(int a[3]){
3     int min=a[0];
4     if( min > a[1] )
5       min=a[1];
6     if( min > a[2] )
7       min=a[2];
8     return min; }
```

Figure 5: Function min3 returning the minimum in array a

takes one parameter, an array `a` of three integers, and returns the minimal value in the array. To simplify the example, we restrict the domain of elements of `a` to $[0, 10]$.

For test case generation, the user needs to define a *precondition*, i.e. the conditions on the program's input for which the behavior is defined. Here, the precondition contains the definition of the variables' domains. The user can also provide an oracle function to check on-the-fly, during the concrete execution of every generated test case on the instrumented code, whether the observed behavior of the program is correct. We assume the oracle is provided, and focus on the generation of test data.

A decision is denoted by the line number of the condition followed by a "+" if the condition is true, and by a "−" otherwise. We can denote an execution path by a sequence of line numbers, e.g. $3, 4^-, 6^+, 7, 8$. The mark "$\star$" after a condition indicates that the other branch has already been explored (it will be explained in detail below).

Let us now describe a simplified version of the PathCrawler method (following the presentation in (Kosmatov, 2008)). It needs an instrumented version of the program under test to trace the execution path. The main loop in the PathCrawler method is rather simple. Given a partial program path π, also called below *a path prefix*, the main idea is to symbolically execute it in constraints. PathCrawler uses COLIBRI, an efficient constraint solver developped at CEA LIST and shared with two other testing tools: GATeL (Marre & Arnould, 2000) and OSMOSE (Bardin & Herrmann, 2008). A solution of the resulting constraint solving problem will provide a test case exercising a path starting by the prefix π.

Then the trick is to use concrete execution of the test case on the instrumented version to obtain the complete path. The path prefixes are explored in a depth-first search.

To symbolically execute a program in constraints, the PathCrawler tool maintains:

- a memory map that represents the program memory state at every moment of symbolic execution. It can be seen as a mapping which associates a value to a symbolic name. The symbolic name may be a variable name or an array element. The value may be a constant or a logical variable.

- current path prefix π in the program under test. When a test case is successfully generated for the prefix π, the remaining part of the path it activates is denoted σ.

- a constraint store containing the constraints added during the symbolic execution of the current prefix π.

The method contains the following steps:

> Initialization: Create a logical variable for each input and associate it with the input. Set initial values of initialized variables. Add constraints for the precondition. Let the initial prefix π be empty. Continue to (Step 1).
>
> (Step 1) Let σ be empty. Execute symbolically the path π, that is, add constraints and update the memory according to the instructions in π. If some constraint fails, continue to (Step 4). Otherwise, continue to (Step 2).
>
> (Step 2) Call the constraint solver to generate a test case, that is, concrete values for the inputs, satisfying the current constraints. If it fails, go to (Step 4). Otherwise, continue to (Step 3).
>
> (Step 3) Run traced execution of the program on the test case generated in the previous step to obtain the complete execution path. The complete path must start by π. Save the remaining part into σ. Continue to (Step 4).
>
> (Step 4) Let ρ be the concatenation of π and σ. Try to find in ρ the last unmarked decision, i.e. the last decision without a "$\star$" mark. If ρ contains no unmarked decision, exit. Otherwise, if $x^{\pm}$ is the last unmarked decision in ρ, set π to the subpath of ρ before $x^{\pm}$, followed by $x_{\star}^{\mp}$ (i.e. the negation of $x^{\pm}$ marked as already processed), and continue to (Step 1).

Notice that Step 4 chooses the next path prefix in a depth-first search. It changes the last unmarked decision in ρ to look for differences as deep as possible first, and marks a decision by a "$\star$" when its negation (i.e. the other branch from this node in the tree of all execution paths) has already been fully explored. For example, if $\rho = a_{\star}^{-}, b, c^{+}, d, e_{\star}^{+}, f$, the last unmarked decision is c^{+}, so we take the subpath of ρ before this decision $a_{\star}^{-}, b$, and add $c_{\star}^{-}$ to obtain the new prefix $\pi = a_{\star}^{-}, b, c_{\star}^{-}$.

