Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-based Instance, and Actor-based Concurrency

Elvira Albert[1], Puri Arenas[1], Miguel Gómez-Zamalloa[1], Jose Miguel Rojas[2]

[1] DSIC, Complutense University of Madrid (UCM), Spain
[2] Department of Computer Science, University of Sheffield, UK

Abstract. The focus of this tutorial is white-box test case generation (TCG) based on symbolic execution. Symbolic execution consists in executing a program with the contents of its input arguments being symbolic variables rather than concrete values. A symbolic execution tree characterizes the set of execution paths explored during the symbolic execution of a program. Test cases can be then obtained from the successful branches of the tree. The tutorial is split into three parts: (1) The first part overviews the basic techniques used in TCG to ensure termination, handling heap-manipulating programs, achieving compositionality in the process and guiding TCG towards interesting test cases. (2) In the second part, we focus on a particular implementation of the TCG framework in constraint logic programming (CLP). In essense, the imperative object-oriented program under test is automatically transformed into an equivalent executable CLP-translated program. The main advantage of CLP-based TCG is that the standard mechanism of CLP performs symbolic execution for free. The PET system is an open-source software that implements this approach. (3) Finally, in the last part, we study the extension of TCG to actor-based concurrent programs.

1 Introduction

A lot of research has been devoted in the last years to the problem of generating test cases automatically. A recent survey [6] describes some of the most prominent approaches to TCG, namely *model-based TCG*, *combinatorial TCG*, *(adaptive) random TCG*, *search-based TCG* and *structural (white-box) TCG*. This tutorial focuses on *structural (white-box) TCG*, an approach in which the availability of the code of the program under test is assumed and test cases are obtained from the concrete program (e.g., using its control flow graph) in contrast to *black-box* testing, where they are deduced from a specification of the program. Also, our focus is on *static* testing, since we assume no knowledge about the input data, in contrast to *dynamic* approaches [17, 24] which execute the program under test using concrete input values.

Symbolic execution [11, 13, 15, 23, 31, 35, 36, 46] is arguably the most widely used enabling technique for structural white-box TCG. It has received a renewed interest in recent years, thanks in part to the increased availability of computational power and decision procedures [9]. Structural white-box TCG is among the most studied applications of symbolic execution, with several tools available [10]. Symbolic execution consists in executing a program with the contents of its input arguments being symbolic variables rather than concrete values. A symbolic execution *tree* characterizes the set of execution paths explored during the symbolic execution of a program. Test cases are obtained from the successful branches of the tree. The set of obtained test cases forms a test suite.

The first part of the tutorial is devoted to review the basic concepts of TCG by symbolic execution. We start by explaining the challenges to efficiently handle heap-manipulating programs [38] in symbolic execution. The presence of dynamic memory operations such as object creation and read/write field accesses requires special treatment during symbolic execution. Moreover, in order to ensure reliability, symbolic execution must consider all possible shapes these dynamic data structures can take. We proceed next to see how one can go to symbolic execution to the actual production of test cases. An important issue that is discussed afterwards is the compositionality of the TCG process. Finally, we overview a practical issue to efficiently generate more relevant test cases. In particular, guided TCG is a methodology that aims at steering symbolic execution towards specific program paths in order to generate relevant test cases and filter out less interesting ones.

The second part of the tutorial introduces CLP-based Test Case Generation. CLP-based TCG advocates the use of CLP technology to perform test case generation of imperative object-oriented programs. The process has two phases. In the first phase, the imperative object-oriented program under test is automatically transformed into an equivalent executable *CLP-translated* program. Instructions that manipulate heap-allocated data are represented by means of calls to specific *heap operations*. In the second phase, the CLP-translated program is symbolically executed using the standard CLP execution and constraint solving mechanisms. The above-mentioned heap operations are also implemented in standard CLP, in a suitable way in order to support symbolic execution. We will see the advantages of the CLP-based framework and, in particular, why it is very relevant to implement guided TCG and an efficient heap solver. In this context, we present the PET system, a system that implements the CLP-based TCG framework described in this part and which is available online.

The last part of the tutorial is focused on TCG of actor-based concurrent programs. It is known that writing correct concurrent programs is harder than writing sequential ones, because with concurrency come additional hazards not present in sequential programs such as race conditions, data races, deadlocks, and livelocks. However, due to the non-deterministic interleavings of processes, traditional testing for concurrent programs is not as effective as for sequential programs. Systematic and exhaustive exploration of all interleavings is typically too time-consuming and often computationally intractable (see, e.g., [45] and its references). Furthermore, the fact that different scheduling policies can be implemented affects the order in which tasks are selected for execution and, thus, the initial state when resuming a task can be different by adopting one policy or another. As a result, computation is often non-deterministic and multiple (possibly different) solutions can be produced depending on the interleaved tasks and the scheduler.

The adoption of actor systems has some advantages in the regard. Very briefly, actors [1, 25] constitute a model of concurrent programming that has been gaining popularity and that it is being used in many systems (such as ActorFoundry, Asynchronous Agents, Charm++, E, ABS, Erlang, and Scala). Actor programs consist of computing entities called actors or objects, each with its own local state and thread of control, that communicate by exchanging messages asynchronously. An object configuration consists of the local state of the objects and a set of pending messages (or *tasks*). In response to receiving a message, an object can update its local state, send messages, or create new objects. At each step in the computation of an object system, an object from the system is scheduled to process one of its pending messages. The advantage of using actor-systems in testing is that, as objects do not share their states, one can assume [41] that the evaluation of all statements of a task takes place serially (without interleaving with any other task) until it releases the processor (gets to a return instruction). This assumption alleviates already a lot the scalability issues mentioned above. We will discuss a basic algorithm and the main challenges in TCG of actor systems.

2 Test Case Generation by Symbolic Execution

This section provides a general overview of TCG by symbolic execution and the main challenges that currently the method poses.

2.1 Basic Concepts in Symbolic Execution

A symbolic execution *tree* characterizes the set of execution paths explored during the symbolic execution of a program. During the course of symbolic execution, the values of the program's variables are represented as symbolic expressions over the input symbolic values and a *path condition* is maintained. Such a path condition is updated whenever a branch instruction is executed. For instance, for each conditional statement in the program, symbolic execution explores both the "then" and the "else" branch, refining the path condition accordingly. The satisfiability of each of these branches is checked and symbolic execution stops exploring any path whose path condition becomes unsatisfiable, hence only feasible paths are followed. Test cases are obtained from the successful branches of the tree. The set of obtained test cases forms a test suite.

In this context, the quality of a test suite is usually assessed by using code coverage criteria. A coverage criterion aims at measuring how well the program under test is exercised by a test suite. Some popular coverage criteria are: *statement coverage* which requires that every statement of the code is executed; *branch coverage* which requires all conditional statements in the program to be evaluated both to true and false; and *path coverage* which requires that every possible trace through a given part of the code is executed. These criteria are however not *finitely applicable* [49]. That is, they can not always be satisfied by a *finite* test suite, due to infinite paths and infeasible statements in the program under test (i.e., dead code). An alternative to path coverage, which is *finitely applicable* is the *loop-k* coverage criterion, which requires traversing all paths in the program except those with more than k iterations on any loop.

Observe that by construction symbolic execution achieves the *path coverage* criterion above described. However, since the symbolic execution tree is in general infinite, a termination criterion must be imposed to ensure its finiteness. Such a termination criterion can be expressed in different forms. For instance, a computation time budget can be established, or an explicit bound on the depth of the symbolic execution tree can be imposed. We adopt a more code-oriented termination criterion. Concretely, we impose an upper bound k on the number of times each loop is iterated. By doing so, the finitely applicable (feasible) version of the *path coverage* criterion, i.e., the *loop-k* coverage, is achieved.

```
 1 int intExp(int a,int n) {
 2   if (n < 0)
 3     throw new ArithmeticException();
 4   else {
 5     int out = 1;
 6     while (n > 0) {
 7       out = out*a;
 8       n--;
 9     }
10     return out;
11   }
12 }
```

Fig. 1: Java source code

Example 1. Figure 1 shows the Java source code for method `intExp` which takes two integer input arguments `a` and `n` and computes `a`$^{\texttt{n}}$ by successive multiplications. If the value of the input argument `n` is less than 0, an arithmetic exception is thrown. For simplicity, we assume that the method cannot receive values 0 for both of its arguments (undefined 0^0). Figure 2 shows the symbolic execution tree of method `intExp` for *loop-1* termination criterion (*loop-k* with *k=1*). That is to say, we require all paths that do not exercise the loop body (zero times) and those that exercise the loop body one time. Nodes in the tree denote symbolic states, and the edges are labeled with the line number of the instruction that is executed. Observe that symbolic execution starts with the empty path condition (`PC`:*true*). At each branching point, `PC` is updated with different conditions over the input arguments. For instance, when the `if` statement is executed, both `then` (*true*) and `else` (*false*) alternatives are feasible, therefore symbolic execution forks and the `PC` is updated accordingly in each of the resulting paths.

In the tree, solid squares denote intermediate symbolic states, solid double squares denote successful (terminating) symbolic execution paths, and the only dashed square denotes an unfinished path, i.e., a path that is about to enter the loop body a second time and hence is pruned by the *loop-1* criterion. □

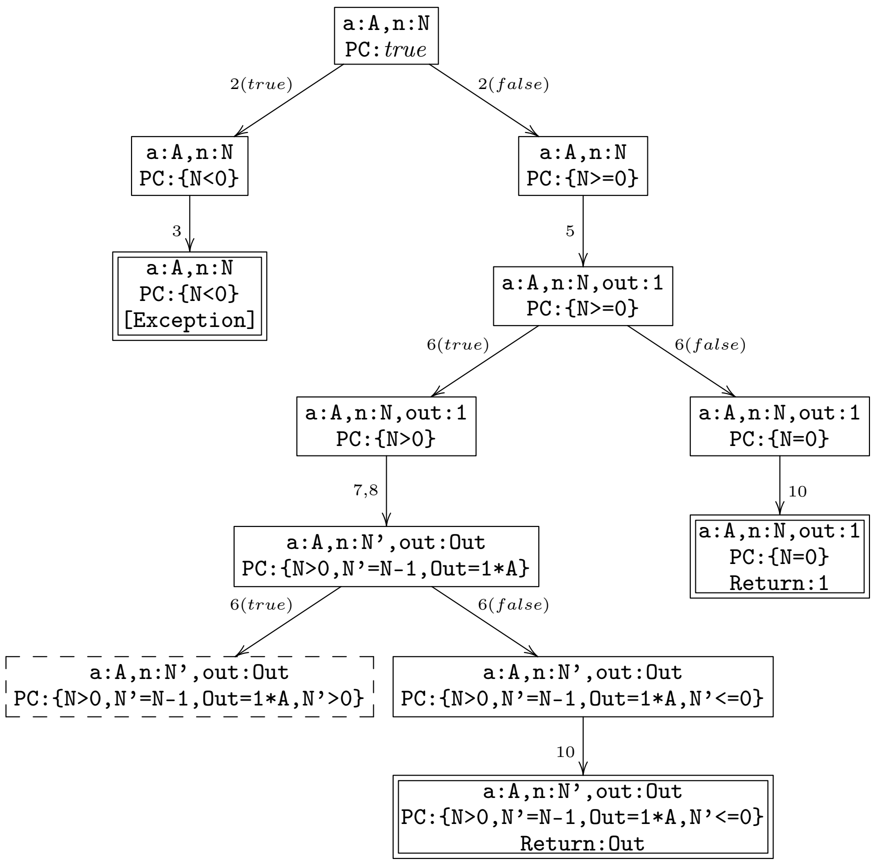


Fig. 2: Symbolic execution tree

2.2 Handling Heap-manipulating programs

One of the main challenges in symbolic execution is to efficiently handle heap-manipulating programs [38]. As will be illustrated later through an intuitive example, these kind of programs often create and use complex dynamically heap-allocated data structures. The presence of dynamic memory operations such as object creation and read/write field accesses requires special treatment during symbolic execution. Moreover, in order to ensure reliability, symbolic execution must consider all possible shapes these dynamic data structures can take. In trying to do so, however, scalability issues arise since high (often exponential) numbers of shapes may be built due to the *aliasing* of references.