Test Generation for min3

We apply this method to our example and show in Figure 6 how it proceeds. In this figure, $\mapsto$ indicates the memory mapping, $\rightsquigarrow$ denotes the application of Step 2 and Step 3, and $\rightarrow$ the application of Step 4 and Step 1. The empty path is denoted by ϵ.

In the state (1) in Figure 6, we see that the initialization step associates a logical variable to each input, i.e. to each element of `a`, and posts the precondition $\langle pre \rangle$ to the constraint store. Here, $\langle pre \rangle$ denotes the constraints:

$$X_0 \in [0, 10], \quad X_1 \in [0, 10], \quad X_2 \in [0, 10].$$

As the original prefix π is empty, Step 1 is trivial and adds no constraints. Step 2 chooses a first test case. It can be shown that this choice is not important for a complete depth-first

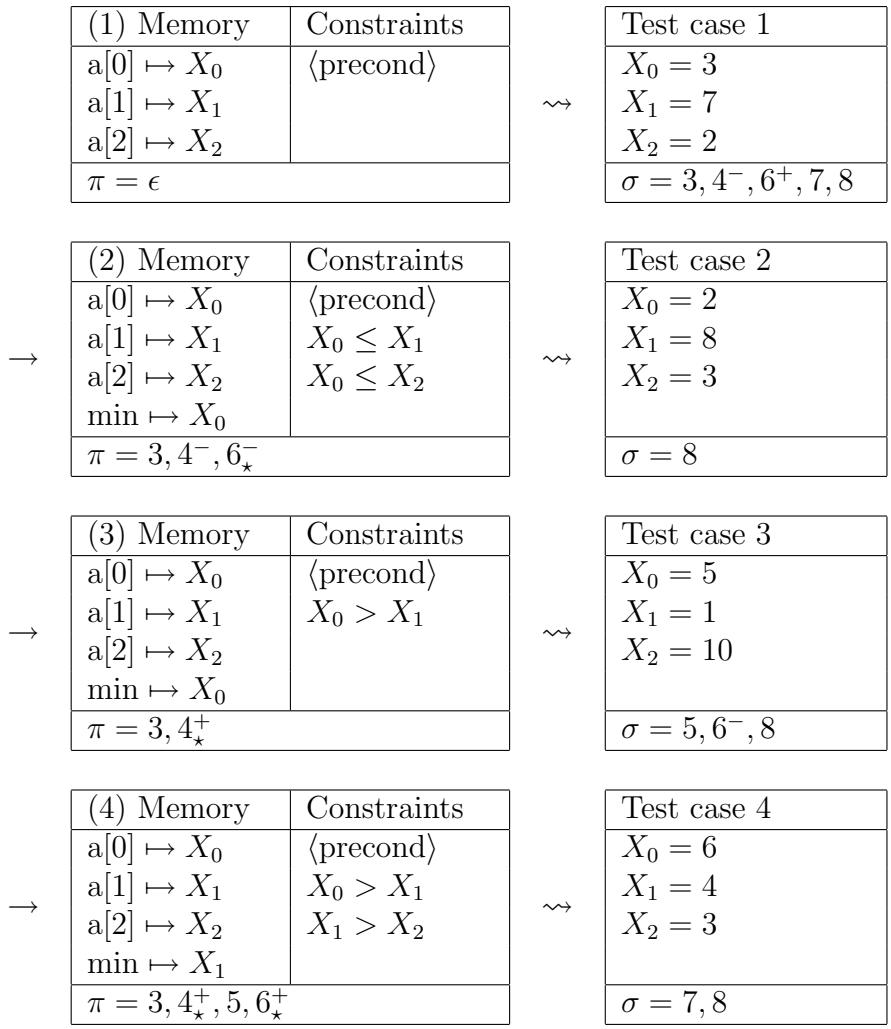

Figure 6: Depth-first generation of all-paths tests for the function min3 of Figure 5

search, so we use random generation here. Some solvers may follow deterministic strategies, e.g. minimal values first. In Step 3, we retrieve the complete path traced during the concrete execution of Test case 1, and obtain $\sigma = 3, 4^-, 6^+, 7, 8$.

Step 4 sets $\rho = 3, 4^-, 6^+, 7, 8$ and, therefore, the new path prefix $\pi = 3, 4^-, 6^-_\star$ by negating the last not-yet-negated decision. Now, Step 1 symbolically executes this path prefix in constraints for unknown inputs, and the resulting state is shown in (2). Let us explain this execution in detail. First, the execution of the assignment 3 adds $\texttt{min} \mapsto X_0$ into the memory since X_0 is the current value of a[0]. The execution of the decision 4^- adds the constraint $X_0 \leq X_1$ after replacing the variable `min` by its current value in the memory map X_0. Similarly, the execution of $6^-_\star$ adds the constraint $X_0 \leq X_2$.

During symbolic execution, evaluation routines are called each time when it is necessary to find the current value of an expression (r-value) or the correct symbolic name of the variable being assigned (l-value). The evaluation of complex expressions may introduce additional

logical variables and constraints. For instance, if we had an assignment `z = a[0]+5*a[2]`, its symbolic execution now would create two new logical variables Y and Z, add $\mathtt{z} \mapsto Z$ to the memory map and post two new constraints: $Y = 5X_2$ and $Z = X_0 + Y$.

Next, Step 2 generates Test case 2, and Step 3 executes it and finds $\sigma = 8$. We are now going from (2) and Test case 2 to (3) in Figure 6. Step 4 computes the complete path $\rho = 3, 4^-, 6^-_\star, 8$. As $6^-_\star$ means that its negation has already been explored, the new prefix π is $3, 4^+_\star$. Step 1 symbolically executes this partial path as shown in (3).

Next, Step 2 generates Test case 3. Step 3 finds $\sigma = 5, 6^-, 8$. We are now moving from (3) and Test case 3 to (4) in Figure 6. Step 4 computes the new prefix $\pi = 3, 4^+_\star, 5, 6^+_\star$. Step 1 executes π symbolically and updates the memory state and the constraint store as shown in (4). By Step 2 and Step 3, we obtain Test case 4 and the new path end $\sigma = 7, 8$. Finally, Step 4 exits since the whole path $\rho = 3, 4^+_\star, 5, 6^+_\star, 7, 8$ does not have any unmarked decision. In other words, all the paths have been explored.

Advantages and Applications of All-Paths Testing

The presented method of all-paths test generation mixing symbolic and concrete execution has the following benefits:

- **Soundness.** Concrete execution of the generated test cases on the instrumented code allows to check that each test case really executes the path for which it was generated.
- **Completeness.** If the program has finitely many paths (in particular, all loops are bounded, as it is often required in critical software), depth-first search allows to iterate over all paths of the program. However, this property can be achieved in practice on a program only when symbolic execution of all features of the program is correct and when constraint solving for its paths terminates within a reasonable timeout.
- **Incrementality.** Depth-first search allows us to reuse as much as possible the results of symbolic execution. Each instruction of any given path prefix is executed exactly once, independently of how many paths start by this prefix. This encourages the use of constraint logic programming, which offers backtracking.
- **Fast entry.** Concrete execution of instrumented code permits to quickly deduce a complete feasible path in the program.

All these qualities make this method one of the most scalable test generation methods until now. Moreover, its applications are not limited to software testing of C programs.

PathCrawler has been adapted by (Williams, 2005) to measure the worst-case execution time (WCET). While static analysis is often used to find an upper bound of the WCET, the PathCrawler method with specific heuristics may be used to find and to execute a set of maximal paths with respect to a partial order on paths, and to obtain a close lower bound for the WCET.