In practice, symbolic execution assumes no knowledge about the heap shape (unless explicitly provided in advance via e.g., preconditions), in contrast to standard execution, where a program runs on concrete and fully-known initial heap (as part of the execution context). Let us motivate the importance of special treatment for heap operations and aliasing of references on a simple example.

Example 2. Consider the following method `mist`. It receives as input arguments two references `r1` and `r2` to objects of type `C` (contains a field `f` of integer type), checks the value of `r1.f` and writes `r2.f` in the `then` branch or writes `r1.f` in the `else` branch.

```
1 void mist(C r1, C r2) {
2   if (r1.f > 0)
3     r2.f = 1;
4   else
5     r1.f = 0;
6 }
```

Seemingly, the method contains only two feasible paths, each corresponding to one branch of the `if` statement:

1. If `r1.f>0`, then write `r2.f=1` (line 3).
2. If `r1.f<=0`, then write `r1.f=0` (line 5). Nothing is learned about `r2`.

However, these cases fall short to cover all possible executions of method `mist`. There are other unapparent execution paths that must also be explored. Namely:

3. If `r1` points to *null*, then a null pointer exception is thrown at line 2.
4. If `r1.f>0` and `r2` points to *null*, then a null pointer exception is thrown at line 3.
5. If `r1` and `r2` point to the same object `o` and `o.f>0`, then write `o.f=1` (line 3). We say that `r1` and `r2` are *aliased*.

Notice that only by exhaustive exploration of all possible heap configuration can symbolic execution generate these "hidden" paths and hence reveal the presence of potential runtime errors for this rather simple method. Furthermore, let this example also serve to see the relevance of the *loop-k* coverage criterion. Observe that the set of the first two cases above, while not being sufficient to exercise the complete behavior of method `mist`, would still be enough to achieve 100% branch and statement coverage, which may convey an illusory sense of confidence on the correctness of a *possibly* buggy program. □

Lazy Initialization. Lazy initialization [30] is the *de facto* standard technique to enable symbolic execution to systematically handle arbitrary input data structures, and to explore all possible heap shapes that can be generated during the process, including those produced due to aliasing of references. The main idea is that symbolic execution starts with no knowledge about the program's input arguments and, as the program symbolically executes and accesses object fields, the components of the program's inputs are initialized on an "as-needed" basis. The intuition is as follows. To symbolically execute method `m` of class `C`, a new object `o` of class `C` with all its fields uninitialized is created (the `this` object in Java). When an unknown field of primitive type is read, a fresh unconstrained variable is created for that field. When an unknown reference field is accessed, all possibilities are explored non-deterministically choosing among the following values: (a) `null`; (b) any existing symbolic object whose type is compatible with the field's type and might *alias* with it; and (c) a fresh symbolic object. Such non-deterministic choices are materialized into branches in the symbolic execution tree. As a result, the heap associated with any particular execution path is built using only the constraints induced by the visited code.

The practicality and effectiveness of lazy initialization has been proved with its use by existing symbolic execution engines such as PET and SPF. However, the very nature of the technique, i.e., producing branching due to *aliasing* choices at *every* heap operation point, hampers the overall efficiency of symbolic execution and its applicability to real-world programs.

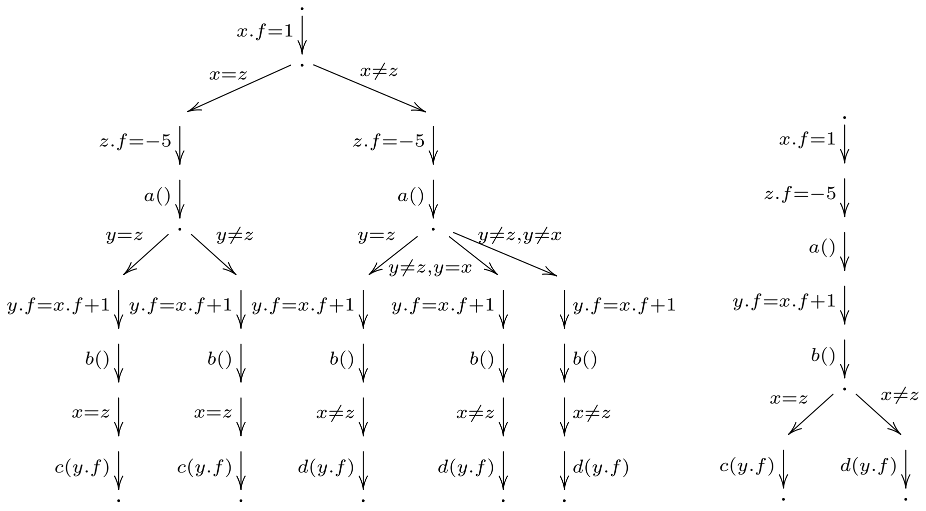
A Heap Solver. The observation that branching due to *aliasing* choices can be made "more lazily" than in lazy initialization by delaying such choices as much as possible lead to the development of a *heap solver* [4] which enables a more efficient symbolic execution of heap-manipulating programs. The key features of the heap solver are the treatment of reference aliasing by means of *disjunctive reasoning*, and the use of advanced *back-propagation* of heap related constraints. In addition, the heap solver supports the use of *heap assumptions* to avoid aliasing of data that, though legal, should not be provided as input.

Let us further illustrate the benefits of the heap solver over lazy initialization by symbolically executing method `m` from Figure 3 using both approaches. For simplicity, let us assume that the executions of methods `a` and `b` do not modify the heap. The symbolic execution tree computed using lazy initialization (as in, e.g., PET and SPF) is shown in Figure 4a. Note that before a field is accessed, the execution branches if it can alias with previously accessed fields. For example, the second field access `z.f` branches in order to consider the possible aliasing with the previously accessed `x.f`. Similarly, the write access to `y.f` must consider all possible aliasing choices with the two previous accessed fields `x.f` and `z.f`. This ensures that the effect of the field access is known within each branch. For example, in the leftmost branch the statement `y.f=x.f+1` assigns -4 to `x.f`, `y.f` and `z.f`, since in that branch all these objects are aliased. The advantage of this approach is that by the time we reach the `if` statement we know the result of the test, since each variable is fixed. However, such early branching creates

```
1 void m(Ref x, Ref y, Ref z) {
2   x.f=1;
3   z.f=-5;
4   a();
5   y.f=x.f+1;
6   b();
7   if (x==z)
8     c(y.f);
9   else
10    d(y.f);
11 }
```

Fig. 3: Heap Solver: Motivating example

a combinatorial explosion problem since, for example, method `a` is symbolically executed in two branches and method `b` in five.



(a) Lazy Initialization (b) Heap Solver

Fig. 4: Symbolic Execution Trees: Lazy Initialization and Heap Solver

On the other hand, the heap solver enables symbolic execution to perform as shown in Figure 4b, where branching only occurs due to explicit branching in the program, rather than to aliasing. For this purpose, the heap solver handles non-determinism due to aliasing of references by means of *disjunctions*. In particular, at instruction 5 the solver will carry the following conditional information for `x.f`' (the current value of field `f` of `x`): $x = z \rightarrow x.f' = z.f \wedge x \neq z \rightarrow x.f' = x.f$ indicating that if `x` and `z` are aliased, then `x.f`' will take its value from `z.f` and, otherwise, from `x.f`. Once the conditional statement at line 7 is executed and we learn that `x` and `z` are aliased (in the `then` branch), we need to look up *backwards* in the heap and propagate this unification so that instruction 5 can be fully executed. This allows the symbolic execution of `d(y.f)` with a known value for `y.f`. The heap solver works on a novel internal representation of the heap that encodes the disjunctive information and easily allows looking up backwards in the heap. In addition, it is possible to provide *heap assumptions* on non-aliasing, non-sharing and acyclicity of heap-allocated data in the initial state. The heap solver can take these assumptions into account to discard aliasing that is known not to occur for some input data. Importantly, the heap solver can be used by any symbolic execution tool for imperative languages through its interface heap operations.

Backwards Propagation, Arrays, and Heap Assumptions. As described in the previous section, the heap solver uses information about equality and disequality of references to determine equality among the heap cells. This is done by propagating such information forwards in the rules of attributes. A straightforward extension to the solver allows propagating information backwards as well. In doing so, the heap solver is capable of further refining disjunctive information and variables' domains, which in turn can lead to promptly pruning unfeasible symbolic execution branches.

Example 3. Consider the method `m` but with the condition of the `if` (in instruction 7) changed to "`if (x.f == 1)`". Thanks to backwards propagation, the solver can infer that in the `if` branch, variables `x`, `y` and `z` do not alias, and therefore the call `call_c` is performed with a 2 value. □

Another straightforward extension to the heap solver allows to handle arrays in a similar fashion to how object fields are handled, with the difference being that array indices play the role of object references that point to the heap-allocated data.

The last important feature of the heap solver is the support for heap assumptions. As we have seen so far, symbolic execution assumes feasible all possible kinds of aliasing among heap-allocated (reference) input data of the same type. However, it may be the case that while some of these aliasings might indeed occur, others might not (consider, for instance, aliased data structures that cannot be constructed using the public methods in the Java class). In order to avoid generating such inputs, the heap solver provides support for *heap assumptions*, that is, assertions describing reachability, aliasing, separation and sharing conditions in the heap. Concretely, the following heap assumptions are supported:

- *non-aliasing(a,b):* specifies that memory locations a and b are not the same.
- *non-sharing(a,b):* specifies disjointness, i.e., that references a and b do not share any common region in the heap.
- *acyclic(a):* specifies that a is an acyclic data structure.

2.3 From Symbolic Execution to TCG

The outcome of symbolic execution is a set of *path conditions*, one for each symbolic execution path. Each *path condition* represents the conditions over the input variables that characterize the set of feasible concrete executions of the program that take the same path. In a next step, off-the-shelf constraint solvers can be used to solve such path conditions and generate concrete instantiations for each of them. This last step provides actual test inputs for the program, amenable to further validation by testing frameworks such as JUnit, which execute such test inputs and check that the output is as expected.

Example 4. Let us look at the symbolic execution tree of Figure 2 again. Intuitively, the union of the three successful paths denoted with solid double squares make up the symbolic test suite for method `intExp` that optimally satisfies the *loop-1* criterion:

#	**Input**	**Output**	**Path condition**
1	`A, N`	`[exception]`	`{N<0}`
2	`A, N`	`1`	`{N=0}`
3	`A, N`	`Out`	`{N>0,N'=N-1,Out=1*A,N'<=0}`

The following are *concrete* test cases that can be derived from the above symbolic ones.

#	**Input**	**Output**
1	`-10, -10`	`[Exception]`
2	`-10, 0`	`1`
3	`-10, 1`	`10`

And from these concrete test cases, the JUnit tests shown in Figure 5 can be obtained.

It is important to note that imposing a larger k would allow to continue the exploration through the unfinished, pruned path (dashed square) thus generating test cases corresponding to further loop unrollings. □

2.4 Compositionality

Compositional reasoning is a general purpose methodology that has been successfully applied in the past to scale up static analysis and software verification techniques and that has also proved effective for scaling up symbolic execution and TCG [5, 7, 19, 40]. The overall goal of compositionality is to alleviate the inter-procedural path explosion problem. That is, in the context of symbolic execution and TCG, the path explosion caused by repeatedly conjoining the symbolic execution trees of methods when their invocations occur. The main idea is that symbolic execution and TCG of large programs can be done more effectively, and more efficiently, by first performing symbolic execution and TCG

```
public void test_1(){
  int input0 = -10;
  int input1 = -10;
  try{
    int output = Test.intExp(input0,input1);
  }
  catch(Exception ex){
    assertEquals("exception","java.lang.ArithmeticException",
                ex.getClass().getName());
    return;
  }
  fail("Fail");
}
public void test_2(){
  int input0 = -10;
  int input1 = 0;
  int output = Test.intExp(input0,input1);
  int expected = 1;
  assertEquals("OK",expected,output);
}
public void test_3(){
  int input0 = -10;
  int input1 = 1;
  int output = Test.intExp(input0,input1);
  int expected = -10;
  assertEquals("OK",expected,output);
}
```

Fig. 5: JUnit tests generated for introductory example

of their individual components separately. In the context of object-oriented programming, a method is the basic code component.