A similar technique of all-paths testing is used by the OSMOSE testing tool developed at CEA LIST, which allows to generate test cases based on the binary code only (Bardin & Herrmann, 2008). Binary code testing is very challenging in software engineering. For instance, source code offers syntax to locate jump targets while binary code does not. Because of dynamic jumps, i.e. jumps to a location which must be computed, such tools need to guess possible targets.

Recent research suggested that path-oriented testing can be also used in combination with static analysis techniques (Kröning, Groce, & Clarke, 2004; Yorsh, Ball, & Sagiv, 2006; Gulavani, Henzinger, Kannan, Nori, & Rajamani, 2006). For example, SYNERGY (Gulavani et al., 2006) simultaneously looks for bugs and proofs by combining PathCrawler-like testing and model checking, and takes advantage of information obtained by one technique for the other. Tests give valuable information for refinement of abstractions used in model checking, and therefore contribute to the formal proof.

Future Research Directions

Software testing continues to offer new challenges for artifitial intelligence. Possible NP-hardness (respectively, undecidability) of satisfiability for constraint problems with a finite (respectively, infinite) number of potential solutions are inherent difficulties of artificial intelligence problems. They make it impossible to find efficient algorithms in some cases. Nevertheless, specific search heuristics and propagation techniques working well in practice should be identified. We believe that future research in automatic constraint-based testing will be centered along three main axes:

1. improving the representation of programs in constraints,
2. developing more efficient constraint solving techniques,
3. looking for new applications.

Constraint-based symbolic execution is often imperfect. An appropriate representation and efficient algorithms must be found for domains which are not fully supported today by the existing testing tools. For instance, the semantics of operations on *floating-point numbers* often depends on the language, compiler and actual machine architecture, and is difficult to be correctly modeled in constraints (Botella, Gotlieb, & Michel, 2006). Another example is *sequences,* used in models to represent finite lists of elements such as stacks, queues, communication channels, sequences of transitions or any other data with consecutive access to elements. On the borderline of decidability, this data type also requires specific constraint solving techniques and their integration into existing constraint solvers (Kosmatov, 2006). *Aliasing problems* appear during constraint-based symbolic execution with unknown inputs when the actual memory location of a variable value is uncertain. They continue to be a very challenging research area in software testing (Visvanathan & Gupta, 2002; Kosmatov, 2008).

Despite the increasing performances of modern computers, the combinatorial explosion and slowness of constraint solving are still important obstacles to wider application of constraint-based techniques in software engineering. In goal-oriented test generation, (Gotlieb, Botella, & Rueher, 1998) proposes to represent in constraints a whole program, rather than just one path, by modeling conditional and loop instructions by specific constraints. Among the most recent approaches to the path explosion problem in all-paths testing, CUTE (Sen et al., 2005) proposes to approximate function return values and pointer constraints by concrete values, but it makes the search incomplete. Path exploration can be guided by particular heuristics (Cadar et al., 2006), or using a combination of random testing and symbolic execution (Majumdar & Sen, 2007). SMART (Godefroid, 2007) suggests to create on-the-fly function summaries to limit path explosion. (Mouy, Marre, Willams, & Le Gall, 2008) proposes to use a specification of a called function rather than its code while testing the calling function. State-caching, a technique arising from static analysis, is used by (Boonstoppel, Cadar, & Engler, 2008) to prune the paths which are not interesting with respect to given test objectives.

Improved test generation algorithms and larger support of various program features should allow to expand applications of constraint-based methods to new areas of software testing, and more generally, in software engineering. Model-based testing, focused today mostly on functional testing, should spread to other kinds of testing, such as security testing, robustness testing and performance testing. Some new applications of constraint-based path exploration in software engineering were mentioned in the previous section.

Recent techniques are often difficult to objectively evaluate and compare because they are developed for different areas and/or tested on different benchmarks. More comparative studies and testing-tool competitions should be conducted to improve our knowledge of the efficiency of different algorithms, heuristics, solving strategies and modeling paradigms.