In symbolic execution for TCG, compositionality means that when a method m invokes another method p, for which TCG has already been performed, the execution can *compose* the *test cases* available for p (also known as *method summary* for p) with the current execution state and continue the process, instead of having to symbolically execute p again. By test cases (or method summary), we refer to the set of path conditions obtained by symbolically executing p. As a result of this composition step, a method summary for m is created. Then, larger portions of the system under test (components, modules, libraries, etc.) are incrementally executed, following a bottom-up traversal of its call graph, composing previously computed components results (*summaries*) until finally whole-program results can be computed. Let us recall that since the symbolic execution tree is in general infinite, a *termination criterion* is essential to ensure finiteness of the process. Then, a method summary is a *finite* set of *summary cases*, one for each terminating path through the symbolic execution tree of the method. Intuitively, a summary can be regarded as a complete specification of the method for a certain termination criterion, but it is still a partial specification of the method in general.

Intuitively, compositional TCG has several advantages over traditional non-compositional TCG. First, it avoids repeatedly performing TCG of the same method. Second, components can be tested with higher precision when they are chosen small enough. Third, since separate TCG is done on parts and not on the whole program, total memory consumption may be reduced. Fourth, separate TCG can be performed in parallel on independent computers and the global TCG time can be reduced as well. Furthermore, having a compositional TCG approach in turn provides a practical solution to handle *native code*, i.e., code which is implemented in a different programming language and may be unavailable. This is achieved by modeling the behavior of native code as a method summary which can be composed with the current state during symbolic execution in the same way as the test cases inferred automatically by the testing tool are. By treating native code, we overcome one of the inherent limitations of symbolic execution (see [38]).

Approaches to Compositional TCG. In order to perform compositional TCG, two main approaches can be considered:

Context-sensitive. Starting from an entry method m (and possibly a set of preconditions), TCG performs a top-down symbolic execution such that, when a method call p is found, its code is executed from the actual state ϕ. In a context-sensitive approach, once a method is executed, we store the summary computed for p in the context ϕ. If we later reach another call to p within a (possibly different) context ϕ', we first check if the stored context is sufficiently general. In such case, we can adapt the existing summary for p to the current context ϕ'. At the end of each execution, it can be decided which of the computed (context-sensitive) summaries are stored for future use.

Context-insensitive. Another possibility is to perform the TCG process in a context-insensitive way. This strategy comprises the following steps. First, the call graph for the entry method $m_{\mathcal{P}}$ of the program under test is computed, which gives us the set of methods that must be tested. Then, the strongly connected components (SCCs for short) for such graph are computed. SCCs are traversed in reverse topological order starting from those which do not depend on any other. The idea is that each SCC is symbolically executed from its entry m_{scc} w.r.t. the most general context (i.e., *true*). If there are several entries to the same SCC, the process is repeated for each of them. Hence, it is guaranteed that the obtained summaries can always be adapted to more specific contexts.

In general terms, the advantages of the context-insensitive approach are that composition can always be performed and that only one summary needs to be stored per method. However, since no context information is assumed, summaries can contain more test cases than necessary and can be thus more expensive to obtain. In contrast, the context-sensitive approach ensures that only the required information is computed, but it can happen that there are several invocations to the same method that cannot reuse previous summaries (because the associated contexts are not sufficiently general). In such case, it is more efficient to obtain the summary without assuming any context. A context-insensitive approach is used in what follows.

Method Summaries. A *method summary* for m is a finite set of *summary cases*, each of which mainly consists of the path condition for a particular symbolic execution path of m. Each element in a summary is said to be a *summary case* of the summary. Intuitively, a method summary can be seen as a *complete specification* of the method for the considered coverage criterion, so that each summary case corresponds to the *path constraints* associated to each finished path in the corresponding (finite) execution tree. Note that, though the specification is complete for the criterion considered, it will be, in general, a *partial specification* for the method, since the finite tree may contain incomplete branches which, if further expanded, may result in (infinitely) many execution paths.

When the method does not include any heap-related operation, the path condition alone sufficiently characterizes the symbolic execution path (as in [7,19]). However, in the presence of heap-manipulating methods, special mechanisms must be employed. We adopt an intuitive alternative which consists in explicitly encoding the input and output heaps and store them along with the path condition. Doing so, requires the implementation of two operations, a heap compatibility check and a heap composition operation.

Compatibility and Composition of Summaries. Let us assume that during the symbolic execution of a method m, there is a method invocation to another method p within a current state ϕ. The challenge is to define a composition operation so that, instead of symbolically executing p, its previously computed summary $\mathcal{S}_p$ can be reused. As a result, TCG for m should produce the same results regardless of whether we use a summary for p or we inline symbolical execution of p within TCG for m, in a non-compositional way. Roughly speaking, the state ϕ_c stored in a summary case is *compatible* with the current state ϕ if: 1) the path condition stored in the summary case can be conjoined to the current path condition, and 2) the structure of the input heap in the summary case match with the structure of the current heap. Note that compatibility of a summary case is checked on the fly, so that if ϕ is not compatible with ϕ_c, the composition will fail, the summary case will be discarded, and symbolic execution will proceed to attempt to compose the next summary case in $\mathcal{S}_p$.

Example 5. Table 1 shows the summary obtained by symbolically executing method `simplify` using the *loop-1* coverage criterion: The summary contains 5 cases, which correspond to the different execution paths induced by the calls to methods `gcd` and `abs`. For the sake of clarity, we adopt a graphical representation for the input and output heaps. Heap locations are shown as arrows labeled

```
class Arithmetics {
    static int abs(int a) {
        if (a >= 0) return a;
        else return -a;
    }
    static int gcd(int a,int b) {
        int res;
        while (b != 0) {
            res = a%b;  a = b;  b = res;
        }
        return abs(a);
    }
}
class Rational {
    int n; int d;
    void simplify() {
        int gcd = Arithmetics.gcd(n,d);
        n = n/gcd;  d = d/gcd;
    }
    Rational[] simp(Rational[] rs) {
        int length = rs.length;
        Rational[] oldRs = new Rational[length
            ];
        arraycopy(rs,oldRs,length);
        for (int i = 0; i < length; i++)
            rs[i].simplify();
        return oldRs;
    }
}
```

Fig. 6: Example for Compositional TCG.

Table 1: Summary of method `simplify`

A_{in}	A_{out}	$Heap_{in}$	$Heap_{out}$	EF	$Constraints$
r(A)		$A \rightarrow \begin{array}{c} F \\ \hline 0 \end{array}$	$A \rightarrow \begin{array}{c} M \\ \hline 0 \end{array}$	ok	F<0, N=-F, M=F/N
r(A)		$A \rightarrow \begin{array}{c} F \\ \hline 0 \end{array}$	$A \rightarrow \begin{array}{c} 1 \\ \hline 0 \end{array}$	ok	F>0
r(A)		$A \rightarrow \begin{array}{c} 0 \\ \hline 0 \end{array}$	$A \rightarrow \begin{array}{c} 0 \\ \hline 0 \end{array}$ $B \rightarrow$ AE	exc(B)	
r(A)		$A \rightarrow \begin{array}{c} F \\ \hline G \end{array}$	$A \rightarrow \begin{array}{c} M \\ \hline N \end{array}$	ok	G<0, F *mod* G=0, K=-G, M=F/K, N=G/K
r(A)		$A \rightarrow \begin{array}{c} F \\ \hline G \end{array}$	$A \rightarrow \begin{array}{c} M \\ \hline 1 \end{array}$	ok	G>0, F *mod* G=0, M=F/G

Table 2: Summary of method `arraycopy`

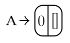
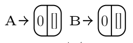
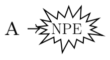
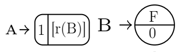
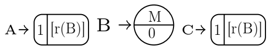
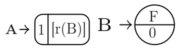
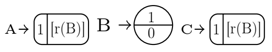
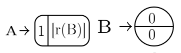
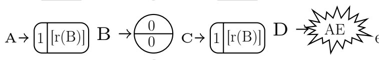
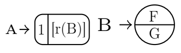
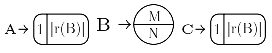
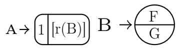
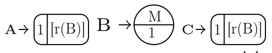
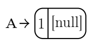
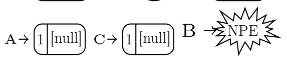
A_{in}	A_{out}	$Heap_{in}$	$Heap_{out}$	EF	$Constraints$
[X,Y,0]		H	H	ok	∅
[r(A),null,Z]		A→ L | [V|_]	A→ L | [V|_] B → NPE	exc(B)	Z>0, L>0
[null,Y,Z]		H	A → NPE	exc(A)	Z>0
[X,Y,Z]		H	A → AE	exc(A)	Z<0
[r(A),r(B),1]		A→ L1 | [V|_] B→ L2 | [V2|_]	A→ L1 | [V|_] B→ L2 | [V|_]	ok	L1>1, L2>0

with their reference variable names. Split-circles represent objects of type `R` and fields `n` and `d` are shown in the upper and lower part, respectively. Exceptions are shown as starbursts, like in the special case of the fraction "0/0", for which an arithmetic exception (`AE`) is thrown due to a division by zero. In the method summary examples of Tables 2 and 3, split-rectangles represent arrays, with the length of the array in the upper part and its list of values in the lower one. Assume that method `arraycopy` is native. This means that its code is not available and we cannot symbolically execute it. A method summary for `arraycopy` can be provided, as shown in Table 2, where we have (manually) specified five cases: the first one for arrays of length zero, the second and third ones for null array references, the fourth one for a negative length, and finally a normal execution on non-null arrays. Now, by using our compositional reasoning, we can continue symbolic execution for `simp` by composing the specified summary of `arraycopy` and the one computed for `simplify`. The result of compositional symbolic execution is presented in Table 3, that is, the entire summary of method `simp` for a *loop-1* coverage criterion. □

2.5 Guided TCG

A common limitation of symbolic execution in the context of TCG is that it tends to produce an unnecessarily large number of test cases for all but tiny programs. This limitation not only hinders scalability but also complicates human reasoning on the generated test cases. Guided TCG is a methodology that aims at steering symbolic execution towards specific program paths in order to efficiently generate more relevant test cases and filter out less interesting ones with respect to a given structural *selection criterion*. The goal is thus to improve on scalability and efficiency by achieving a high degree of control over the coverage criterion and hence avoiding the exploration of unfeasible paths. This has potential applicability for industrial software testing practices such as unit testing, where units of code (e.g. methods) must be thoroughly tested in isolation, or *selective testing*, in which only specific paths of a program must be tested.

Table 3: Summary of method `simp`

A_{in}	A_{out}	$Heap_{in}$	$Heap_{out}$	EF	$Constraints$
r(A)	r(B)			ok	∅
null	X	H		exc(A)	∅
r(A)	r(C)			ok	F<0, K=-F, M=F/K
r(A)	r(C)			ok	F>0
r(A)	X			exc(D)	∅
r(A)	r(C)			ok	G<0, F *mod* G=0, K=-G, M=F/K, N=G/K
r(A)	r(C)			ok	G>0, F *mod* G=0, M=F/G
r(A)	X			exc(B)	∅

Example 6. Let us consider the unit-testing for method `simplify` (see Figure 6). A proper set of unit-tests should include one test to exercise the exceptional behavior arising from the division by zero, and another test to exercise the normal behavior. Ideally, no more tests should be provided since there is anything else to be tested in method `simplify`. This methodology works well under the assumption that called methods are tested on their own, in this case method `gcd`. Standard TCG by symbolic execution would consider all possible paths including those arising from the different executions of method `gcd`, in this case 5 paths. The challenge in Guided TCG is to generate only the two test-cases above, avoiding as much as possible traversing the rest of the paths (which for this criterion can be considered redundant). As another example, let us consider selective testing for method `simplify`. E.g., one could be interested in generating a test-case (if any) that makes method `simplify` produce an exception due to a division by zero. The challenge in Guided TCG is again to generate such a test avoiding traversing as much as possible the rest of the paths. □

The intuition of Guided TCG is as follows: (1) A heuristics-based *trace-generator* generates possibly partial traces, i.e., partial descriptions of paths, according to a given selection criterion. This can be done by relying on the control-flow graph of the program. (2) Bounded symbolic execution is guided by the obtained traces. The process is repeated until the selection criterion is satisfied or until no more traces are generated. Section 3.6 presents a concrete CLP-based methodology for guided TCG and formalizes a concrete guided TCG scheme to support the criteria for unit testing considered in the above example.

3 CLP-based TCG

We present a particular instance of TCG based on symbolic execution, and an implementation, in which CLP is used as enabling technology.

3.1 Constraint Logic Programming

We assume certain familiarity with Logic Programming (LP) [33] and Constraint Logic Programming (CLP) [27,34]. Hence we only briefly overview both paradigms.

Logic Programming. Logic Programming is a programming paradigm based on the use of formal logic as a programming language. A logic program is a finite set of predicates defining relationships between logical terms. An atom (or call) A is a syntactic construct of the form $p(t_1, \ldots, t_n)$, with $n \geq 0$, where p/n is a predicate signature and $t_1, \ldots, t_n$ are terms. A clause is of the form $H : -B_1, \ldots, B_m.$, with $m \geq 0$, where its head H is an atom and its body $B_1, \ldots, B_m$ is a conjunction of m atoms (commas denote conjunctions). When $m = 0$ the clause is called a fact and is written "H.". The standard syntactic convention is that names of predicates and atoms begin with a lowercase letter. A goal is a conjunction of atoms. We denote by $\{X_1 \to t_1, \ldots, X_n \to t_n\}$ the substitution σ with $\sigma(X_i) = t_i$ for $i = 1, \ldots, n$ (with $X_i \neq X_j$ if $i \neq j$), and $\sigma(X) = X$ for all other variables X. Given an atom A, $\theta(A)$ denotes the application of substitution θ to A. Given two substitutions θ_1 and θ_2 , we denote by $\theta_1\theta_2$ their composition. An atom A' is an instance of A if there is a substitution σ with $A' = \sigma(A)$.

SLD (Selective Linear Definite clause)-resolution is the standard operational semantics of logic programs. It is based on the notion of derivations. A derivation step is defined as follows. Let G be $A_1, \ldots, A_R, \ldots, A_k$ and $C = H : -B_1, \ldots, B_m.$ be a renamed apart clause in P (i.e., it has no common variables with G). Let A_R be the selected atom for its evaluation. As in Prolog, we assume the simple leftmost selection rule. Then, G' is derived from G if θ is a *most general unifier* between A_R and H, and G' is the goal $\theta(A_1, \ldots, A_{R-1}, B_1, \ldots, B_m, A_{R+1}, \ldots, A_k)$.

As customary, given a program P and a goal G, an SLD derivation for $P \cup \{G\}$ consists of a possibly infinite sequence $G = G_0, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of properly renamed apart clauses of P (i.e. C_i has no common variables with any G_j nor C_j with $j < i$), and a sequence of computed answer substitutions $\theta_1, \theta_2, \ldots$ (or most-general unifiers, *mgus* for short) such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. Finally, we say that the SLD derivation is composed of the subsequent goals $G_0, G_1, G_2, \ldots$

A derivation step can be non-deterministic when A_R unifies with several clauses in P, giving rise to several possible SLD derivations for a given goal. Such SLD derivations can be organized in SLD trees. A finite derivation $G = G_0, G_1, G_2, \ldots, G_n$ is called successful if G_n is the empty goal, denoted ϵ. In that case $\theta = \theta_1\theta_2 \dots \theta_n$ is called the computed answer for goal G. Such a derivation is called failing if it is not possible to perform a derivation step with G_n.

Executing a logic program P for a goal G consists in building an SLD tree for $P \cup \{G\}$ and then extracting the computed answer substitutions from every non-failing branch of the tree.

Constraint Logic Programming. Constraint Logic Programming is a programming paradigm that extends Logic programming with *Constraint solving*. It augments the LP expressive power and application domain while maintaining its semantic properties (e.g., existence of a fixpoint semantics).

In CLP, the bodies of clauses may contain constraints in addition to ordinary literals. CLP integrates the use of a constraint solver to the operational semantics of logic programs. As a consequence of this extension, whereas in LP a computation state consists of a goal and a substitution, in CLP a computation state also contains a *constraint store*. The special *constraint* literals are stored in the *constraint store* instead of being solved according to SLD-resolution. The satisfiability of the constraint store is checked by a constraint solver. Then, we say that a CLP computation is successful if there is a derivation leading from the initial state $S_0 = \langle G_0 \mid \mathit{true} \rangle$ (initially the constraint store is empty, i.e., *true*) to the final state $S_n = \langle \epsilon \mid S \rangle$ such that ϵ is the empty goal and S is satisfiable.

The CLP paradigm can be instantiated with many constraint domains. A constraint domain defines the class of constraints that can be used in a CLP program. Several constraint domains have been developed (e.g., for terms, strings, booleans, reals). A particularly useful constraint domain is CLP(FD) (Constraint Logic Programming over Finite Domains) [47]. CLP(FD) constraints are usually intended to be arithmetic constraints over finite integer domain variables. It has been applied to constraint satisfaction problems such as planning and scheduling [14,34]. Some features of CLP(FD) that make it suitable for TCG of programs working with integers are:

– It provides a mechanism to define the initial finite domain of variables as an interval over the integers and operations to further refine this initial domain.
– It provides a built-in *labeling* mechanism, which can be applied on a list of variables to find values for them such that the current constraint store is satisfied.

As we will see in the next section, our CLP-based TCG framework will rely on CLP(FD) to translate conditional statements over integer variables into CLP constraints. Moreover, the labeling mechanism is essential to concretize the obtained test cases in order to obtain concrete input data amenable to be used and validated by testing tools.

3.2 CLP-based Test Case Generation

CLP-based Test Case Generation advocates the use of CLP technology to perform test case generation of imperative object-oriented programs. The process has two phases. In the first phase, the imperative object-oriented program under test is automatically transformed into an equivalent executable *CLP-translated* program. Instructions that manipulate heap-allocated data are represented by means of calls to specific *heap operations*. In the second phase, the CLP-translated program is symbolically executed using the standard CLP execution and constraint solving mechanism. The above-mentioned heap operations are also implemented in standard CLP, in a suitable way in order to support symbolic execution. The next two sections overview these two phases, which are also shown graphically in Figure 7.

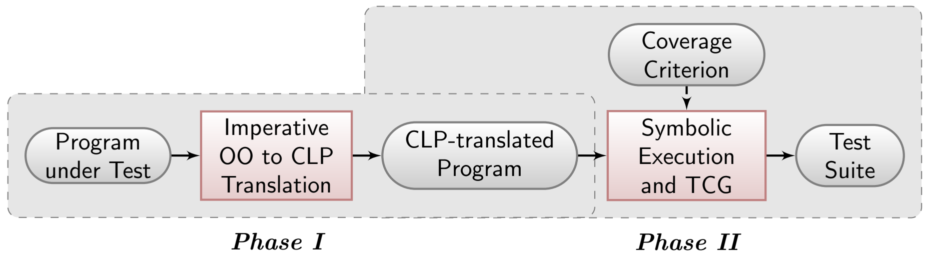


Fig. 7: CLP-based Test Case Generation Framework

The Imperative Object-Oriented Language. Although our approach is not tied to any particular imperative object-oriented language, we consider as the source language a subset of Java. For simplicity, we leave out of such subset features like concurrency, bitwise operations, static fields, access control (i.e., the use of `public`, `protected` and `private` modifiers) and primitive types besides integers and booleans. Nevertheless, these features can be relatively easy to handle in practice by our framework, except for concurrency, which is well-known to pose further challenges to symbolic execution and its scalability.

CLP-translated Programs. The translation of imperative object-oriented programs into equivalent CLP-translated programs has been subject of previous work (see, e.g., [2, 21]). Therefore, we will recap the features of the translated programs without going into deep details of how the translation is done. The translation is formally defined as follows:

Definition 1 (CLP-translated program). *The CLP-translated program for a given method m from the original imperative object-oriented program consists of a finite, non-empty set of predicates $m, m_1, \ldots, m_n$. A predicate m_i is defined by a finite, non-empty set of mutually exclusive rules, each of the form $m_i^k(In, Out, H_{in}, H_{out}, E) :- [g,]b_1, \ldots, b_j.$, where:*

1. *In and Out are, resp., the (possibly empty) list of input and output arguments.*
2. *H_{in} and H_{out} are, resp., the input and (possibly modified) output heaps.*

3. *E is an exception flag that indicates whether the execution of m_i^k ends normally or with an uncaught exception.*
4. *If m_i is defined by more than one rule, then g is the constraint that guards the execution of m_i^k, i.e., it must hold for the execution of m_i^k to proceed.*
5. *$b_1, \ldots, b_j$ is a sequence of instructions including arithmetic operations, calls to other predicates and built-ins to operate on the heap, etc., as defined in Figure 8. As usual, an SSA transformation is performed [12].*

$$
\begin{array}{rcl}
Clause & ::= & Pred(Args_{in}, Args_{out}, H_{in}, H_{out}, ExFlag) \text{ :- } [G,]B_1, B_2, \ldots, B_n. \\
G & ::= & Num^* \; ROp \; Num^* \mid Ref_1^* \; \backslash{==} \; Ref_2^* \mid \mathsf{type}(H, Ref^*, T) \\
B & ::= & Var \; \#{=} \; Num^* \; AOp \; Num^* \\
 & & \mid Pred(Args_{in}, Args_{out}, H_{in}, H_{out}, ExFlag) \\
 & & \mid \mathsf{new\_object}(H_{in}, C^*, Ref^*, H_{out}) \\
 & & \mid \mathsf{new\_array}(H_{in}, T, Num^*, Ref^*, H_{out}) \mid \mathsf{length}(H_{in}, Ref^*, Var) \\
 & & \mid \mathsf{get\_field}(H_{in}, Ref^*, FSig, Var) \mid \mathsf{set\_field}(H_{in}, Ref^*, FSig, Data^*, H_{out}) \\
 & & \mid \mathsf{get\_array}(H_{in}, Ref^*, Num^*, Var) \\
 & & \mid \mathsf{set\_array}(H_{in}, Ref^*, Num^*, Data^*, H_{out})
\end{array}
$$

$$
\begin{array}{rcl rcl}
Pred & ::= & Block \mid MSig & ROp & ::= & \#{>} \mid \#{<} \mid \#{>=} \mid \#{=<} \mid \#{=} \mid \#\backslash{=} \\
Args & ::= & [\,] \mid [Data^* | Args] & AOp & ::= & + \mid - \mid * \mid / \mid mod \\
Data & ::= & Num \mid Ref \mid ExFlag & T & ::= & bool \mid int \mid C \mid array(T) \\
Ref & ::= & null \mid r(Var) & FSig & ::= & C{:}FN \\
ExFlag & ::= & ok \mid exc(Var) & H & ::= & Var
\end{array}
$$

Fig. 8: Syntax of CLP-translated programs

Specifically, CLP-translated programs adhere to the grammar in Figure 8. As customary, terminals start with lowercase (or special symbols) and non-terminals start with uppercase; subscripts are provided just for clarity. Non-terminals *Block*, *Num*, *Var*, *FN*, *MSig*, *FSig* and *C* denote, resp., the set of predicate names, numbers, variables, field names, method signatures, field signatures and class names. A clause indistinguishably defines either a method which appear in the original source program (*MSig*), or an additional predicate which correspond to an intermediate block in the control flow graph of original program (*Block*). A field signature *FSig* contains the class where the field is defined and the field name *FN*. An asterisk on a non-terminal denotes that it can be either as defined by the grammar or a (possibly constrained) variable (e.g., Num^*, denotes that the term can be a number or a variable). Heap references are written as terms of the form $r(Ref)$ or *null*. The operations that handle data in the heap are translated into built-in heap-related predicates.