Conclusion

In this chapter, we gave an overview on the use of artificial intelligence techniques for automation of software testing. We presented two of the most innovative strategies of automatic constraint-based test generation: model-based testing from a formal model written in a state-based notation, and all-paths testing of C programs using symbolic execution. Each method was illustrated by an example showing step-by-step how automatic testing tools use constraint-based techniques to generate tests.

The idea to apply artificial intelligence techniques to software testing was revolutionary in software engineering. It allowed the development of several automatic test generation methods. Extremely expensive and laborious manual testing is more and more often accompanied, or even replaced, by automatic testing. Constraint-based test generation is used nowadays for testing various types of software with different coverage criteria, and will certainly become more and more popular in the future.

Acknowledgments

The author would like to thank Mickaël Delahaye for many valuable ideas during the preparation of an earlier version of the chapter, as well as Sébastien Bardin, Bernard Botella, Arnaud Gotlieb, Philippe Herrmann, Bruno Legeard, Bruno Marre and Nicky Williams for their comments and/or useful discussions.

References

Ambert, F., Bouquet, F., Chemin, S., Guenaud, S., Legeard, B., Peureux, F., et al. (2002). BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. In *Formal Approaches to Testing of Software Workshop (FATES'02) at CONCUR'02* (pp. 105–120). Brnö, Czech Republic.

Apt, K. (2003). *Principles of constraint programming*. Cambridge University Press.

Bardin, S., & Herrmann, P. (2008). Structural testing of executables. In *the First IEEE International Conference on Software Testing, Verification, and Validation (ICST'08)* (p. 22-31). Lillehammer, Norway: IEEE Computer Society.

Boonstoppel, P., Cadar, C., & Engler, D. R. (2008). RWset: attacking path explosion in constraint-based test generation. In *the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08), Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'08)* (pp. 351–366). Budapest, Hungary: Springer.

Botella, B., Gotlieb, A., & Michel, C. (2006). Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, *16*(2), 97–121.

Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., & Engler, D. R. (2006). EXE: automatically generating inputs of death. In *the 13th ACM Conference on Computer and Communications Security (CCS'06)* (pp. 322–335). Alexandria, Virginia, USA: ACM.

Clarke, L. A. (1976). A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, *2*(3), 215–222.

DeMillo, R. A., & Offutt, A. J. (1991). Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, *17*(9), 900–910.

Godefroid, P. (2007). Compositional dynamic test generation. In *the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)* (pp. 47–54). Nice, France: ACM.

Godefroid, P., Klarlund, N., & Sen, K. (2005). DART: Directed automated random testing. In *the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)* (pp. 213–223). Chicago, IL, USA: ACM.

Gotlieb, A., Botella, B., & Rueher, M. (1998). Automatic test data generation using constraint solving techniques. In *the ACM SIGSOFT 1998 International Symposium on Software Testing and Analysis (ISSTA'98)* (pp. 53–62). Clearwater Beach, Florida, USA: ACM.

Gulavani, B. S., Henzinger, T. A., Kannan, Y., Nori, A. V., & Rajamani, S. K. (2006). SYNERGY: a new algorithm for property checking. In *the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'05)* (pp. 117–127). Portland, Oregon, USA: ACM.

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, *19*(7), 385–394.

Kosmatov, N. (2006). A constraint solver for sequences and its applications. In *the 21st Annual ACM Symposium on Applied Computing (SAC'06)* (pp. 404–408). Dijon, France: ACM.

Kosmatov, N. (2008). All-paths test generation for programs with internal aliases. In *the 19th IEEE International Symposium on Software Reliability Engineering (ISSRE'08)* (pp. 147–156). Redmond, WA, USA: IEEE Computer Society.

Kosmatov, N., Legeard, B., Peureux, F., & Utting, M. (2004). Boundary coverage criteria for test generation from formal models. In *the 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04)* (pp. 139–150). Saint-Malo, France: IEEE Computer Society.