Let us observe the following:

– There exists a one-to-one correspondence between blocks in the control flow graph of the original program and rules in the CLP-translated one.
– Mutual exclusion between the rules of a predicate is ensured either by means of mutually exclusive *guards*, or by information made explicit on the heads of rules, as usual in CLP. This makes the CLP-translated program deterministic, as the original imperative one is (point 4 in Definition 1).
– The global memory (or heap) is explicitly represented by means of logic variables. When a rule is invoked, the input heap $\mathtt{H_{in}}$ is received and, after executing the body of the rule, the heap might be modified, resulting in $\mathtt{H_{out}}$. The operations that modify the heap will be shown later.
– Virtual method invocations are resolved at compile-time in the original imperative object-oriented language by looking up all possible runtime instances of the method. In the CLP-translated program, such invocations are translated into a choice of type instructions which check the actual object type, followed by the corresponding method invocation for each runtime instance.
– Exceptional behavior is handled explicitly in the CLP-translated program.

These observations will become more noticeable later on Example 7.

Note that the above definition proposes a translation to CLP as opposed to a translation to pure logic (e.g. to predicate logic or even to propositional logic, i.e., a logic that is not meant for "programming"). This is because we then want to execute the resulting translated programs to perform TCG and this requires, among other things, handling a constraint store and then generating actual data from such constraints. CLP is a natural paradigm to perform this task.

Heap Operations. Figure 9 summarizes the CLP implementation of the operations to create heap-allocated data structures (new_object and new_array) and to read and modify them (getfield , set_array, etc.) [22]. These operations rely on some auxiliary predicates (like deterministic versions of member member_det, of replace replace_det, and nth0 and replace_nth0 for arrays) which are quite standard and hence their implementation is not shown. For instance, a new object is created through a call to predicate new_object(H_{in},Class,Ref,H_{out}), where H_{in} is the current heap, Class is the new object's type, Ref is a unique reference in the heap for accessing the new object and H_{out} is the new heap after allocating the object. Read-only operations do not produce any output heap. For example, get_field(H_{in},Ref,FSig,Var) retrieves from H_{in} the value of the field identified by *FSig* from the object referenced by *Ref*, and returns its value in *Var* leaving the heap unchanged. Instruction set_field(H_{in},Ref,FSig,Data,H_{out}) sets the field identified by *FSig* from the object referenced by *Ref* to the value *Data*, and returns the modified heap H_{out}. The remaining operations are implemented likewise.

The Heap term. Our CLP-translated programs manipulate the heap as a black-box through its associated operations. The heaps generated and manipulated by

```
   new_object(H,C,Ref,H') :- build_object(C,Ob), new_ref(Ref),
                             H' = [(Ref,Ob)|H].
  new_array(H,T,L,Ref,H') :- build_array(T,L,Arr), new_ref(Ref),
                             H' = [(Ref,Arr)|H].

            type(H,Ref,T) :- get_cell(H,Ref,Cell), Cell = object(T,_).
          length(H,Ref,L) :- get_cell(H,Ref,Cell), Cell = array(_,L,_).

  get_field(H,Ref,FSig,V) :- get_cell(H,Ref,Ob), FSig = C:FN,
                             Ob = object(T,Fields), subclass(T,C),
                             member_det(field(FN,V),Fields).
     get_array(H,Ref,I,V) :- get_cell(H,Ref,Arr), Arr = array(_,_,Xs),
                             nth0(I,Xs,V).

set_field(H,Ref,FSig,V,H') :- get_cell(H,Ref,Ob), FSig = C:FN,
                             Ob = object(T,Fields), subclass(T,C),
                             replace_det(Fields,field(FN,_),field(FN,V),
                                         Fields'),
                             set_cell(H,Ref,object(T,Fields'),H').
  set_array(H,Ref,I,V,H') :- get_cell(H,Ref,Arr), Arr = array(T,L,Xs),
                             replace_nth0(Xs,I,V,Xs'),
                             set_cell(H,Ref,array(T,L,Xs'),H').
```

```
  get_cell([(Ref',Cell')|_],Ref,Cell) :- Ref == Ref', !, Cell = Cell'.
            get_cell([_|RH],Ref,Cell) :- get_cell(RH,Ref,Cell).

   set_cell([(Ref',_)|H],Ref,Cell,H') :- Ref == Ref', !,
                                         H' = [(Ref,Cell)|H].
set_cell([(Ref',Cell')|H'],Ref,Cell,H) :- H = [(Ref',Cell')|H''],
                                         set_cell(H',Ref,Cell,H'').
```

Fig. 9: Heap operations for ground execution [22]

using these operations adhere to this grammar:

$$\begin{aligned} Heap &::= [\,] \mid [Loc|Heap] \\ Cell &::= object(C^{*},Fields^{*}) \mid array(T^{*},Num^{*},Args^{*}) \\ Loc &::= (Num^{*},Cell) \\ Fields &::= [\,] \mid [field(FN,Data^{*})|Fields^{*}] \end{aligned}$$

The heap is represented as a list of locations which are pairs formed by a unique reference and a cell. Each cell can be an object or an array. An object contains its type and its list of fields, each of which is made of its signature and data content. An array contains its type, its length and its list of elements.

Example 7. Figure 10a shows the Java source code of class `List`, which implements a singly-linked list. The class contains one field `first` of type `Node`. As customary, `Node` is a recursive class with two fields: `data` of type `int` and `next` of type `Node`. Method `remAll` takes as argument an object `l` of type `List`, traverses it (outer while loop) and for each of its elements, traverses the `this` object and removes all their occurrences (inner loop). Figure 10b shows the equivalent

```
 1 class Node {
 2   int data;
 3   Node next;
 4 }
 5 class List {
 6   Node first;
 7   void remAll(List l) {
 8     // block1
 9     Node lf = l.first;
10     // loop1
11     while (lf != null) {
12       // block2
13       Node prev = null;
14       Node p = null;
15       Node next = first;
16       // loop2
17       while (next != null) {
18         // block3
19         prev = p;
20         p = next;
21         next = next.next;
22         // if1
23         if (p.data == lf.data)
24           // if2
25           if (prev == null) {
26             first = next;
27             p = null;
28           } else {
29             prev.next = next;
30             p = prev;
31           }
32       }
33       // block4
34       lf = lf.next;
35     }
36   }
37 }
```

(a) Java source code

```
remAll([r(Th),L],[],Hi,Ho,E) :-
  block1([Th,L],Hi,Ho,E).
block1([Th,r(L)],Hi,Ho,E) :-
  get_field(Hi,L,first,LfR),
  loop1([Th,L,LfR],Hi,Ho,E).
block1([Th,null],Hi,Ho,exc(E)) :-
  create_object(Hi,'NPE',E,Ho).
loop1([Th,L,null],H,H,ok).
loop1([Th,L,r(Lf)],Hi,Ho,E) :-
  block2([Th,L,Lf],Hi,Ho,E).
block2([Th,L,Lf],Hi,Ho,E) :-
  get_field(Hi,Th,first,FR),
  loop2([Th,L,Lf,null,null,FR],Hi,Ho,E).
loop2([Th,L,Lf,Prev,P,null],Hi,Ho,E) :-
  block4([Th,L,Lf],Hi,Ho,E).
loop2([Th,L,Lf,Prev,P,r(F)],Hi,Ho,E) :-
  block3([Th,L,Lf,P,F],Hi,Ho,E).
block3([Th,L,Lf,P,F],Hi,Ho,E) :-
  get_field(Hi,F,next,FRN),
  get_field(Hi,F,data,A),
  get_field(Hi,Lf,data,B),
  if1([A,B,Th,L,Lf,P,F,FRN],Hi,Ho,E).
if1([A,B,Th,L,Lf,Prev,P,FRN],Hi,Ho,E) :-
  #\=(A,B),
  loop2([Th,L,Lf,Prev,P,FRN],Hi,Ho,E).
if1([A,A,Th,L,Lf,Prev,P,FRN],Hi,Ho,E) :-
  if2([Th,L,Lf,Prev,P,FRN],Hi,Ho,E).
if2([Th,L,Lf,r(F),P,N],Hi,Ho,E) :-
  set_field(Hi,F,next,N,H2),
  loop2([Th,L,Lf,F,F,N],H2,Ho,E).
if2([Th,L,Lf,null,P,N],Hi,Ho,E) :-
  set_field(Hi,Th,first,N,H2),
  loop2([Th,L,Lf,null,null,N],H2,Ho,E).
block4([Th,L,Lf],Hi,Ho,E) :-
  get_field(Hi,Lf,next,LfRN),
  loop1([Th,L,LfRN],Hi,Ho,E).
```

(b) CLP-translation

Fig. 10: CLP-based TCG example

(simplified and pretty-printed) CLP-translated code for method `remAll`. Let us observe some of the main features of the CLP-translated program. The `if` statement in line 23 is translated into two mutually exclusive rules (predicate `if1`) guarded by an arithmetic condition. Similarly, the `if` statement in line 25 is translated into predicate `if2`, implemented by two rules whose mutual exclusion is guaranteed by terms `null` and `r(_)` appearing in each rule head. Observe that iteration in the original program (`while` constructions) is translated into recursive predicates. For instance, the head of the inner `while` loop is translated into predicate `loop2`, its condition is guarded by the rules of predicate `cond2` (`null` or `r(_)`), and recursive calls are made from predicates `if1` (first rule) and `if2` (both rules). Finally, exception handling is made explicit in the CLP-translated program; the second rule of predicate `block1` encodes the runtime null pointer exception (`'NPE'`) that raises if the input argument `l` is `null`. □

3.3 Semantics of CLP-translated Programs

The standard CLP execution mechanism suffices to execute the CLP-translated programs. Let us focus on the concrete execution of CLP-translated programs by assuming that all input parameters of the predicate to be executed (i.e., In and H_{in}) are fully instantiated in the initial input state.

Let M be a method in the original imperative program, m be its corresponding predicate in the CLP-translated program P, and P' be the union of P and the predicates in Figure 9. As explained in the previous section, the operational semantics of the CLP program P' can be defined in terms of *derivations*. A derivation is a sequence of reductions between states $S_0 \rightarrow_p S_1 \rightarrow_P \ldots \rightarrow_P S_n$, also denoted $S_0 \rightarrow_P S_n$, where a *state* $\langle G \mid \theta \rangle$ consists of a goal G and a constraint store θ. The concrete execution of m with input θ is the derivation $S_0 \rightarrow S_n$, where $S_0 = \langle m(In, Out, H_{in}, H_{out}, ExFlag) \mid \theta \rangle$ and θ initializes In and H_{in} to be fully ground. If the derivation successfully terminates, then $S_n = \langle \epsilon \mid \theta' \rangle$ and θ' is the *output constraint store*.

This definition of concrete execution relies on the correctness of the translation algorithm, which must guarantee that the CLP-translated program captures the same semantics of the original imperative one [2, 21].

Example 8. The following is a *correct* input state for predicate `remAll/5`:

$$\langle \texttt{remAll(}\underline{\texttt{[r(1),null]}}\texttt{,Out,}$$
$$\underline{\texttt{[(1,object('List',[field('Node':first,null)]))]}}\texttt{,Hout,E)} \mid \mathit{true} \rangle$$

Observe that the list of input arguments and the input heap (both underlined) are fully instantiated. Argument `r(1)` corresponds to the implicit reference to the *this* object, which appears in the input heap term with its field `first` being instantiated to `null`. Concrete execution on this input state yields a final state in which:

```
Out  = [ ]∧
Hout = [(1,object('List',[field('Node':first,null)])),
        (2,object('NPE',[ ]))]∧
E    = exc(2)
```

Notice that in this final state, a new object of type NPE (Null Pointer Exception) is created in the heap. The fact that the execution ends with an uncaught exception is indicated in flag E. □

3.4 Symbolic Execution

When the source imperative language does not support dynamic memory, symbolic execution of the CLP-translated programs is attained by simply using the standard CLP execution mechanism to run the main goal (i.e., the predicate name after the method under test) *with all arguments being free variables*. The inherent constraint solving and backtracking mechanisms of CLP allow to keep track of path conditions (or constraint stores), failing and backtracking when unsatisfiable constraints are hit, hence discarding such execution paths; and succeeding when satisfiable constraints lead to a terminating state in the program, which in the context of TCG implies that a new test case is generated.

However, in the case of heap-manipulating programs, the heap-related operations presented in Figure 9 fall short to generate arbitrary heap-allocated data structures and all possible heap shapes when accessing symbolic references. This is a well-known problem in TCG by symbolic execution. A naive solution to this problem could be to fully initialize all the reference parameters prior to symbolic execution. However, this would require imposing bounds on the size of input data structures, which is highly undesirable. Doing so would circumscribe the symbolic search space, hence jeopardizing the overall effectiveness of the technique.

Lazy Initialization. A straightforward generalization of predicate get_cell in Figure 9 provides a simple and flexible solution to the problem of handling arbitrary input data structures during symbolic execution, and constitutes a quite natural implementation of the *lazy initialization* technique in our CLP-based framework. Figure 11 shows the new implementation of the get_cell operations; observe that we have added just two new rules to the implementation shown in Figure 9.

```
                get_cell(H,Ref,Cell) :- var(H), !, H = [(Ref,Cell)|_].
get_cell([(Ref',Cell')|_],Ref,Cell) :- Ref == Ref', !, Cell = Cell'.
get_cell([(Ref',Cell')|_],Ref,Cell) :- var(Ref), var(Ref'), Ref = Ref',
                                       Cell = Cell'.
           get_cell([_|RH],Ref,Cell) :- get_cell(RH,Ref,Cell).
```

Fig. 11: Redefining get_cell operations for symbolic execution [22]

The intuitive idea is that the heap during symbolic execution contains two parts: the *known part*, with the cells that have been explicitly created during symbolic execution appearing at the beginning of the list, and the *unknown part*, which is a logic variable (tail of the list) in which new data can be added. Importantly, the definition of `get_cell/3` distinguishes two situations when searching for a reference: (i) It finds it in the known part (second clause), meaning that the reference has already been accessed earlier (note the use of syntactic equality rather than unification, since references at execution time can be variables); or (ii) It reaches the unknown part of the heap (a logic variable), and it allocates the reference (in this case a variable) there (first clause). The third clause of `get_cell/3` allows to consider all possible aliasing configurations among references. In essence, `get_cell/3` is therefore a CLP implementation of *lazy initialization*.

Let us illustrate the use of lazy initialization in symbolic execution with an example.

Example 9. Figure 12 shows the CLP-translated program for method `mist` from Example 2. Let `mist(In,Out,Hin,Hout,E)` be the initial goal for symbolic execution. Observe that the input heap `Hin` is a free variable (i.e., fully unknown). Let us choose rule $\texttt{mist}^1$. By doing so, the list of input arguments `In` gets instantiated to `[r(A),R2]`, which indicates that the first argument is a reference to an existing object in the heap, as opposed to the `null` reference in rule $\texttt{mist}^2$. The execution of the `get_field` instruction imposes new constraints on the shape of the input heap. Namely, `Hin` is partially instantiated to `[(A,object('C',[field(f,F)|M]))|N]`. Observe that there is still an unknown part in the heap (variable `N`). Also, observe that the list of fields for object `A` is also represented by an open list, meaning that there might be other fields in this object, but nothing has been learned about them yet.

Now, let us assume that the execution proceeds with rules $\texttt{if}^1$ and $\texttt{then}^1$. At this point, the second argument is also set to be a valid reference `r(B)`. The execution of the `set_field` will internally reach predicate `get_cell` (Figure 11), leading to consider two possibilities:

– References `R1=r(A)` and `R2=r(B)` point to two different objects in the heap. In this case, the resulting output heap is

```
Hout = [ (A,object('C',[field(f,D1)|M])),
         (B,object('C',[field(f,1)|P]))|N],
```

and the constraint store is $\theta = \{D1 > 0\}$.

– References `R1=r(A)` and `R2=r(A)` point to the same object in the heap, i.e., they are aliased. Here, the resulting output heap is `Hout=[(A,object('C',[field(f,D1)|M]))|N]`, with $\theta = \{D1 > 0\}$. □

To conclude this section, let us now provide a definition for symbolic execution in terms of the CLP derivation tree of the CLP-translated program extended with built-in operations to handle dynamic memory:

Definition 2 (Symbolic Execution). *Let M be a method, m be its corresponding predicate from its associated CLP-translated program P, and P' be the union of P and the set of predicates in Figure 9. The symbolic execution of m is*

```
mist¹([r(A),R2],[],Hin,Hout,E) :-
    get_field(Hin,A,f,D1),
    if([D1,A,R2],Hin,Hout,E).
mist²([null,R2],[],Hin,Hout,exc(Exc)) :-
    create_object(Hin,'NPE',Exc,Hout).
if¹([D1,A,R2],Hin,Hout,E) :-
    #>(D1,0),
    then([R2],Hin,Hout,E).
if²([D1,A,R2],Hin,Hout,ok) :-
    #<=(D1,0),
    set_field(Hin,A,f,0,Hout),
then¹([r(B)],Hin,Hout,ok) :-
    set_field(Hin,B,f,1,Hout).
then²([null],Hin,Hout,exc(Exc)) :-
    create_object(Hin,'NPE',Exc,Hout).
```

Fig. 12: CLP-translated program for method `mist` (Example 2)

the CLP derivation tree, denoted as $\mathcal{T}_m$, with root $m(In, Out, H_{in}, H_{out}, E)$ and initial constraint store $\theta = \{\}$ obtained using P'.

3.5 Test Case Generation

When handling realistic programs, it is well-known that the symbolic execution tree to be explored is in general infinite. This is because iterative constructs such as loops and recursion, whose number of iterations depend on input arguments, usually induce an infinite number of execution paths when executed with symbolic input values. It is therefore essential to establish a *termination criterion.* Such a termination criterion can be expressed in different forms. For instance, a computation time budget can be established, or an explicit bound on the depth of the symbolic execution tree can be imposed (called *depth-k* criterion). In this thesis, we adopt a more code-oriented termination criterion. Specifically, we impose an upper bound k on the number of times each loop is iterated. As a byproduct of imposing such a bound, the *loop-k* structural coverage criterion below is satisfied.

Finite symbolic execution tree, test case, and TCG. Let us now establish definitions for key concepts of our approach:

Definition 3 (Finite symbolic execution tree, test case, and TCG). *Let m be the corresponding predicate for a method M in a CLP-translated program P, and let $\mathcal{C}$ be a termination criterion.*

- *$\mathcal{T}_m^{\mathcal{C}}$ is the finite and possibly incomplete symbolic execution tree of m with root $m(In, Out, H_{in}, H_{out}, EF)$ w.r.t. $\mathcal{C}$.*

Table 4: Test cases for method `remAll`

N	Input Heap	Output Heap	Constraint Store	EF
1	this l.first = null	this l.first = null	$\emptyset$	ok
2	this.first = null l.first $\rightarrow \boxed{A} \rightarrow$ null	this.first = null l.first $\rightarrow \boxed{A} \rightarrow$ null	$\emptyset$	ok
3	this.first $\rightarrow \boxed{A} \longrightarrow$ null l.first $\longrightarrow \boxed{B} \longrightarrow$ null	this.first $\rightarrow \boxed{A} \longrightarrow$ null l.first $\longrightarrow \boxed{B} \longrightarrow$ null	$\{A \neq B\}$	ok
4	this.first $\rightarrow \boxed{A} \longrightarrow$ null l.first $\longrightarrow \boxed{A} \longrightarrow$ null	this.first = null l.first $\longrightarrow \boxed{A} \longrightarrow$ null	$\emptyset$	ok
5	this l $\longrightarrow$ null	-	$\emptyset$	exc

6	this.first $\rightarrow \boxed{A} \longrightarrow$ null
	l = this
		this.first = null
		l = this
			$\emptyset$	ok

7	this.first $\rightarrow \boxed{A} \longrightarrow$ null l.first $\nearrow$	this.first = null l.first $\rightarrow \boxed{A} \rightarrow$ null	$\emptyset$	ok

– *Let b be a successful (terminating) path in $\mathcal{T}_m^{\mathcal{C}}$. A test case for m w.r.t. $\mathcal{C}$ is a 6-tuple of the form: $\langle \sigma(In), \sigma(Out), \sigma(H_{in}), \sigma(H_{out}),\ \sigma(EF), \theta\rangle$, where σ and θ are, resp., the substitution and the constraint store associated to b.*

– *TCG is the process of generating the set of test cases obtained for all successful (terminating) paths in $\mathcal{T}_m^{\mathcal{C}}$.*

In the remainder of this dissertation, we comply with the above abstract (symbolic) definition of *test case*, hence adopting a non-standard use of the term "test case". Standard test cases are concrete, i.e., actual input values on which the program under test can be run. In contrast, in this thesis a *test case* represents the class of inputs that will follow the same execution path, characterized by a path condition (and symbolic expressions for variables). A *test suite* is hence a set of test cases that characterizes all symbolic execution paths explored by symbolic execution using a particular termination criterion. Nevertheless, it is possible to produce actual values from the obtained *symbolic* test cases. This can be done in a straightforward subsequent stage in our framework. Namely, we can use the *labeling* mechanisms of standard *clpfd* domains to assign concrete values to all variables which satisfy the path condition, thus solving it. As a result of this last step, concrete and executable test cases are obtained.

Example 10. The test suite generated for method `remAll` for a *loop-1* coverage criterion is shown in Table 4. The first 5 cases are generated without considering aliasing of references. By doing so, the last two cases are also generated. Let us explain in detail three of the obtained test cases:

- **Case 3**. Corresponds to the path in which both the `this` list and the input list `l` contain just one element. The constraint $\{A \neq B\}$ indicates that fields `this.first.data` and `l.first.data` must have different values. The output heap is the same as the input heap, which means that the heap remains unchanged at the end of the execution path represented by this test case (although it may have suffered changes in intermediate derivations).
- **Case 4**. The input heap is the almost same as in case 3, but here, the symbolic variables corresponding to `this.first.data` and `l.first.data` are unified (variable A), meaning that their values are the same. In the output heap, notice that the first node from the `this` list has been removed.
- **Case 7**. Reference fields `this.first` and `l.first` are aliased. That is, they point to the same `Node` object in the heap. Removing element `A` from the `this` list boils down to setting reference `this.first` to `null`, leaving the object in the heap intact.

Finally, as mentioned before, by solving the constraint system and applying labeling on the variables involved, concrete inputs can be obtained. A concrete instantiation for this test case would consist of the following input heap $\{\mathsf{this.first} \rightarrow 1 \longrightarrow \mathsf{null}, \mathsf{l.first} \longrightarrow 2 \longrightarrow \mathsf{null}\}$ where variables A and B have been assigned concrete values 1 and 2, respectively, such that the constraint store $A \neq B$ is satisfied. As the test case specifies, the heap in the concrete output state remains unchanged. □

The PET System. PET (Partial Evaluation-based Test case generator) is a system that implements the CLP-based TCG framework described in this chapter. It is is fully implemented in SWI-Prolog [48] and uses the CLP(FD) library [47] (Constraint Logic Programming over Finite Domains) as constraint solver. Some of the important features of the PET system are:

- It is generic. Provided that appropriate CLP translations are available, PET can work with other imperative object-oriented languages. That is, once the CLP translation is done, the language features are abstracted away. That is to say, the TCG phase of the approach implemented in PET is language independent. In this way, we elude the difficulties of explicitly dealing with features like recursion, procedure calls, dynamic memory allocation, exceptions, etc., whose treatment may differ from one language to another.
- It is flexible. Different termination (coverage) criteria can be easily incorporated to the PET system. These criteria are written in PET as predicates which are permanently checked during TCG. Adding new criteria consists in implementing such a predicate, which requires only basic knowledge of logic programming.
- It is incremental. One of the artifacts that the PET system generates is a test case generator. To the best of our knowledge, this is a unique feature in a TCG tool nowadays. Namely, PET allows to extend test suites by exploring further in the symbolic execution tree in an on-demand fashion. In

other words, PET allows to incrementally relax the imposed termination criterion to explore symbolic execution paths that were initially pruned by the termination criterion.