Kröning, D., Groce, A., & Clarke, E. M. (2004). Counterexample guided abstraction refinement via program execution. In *the 6th International Conference on Formal Engineering Methods (ICFEM'04)* (pp. 224–238). Seattle, WA, USA: Springer.

Legeard, B., Peureux, F., & Utting, M. (2002). Automated boundary testing from Z and B. In *the International Confefence on Formal Methods Europe (FME'02)* (pp. 21–40). Copenhaguen, Denmark: Springer.

Majumdar, R., & Sen, K. (2007). Hybrid concolic testing. In *the 29th International Conference on Software Engineering (ICSE'07)* (pp. 416–426). Minneapolis, MN, USA: IEEE Computer Society.

Marre, B., & Arnould, A. (2000). Test sequences generation from Lustre descriptions : GATeL. In *the 15th IEEE International Conference on Automated Software Engineering (ASE'00)* (pp. 229–237). Grenoble, France: IEEE Computer Society.

Mathur, A. P. (2008). *Foundations of software testing*. Pearson Editions.

Mosley, D. J., & Posey, B. A. (2002). *Just enough software test automation*. Prentice Hall PTR.

Mouy, P., Marre, B., Willams, N., & Le Gall, P. (2008). Generation of all-paths unit test with function calls. In *the 2008 IEEE International Conference on Software Testing, Verification, and Validation (ICST'08)* (pp. 32–41). Washington, DC, USA: IEEE Computer Society.

Myers, G. J. (1979). *The art of software testing*. John Wiley and Sons.

Ramler, R., & Wolfmaier, K. (2006). Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *the 2006 International Workshop on Automation of Software Test (AST'06)* (pp. 85–91). Shanghai, China: ACM.

Sen, K., Marinov, D., & Agha, G. (2005). CUTE: a concolic unit testing engine for C. In *the 5th joint meeting of the European Software Engineering Conference and ACM*

SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05) (pp. 263–272). Lisbon, Portugal: ACM.

Smartesting. (2008). *The Test Designer tool.* http://www.smartesting.com/.

Utting, M., & Legeard, B. (2006). *Practical model-based testing - a tools approach.* Elsevier Science.

van Lamsweerde, A. (2000). Formal specification: a roadmap. In *the 22nd International Conference on Software Engineering, Future of Software Engineering Track (ICSE'00)* (pp. 147–159). Limerick, Ireland.

Visvanathan, S., & Gupta, N. (2002). Generating test data for functions with pointer inputs. In *the 17th IEEE International Conference on Automated Software Engineering (ASE'02)* (p. 149). Edinburgh, Scotland, UK: IEEE Computer Society.

Williams, N. (2005). WCET measurement using modified path testing. In *the 5th International Workshop on Worst-Case Execution Time Analysis (WCET'05).* Palma de Mallorca, Spain.

Williams, N., Marre, B., & Mouy, P. (2004). On-the-fly generation of k-paths tests for C functions : towards the automation of grey-box testing. In *the 19th IEEE International Conference on Automated Software Engineering (ASE'04)* (pp. 290–293). Linz, Austria: IEEE Computer Society.

Williams, N., Marre, B., Mouy, P., & Roger, M. (2005). PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In *the 5th European Dependable Computing Conference (EDCC'05)* (pp. 281–292). Budapest, Hungary.

Xu, Z., & Zhang, J. (2006). A test data generation tool for unit testing of C programs. In *the 6th International Conference on Quality Software (QSIC'06)* (pp. 107–116). Beijing, China.

Yorsh, G., Ball, T., & Sagiv, M. (2006). Testing, abstraction, theorem proving: better together! In *the 2006 ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'06)* (pp. 145–156). Portland, Maine, USA: ACM.

Zhu, H., Hall, P. A. V., & May, J. H. R. (1997). Software unit test coverage and adequacy. *ACM Computing Surveys*, *29*(4), 366–427.