The PET system is available for download as open-source software and for online use through its web interface at http://costa.ls.fi.upm.es/pet. Furthermore, an Eclipse plugin called jPET [3] is available. jPET supports full sequential Java and some of its interesting features are:

– Interactive test case visualization. jPET integrates a test case viewer to allow an intuitive, interactive visualization of the information contained in test cases. This includes objects and arrays involved in the input and output heap terms.
– Trace highlighting. On selection of a particular test case, jPET highlights the sequence of instructions in the original Java source code that the test case exercises. Alternatively, a *trace debugging* feature allows for a step-by-step highlighting of the source code, as in the traditional style of code debugging.
– Parsing of method preconditions written in JML [28]. jPET enables the specification of conditions on the input arguments of methods. These conditions are written in a subset of JML (Java Modeling Language), the standard specification language within software verification of Java. Using preconditions allows steering symbolic execution towards interesting parts of the program under test, ignoring others that are less interesting.
– Generation of JUnit. JUnit is a Unit Testing Framework for Java, which provides a set of classes to support writing, executing and reusing test cases. jPET generates self-contained JUnit test cases, as shown in Example 4. Whereas those unit tests therein are rather simple, the generation of JUnit code for heap-manipulating programs is much more challenging, as it often involves the need to synthesize the input and output heaps and compare the output heap stored in the test case with the resulting heap after the execution of the test.

3.6 Guided CLP-based TCG

Whereas standard TCG by symbolic execution aims to cover *all* feasible paths of the program under test w.r.t. a termination criterion, in guided TCG, the termination criterion is combined with a *selection criterion*. To that end, the concept of *coverage criterion* is redefined to be a pair of two components $\langle TC, SC\rangle$. TC is a *termination criterion* that, as discussed earlier, ensures finiteness of symbolic execution. This can be done either based on execution steps or on loop iterations. Again, let us adhere to *loop-k*, which limits to a threshold k the number of allowed loop iterations and/or recursive calls (of each concrete loop or recursive method). SC is a *selection criterion* that determines which test cases the TCG must produce. In guided TCG this will steer symbolic execution towards the paths that should be explored. In particular, we consider the following two coverage criteria:

- all-local-paths: It requires that all *local* execution paths within the method under test are exercised up to a *loop-k* limit. This has a potential interest in the context of unit testing, where each method must be tested in isolation.
- program-points(P): Given a set of program points P, it requires that all of them are exercised by at least one test case up to a *loop-k* limit. This criterion is the most appropriate choice for bug-detection and reachability verification purposes. A particular case of it is *statement coverage* (up to a limit), where all statements in a program or method must be exercised.

This section develops a concrete methodology to incorporate selection criteria into the CLP-based TCG framework. To that end, we could employ a post-processing phase where only the test cases that are sufficient to satisfy the selection criterion are selected by looking at their traces. This is however not an appropriate solution in general due to the exponential explosion of the paths that have to be explored in symbolic execution. Instead, we now aim at using the selection criterion to drive the TCG process towards satisfying paths, stressing to avoid as much as possible the exploration of irrelevant and redundant ones. The key idea that allows us to guide the TCG process is to pass *trace terms* as input arguments to symbolic execution. These trace terms can be complete or partial, which allows guiding completely or partially, the symbolic execution towards specific paths.

First, let us define the notion of *trace term* and update Definition 1 to add a trace term as an additional argument to each rule of the CLP-translated program, which enables us to keep track of the sequence of rules that are symbolically executed. Notice that trace terms are not cardinal components in the translated program, but rather a supplementary argument with a central role in this chapter.

Definition 4 (CLP-translated program with traces). *Given the rule of Definition 1, its CLP-translation with trace is:* $m(In, Out, H_{in}, H_{out}, EF, T) :- g, b'_1, \ldots, b'_n$.*" where:*

- *In, Out,* H_{in}*,* H_{out} *and EF remain as in Definition 1.*
- *T is the trace term for m of the form* $m(k, P, \langle T_{c_i}, \ldots, T_{c_m} \rangle)$*, where*
  - *P is the (possibly empty) list of trace parameters, i.e., the subset of the variables in rule* m^k *on which the resource consumption depends.*
  - $c_i, \ldots, c_m$ *is the (possibly empty) subsequence of method calls in* $b_1, \ldots, b_n$*.*
  - T_{c_j} *is a free logic variable representing the trace term associated to the call* c_j*.*
- *Calls in the body of the rule are extended with their corresponding trace terms, i.e., for all* $1 \leq j \leq n$*, if* $b_j \equiv p(I_p, O_p, H_{in_p}, H_{out_p})$*, then* $b'_j \equiv p(I_p, O_p, H_{in_p}, H_{out_p}, T_{c_j})$*; otherwise* $b'_j \equiv b_j$*.*

Now, let us revisit the definition of test case and TCG (Definition 3) to incorporate the notion of *trace* as an input argument for symbolic execution.

Definition 5 (Test case with trace and TCG). *Given a method m, a termination criterion* $\mathcal{C}$ *and a successful (terminating) path b in the symbolic execution tree* $\mathcal{T}_m^C$ *with root* $m(In, Out, H_{in}, H_{out}, EF, T)$*,* a test case with trace *for m* *w.r.t. $\mathcal{C}$ is a 6-tuple of the form: $\langle \sigma(In), \sigma(Out), \sigma(H_{in}), \sigma(H_{out}), \sigma(EF), \sigma(T), \theta \rangle$, where σ and θ are, resp., the set of bindings and the constraint store associated to b. TCG generates the set of test cases with traces obtained for all successful paths in $\mathcal{T}_m^C$.*

Trace-guided TCG. Given a method m, a coverage criterion $\mathcal{C} = \langle TC, SC \rangle$, and a (possibly partial) trace π, trace-guided TCG generates the set tgTCG of test cases obtained for all successful branches of m using π as a guiding input argument for symbolic execution. Observe that the TCG guided by one trace π generates: (a) exactly one test case if π is complete and corresponds to a feasible path; (b) none if π is unfeasible; or (c) possibly several test cases if π is partial. In the latter case the traces of all test cases are instantiations of the partial trace.

For convenience, let us also define firstOf-tgTCG(m, TC, π) to be the unary set containing the leftmost successful branch of the symbolic execution tree of m. Now, by relying on the existence of a trace generator $TraceGen$ that generates, on demand and one by one, (possibly partial) traces according to $\mathcal{C}$, we define in Algorithm 1 a generic scheme for guided TCG.

Algorithm 1 Generic scheme for guided TCG

```
Input: M, and $\langle TC, SC \rangle$
TestCases = {}
while TraceGen has more traces and TestCases does not satisfy SC
   Ask TraceGen to generate a new trace in Trace
   TestCases = TestCases $\cup$ firstOf-tgTCG(M,TC,Trace)
Output: TestCases
```

The intuition is as follows: the trace generator generates a trace, possibly using for that SC, TC and the current $TestCases$. If the generated trace is feasible, then the first solution of its trace-guided TCG is added to the set of test cases. The process finishes either when SC is satisfied, or when the trace generator has already generated all traces up to TC. If the trace generator is complete (see below), this means that SC cannot be satisfied within the limit imposed by TC. Observe that for some selection criteria, e.g., all-local-paths, the calls to firstOf-tgTCG can be computed in parallel.

Example 11. Figure 13a shows a Java program made up of three methods: `lcm` calculates the least common multiple of two integers, `gcd` calculates the greatest common divisor of two integers, and `abs` returns the absolute value of an integer. Figure 13b shows the equivalent CLP-translated program. Method `lcm` is translated into predicates `lcm`, `cont`, `try` and `div`. As per Section 3.2, the translation preserves the control flow of the program and transforms iteration into recursion (e.g. method `gcd`). Note that the example has been chosen deliberately small and simple to ease comprehension. Let us consider the TCG for method `lcm` with program-points for points μ and κ as selection criterion. Let us assume that

```
int lcm(int a,int b) {
  if (a < b) {
    int aux = a;
    a = b;
    b = aux;
  }
  int d = gcd(a,b);
  try {
    return abs(a*b)/d;
  } catch (Exception e) {
    return -1;              μ
  }
}

int gcd(int a,int b) {
  int res;
  while (b != 0) {
    res = a%b;
    a = b;
    b = res;
  };
  return abs(a);
}

int abs(int a) {
  if (a >= 0)
    return a;               κ
  else
    return -a;
}
```

(a) Java source code

```
lcm([A,B],[R],_,_,E,lcm(1,[T])) :-
    A #>= B,
    cont([A,B],[R],_,_,E,T).
lcm([A,B],[R],_,_,E,lcm(2,[T])) :-
    A #< B,
    cont([B,A],[R],_,_,E,T).
cont([A,B],[R],_,_,E,cont(1,[T,V])) :-
    gcd([A,B],[G],_,_,E,T),
    try([A,B,G],[R],_,_,E,V).
try([A,B,G],[R],_,_,E,try(1,[T,V])) :-
    M #= A*B,
    abs([M],[S],_,_,E,T),
    div([S,G],[R],_,_,E,V).
try([A,B,G],[R],_,_,exc,try(2,[])).
div([A,B],[R],_,_,ok,div(1,[])) :-
    B #\= 0,
    R #= A/B.
div([A,0],[-1],_,_,catch,div(2,[])).   μ
gcd([A,B],[D],_,_,E,gcd(1,[T])) :-
    loop([A,B],[D],_,_,E,T).
loop([A,0],[F],_,_,E,loop(1,[T])) :-
    abs([A],[F],_,_,E,T).
loop([A,B],[E],_,_,G,loop(2,[T])) :-
    B #\= 0,
    body([A,B],[E],_,_,G,T).
body([A,B],[R],_,_,E,body(1,[T])) :-
    B #\= 0,
    M #= A mod B,
    loop([B,M],[R],_,_,E,T).
body([A,0],[R],_,_,exc,body(2,[])).
abs([A],[A],_,_,ok,abs(1,[])) :-
    A #>= 0.                           κ
abs([A],[-A],_,_,ok,abs(2,[])) :-
    A #< 0.
```

(b) CLP-translation

Fig. 13: Guided TCG Example: Java (left) and CLP-translated (right) programs.

the trace generator starts generating the following two traces:

t_1 : `lcm(1,[cont(1,[G,check(1,[A,div(2,[])])])])`
t_2 : `lcm(2,[cont(1,[G,check(1,[A,div(2,[])])])])`

The first iteration does not add any test case since trace t_1 is unfeasible. Trace t_2 is proved feasible and a test case is generated. The selection criterion is now satisfied and therefore the process finishes. The following test case is obtained for the program-points criterion for method `lcm` and program points μ and κ. This particular case illustrates specially well how guided TCG can reduce the number of produced test cases through adequate control of the selection criterion.

Constraint store	*Trace*
`{A=B=0,Out=-1}`	`lcm(1,[cont(1,[gcd(1,[loop(1,[abs(1,[])])]), try(1,[abs(1,[]),div(2,[])])])])`

□

There are two properties of high importance in guided TCG, *completeness* and *effectiveness*. Intuitively, a concrete instantiation of the guided TCG scheme is *complete* if it never reports that the coverage criterion is not satisfied when it is indeed satisfiable. *Effectiveness* is related to the number of iterations the algorithm performs. These two properties depend completely on the trace generator. A trace generator is *complete* if it produces an over-approximation of the set of traces satisfying the coverage criterion. Its *effectiveness* depends on the number of redundant and/or unfeasible traces it generates: the larger the number, the less effective the trace generator.

Trace Generators for Structural Coverage Criteria. Let us now describe a general approach to build complete and effective trace generators for structural coverage criteria by means of program transformations. Then, we describe in detail an instantiation for the all-local-paths coverage criteria.

The *trace-abstraction* of a program can be defined as follows. Given a CLP-translated program with traces P, its trace-abstraction is obtained as follows: for every rule of P, (1) remove all atoms in the body of the rule except those corresponding to rule calls, and (2) remove all arguments from the head and from the surviving atoms of (1) except the last one (i.e., the trace term).

Example 12. Figure 14 shows the trace-abstraction for the CLP-translated program of Figure 13b. Observe that the trace-abstraction basically corresponds the control-flow graph of the CLP-translated program. □

The trace-abstraction can be directly used as a trace generator as follows: (1) Apply the termination criterion in order to ensure finiteness of the process. (2) Select, in a post-processing, those traces that satisfy the selection criterion. Such a trace generator produces on backtracking a superset of the set of traces of the program satisfying the coverage criterion. Note that, this can be done as long as the criteria are structural. The obtained trace generator is by definition complete. However, it can be very ineffective and inefficient due to the large number of unfeasible and/or unnecessary traces that it can generate.

In the following, we propose a concrete, and more effective, instantiation for the all-local-paths coverage criteria. As we will see, this is done by taking advantage of the notion of partial traces and the implicit information on the concrete coverage criteria. A concrete instantiation for the program-points coverage criteria is described at [39].

```
lcm(lcm(1,[T])) :- cont(T).
lcm(lcm(2,[T])) :- cont(T).
cont(cont(1,[T,V])) :- gcd(T), try(V).
try(try(1,[T,V])) :- abs(T), div(V).
try(try(2,[])).
div(div(1,[])).
div(div(2,[])).
gcd(gcd(1,[T])) :- loop(T).
loop(loop(1,[T])) :- abs(T).
loop(loop(2,[T])) :- body(T).
body(body(1,[T])) :- loop(T).
body(body(2,[])).
abs(abs(1,[])).
abs(abs(2,[])).
```

Fig. 14: Trace-abstraction

An Instantiation for the all-local-paths Coverage Criterion. Let us start from the trace-abstraction program and apply a syntactic program slicing which removes from it the rules that do not belong to the considered method.

Definition 6 (slicing for all-local-paths coverage criterion). *Given a trace-abstraction program P and an entry method M:*

1. *Remove from P all rules that do not belong to method M.*
2. *In the bodies of remaining rules, remove all calls to rules which are not in P.*

The obtained sliced trace-abstraction, together with the termination criterion, can be used as a trace generator for the all-local-paths criterion for a method. The generated traces will have free variables in those trace arguments that correspond to the execution of other methods, if any.

```
lcm(lcm(1,[T])) :- cont(T).
lcm(lcm(2,[T])) :- cont(T).
cont(cont(1,[G,T])) :- try(T).
try(try(1,[A,T])) :- div(T).
try(try(2,[])).
div(div(1,[])).
div(div(2,[])).
```

```
lcm(1,[cont(1,[G,try(1,[A,div(1,[])])])])
lcm(1,[cont(1,[G,try(1,[A,div(2,[])])])])
lcm(1,[cont(1,[G,try(2,[])])])
lcm(2,[cont(1,[G,try(1,[A,div(1,[])])])])
lcm(2,[cont(1,[G,try(1,[A,div(2,[])])])])
lcm(2,[cont(1,[G,try(2,[])])])
```

Fig. 15: Slicing of method `lcm` for all-local-paths criterion.

Example 13. Figure 15 shows on the left the sliced trace-abstraction for method `lcm`. On the right is the finite set of traces that is obtained from such trace-abstraction for any *loop-k* termination criterion. Observe that the free variables `G`, resp. `A`, correspond to the sliced away calls to methods `gcd` and `abs`. □

Let us define the predicates: `computeSlicedProgram(M)`, that computes the sliced trace-abstraction for method M as in Definition 6; `generateTrace(M,TC,Trace)`, that returns in its third argument, on backtracking, all partial traces computed using such sliced trace-abstraction, limited by the termination criterion `TC`; and `traceGuidedTCG(M,TC,Trace,TestCase)`, which computes on backtracking the set `tgTCG` (definition of Trace-guided TCG above), failing if the set is empty, and instantiating on success `TestCase` and `Trace` (in case it was partial). The guided TCG scheme in Algorithm 1, instantiated for the all-local-paths criterion, can be implemented in Prolog as follows:

```
(1) guidedTCG(M,TC) :-
(2)     computeSlicedProgram(M),
(3)     generateTrace(M,TC,Trace),
(4)     once(traceGuidedTCG(M,Trace,TC,TestCase)),
(5)     assert(testCase(M,TestCase,Trace)),
(6)     fail.
(7) guidedTCG(_,_).
```

Intuitively, given a (possibly partial) trace generated in line `(3)`, if the call in line `(4)` fails, then the next trace is tried. Otherwise, the generated test case is asserted with its corresponding trace which is now fully instantiated (in case it was partial). The process finishes when `generateTrace/3` has computed all traces, in which case it fails, making the program exiting through line `(7)`.

Example 14. The following test cases are obtained for the all-local-paths criterion for method `lcm`:

Constraint store	*Trace*
`{A>=B}`	`lcm(1,[cont(1,[gcd(1,[loop(1,[abs(1,[])])]), try(1,[abs(1,[]),div(1,[])])])])`
`{A=B=0,Out=-1}`	`lcm(1,[cont(1,[gcd(1,[loop(1,[abs(1,[])])]), try(1,[abs(1,[]),div(2,[])])])])`
`{B>A}`	`lcm(2,[cont(1,[gcd(1,[loop(1,[abs(1,[])])]), try(1,[abs(1,[]),div(1,[])])])])`

This set of 3 test cases achieves full code and path coverage on method `lcm` and is thus a perfect choice in the context of unit-testing. In contrast, the original, non-guided, TCG scheme with *loop-2* as termination criterion produces 9 test cases. □

A thorough experimental evaluation was performed in [39] which demonstrates the applicability and effectiveness of guided TCG.

4 TCG of Concurrent Programs

The focus of this section is on the development of automated techniques for testing *concurrent objects*.

4.1 Concurrent Objects

The central concept of the concurrency model is that of *concurrent object*. Concurrent objects live in a *distributed* environment with asynchronous and unordered communication by means of asynchronous method calls, denoted $y\ !\ m(\overline{z})$. Method calls may be seen as triggers of concurrent activity, spawning new tasks (so-called *processes*) in the called object. After an asynchronous call of the form $x{=}y\ !\ m(\overline{z})$, the caller may proceed with its execution without blocking on the call. Here x is a *future variable* which allows synchronizing with the completion of task $m(\overline{z})$. In particular, the instruction `await` $x?$ allows checking whether m has finished. In this case, execution of the current task proceeds and x can be used for accessing the return value of m via the instruction x.`get`. Otherwise, the current task releases the processor to allow another available task to take it.

A synchronous call of the form $x = y.m(\overline{z})$, is internally transformed into the statement sequence $w = y\ !\ m(\overline{z})$; `if` $(this == y)$ `await` $w?$; $x = w$.`get`, where w is a fresh future variable. This is because when the synchronous call is executed on the same object *this* we do not want to block this object (this would lead to a deadlock on the object *this*). Instead, we use an `await` instruction that will allow that the execution of the synchronous call to m can start to execute. The statement $x = w$.`get` blocks the execution of the current object until $m(\overline{z})$ on y returns a value. The `if` statement avoids a deadlock when the object y is equal to *this*. For simplicity we assume that all methods return a value.

Example 15. Fig. 16 shows the implementation of class A, which contains two integer fields and five methods. Method sumFacts computes $\sum_{k=ft}^{ft+(n-1)} k!$ by asynchronously invoking fact on object ob. The `await` instruction before entering the loop allows releasing the processor if ft is negative. Once ft takes a non-negative value, the task can resume its execution and enter the loop. For instance, the asynchronous call $\mathsf{f} = \mathsf{ob}\ !\ \mathsf{fact}(3, \mathsf{this})$; in sumFacts will add the task $\mathsf{fact}(3, \mathsf{this})$ to the queue of ob. When this task starts executing, it will add the task $\mathsf{fact}(2, \mathsf{ob})$ on the object this, which in turn will add $\mathsf{fact}(1, \mathsf{this})$ on ob and so on, in such a way that the factorial is computed in a distributed way between the two objects. Note that the calls are synchronized on future variables. This means that until the recursive call $\mathsf{fact}(1, \mathsf{this})$ is not completed the other tasks are suspended on their corresponding `await` conditions. □

Let us briefly present the semantics for the concurrency instructions. An *object* is a term $\mathtt{ob}(o, t, h, \mathcal{Q})$ where o is the object identifier, t is the identifier of the *active task* that holds the object's lock or $\perp$ if the object's lock is free, h is its local heap and $\mathcal{Q}$ is the set of tasks in the object. A *task* is a term $tk(t, m, l, s)$

```
class A(Int n, Int ft) {
Int sumFacts(A ob) {
  Fut<Int> f; Int res=0;
  Int m = this.n;
  await this.ft >= 0;
  while (m > 0) {
    f =ob ! fact(this.ft, this);
    await f ?;
    Int a = f.get;
    res = res + a;
    this.ft = this.ft + 1;
    m = m - 1;
  }
  return res;
}
Int fact(Int k, A ob){
  Fut <Int> f; Int res = 1;
  if (k <= 0) res = 1;
  else { f = ob ! fact(k - 1,this);
        await f ?; res = f.get;
        res = k * res;
  }
  return res;
}
Int setN(Int a)  { this.n=a; return 0; }
Int setFt(Int b) { this.ft=b; return 0; }
Int set(Int a, Int b){
  this.setN(a); this.setFt(b);
  return 0;
}
```

Fig. 16: ABS running example.

where t is a unique task identifier, m is the method name executing in the task, l is a mapping from local variables to their values, and s is the sequence of instructions to be executed or ϵ if the task has terminated.

A *state* or *configuration* S has the form $o_0 \cdot o_1 \cdot \dots \cdot o_n$, where $o_i \equiv \mathsf{ob}(o_i, t_i, h_i, \mathcal{Q}_i)$. The execution of a program from a method m starts from an initial state $S_0 = \{\mathsf{ob}(0, 0, \perp, \{tk(0, m, l, body(m))\}$. Here, l maps parameters to their initial values (null in case of reference variables), $body(m)$ is the sequence of instructions in method m, and $\perp$ stands for the empty heap.

Fig. 17 presents the semantics of the concurrent objects. As objects do not share their states, the semantics can be presented as a macro-step semantics [41] (defined by means of the transition "$\longrightarrow$") in which the evaluation of all statements of a task takes place serially (without interleaving with any other task) until it gets to a *release point*, i.e., a point in which the object's processor becomes idle $\perp$ (due to an `await` or `return` instruction). In this case, we apply rule MSTEP to select an available task from an object, namely we apply the function $selectObject(S)$ to select non-deterministically one object in the state with a non-empty queue $\mathcal{Q}$ and $selectTask(\mathcal{Q})$ to select non-deterministically one task of $\mathcal{Q}$.

The transition $\leadsto$ defines the evaluation within a given object. We sometimes label transitions with $o \cdot t$, the name of the object o and task t selected (in rule MSTEP) or evaluated in the step (in the transition $\leadsto$). The notation $h[\bar{f} \mapsto l(\bar{y})]$ (resp. $l[x \mapsto v]$) stands for the result of storing $l(\bar{y})$ in the fields $\bar{f}$ (resp. v in x).

The remaining sequential instructions are standard and thus omitted. In NEWOB, an active task t in object o creates an object o' of class D which is introduced to the state with a free lock. Here h' stands for a default mapping on

$$(\textsc{mstep})\ \frac{\mathit{selectObject}(S) = \mathtt{ob}(o, \bot, h, \mathcal{Q}), \mathcal{Q} \neq \emptyset, \mathit{selectTask}(\mathcal{Q}) = t, S \overset{o\cdot t}{\leadsto^*} S'}{S \xrightarrow{o\cdot t} S'}$$

$$(\textsc{newob})\ \frac{t{=}\mathit{tk}(t, m, l, x{=}\mathtt{new}\ D(\bar{y}); s), \text{fresh}(o'), h'{=}\mathit{newhp}(D), l'{=}l[x{\to}o'], \mathtt{class}\ D(\bar{f})}{\mathtt{ob}(o, t, h, \mathcal{Q}{\cup}\{t\}) \leadsto \mathtt{ob}(o, t, h, \mathcal{Q} \cup \{\mathit{tk}(t, m, l', s)\}) \cdot \mathtt{ob}(o', \bot, h'[\bar{f}{\mapsto}l(\bar{y})], \{\})}$$

$$(\textsc{async})\ \frac{t = \mathit{tk}(t, m, l, y{=}x\ !\ m_1(\bar{z}); s), l(x){=}o_1, \text{fresh}(t_1), l_1{=}\mathit{buildLocals}(\bar{z}, m_1, l)}{\begin{array}{c}\mathtt{ob}(o, t, h, \mathcal{Q} \cup \{t\}) \cdot \mathtt{ob}(o_1, \_, \_, \mathcal{Q}') \leadsto \\ \mathtt{ob}(o, t, h, \mathcal{Q}{\cup}\{\mathit{tk}(t, m, l[y{\mapsto}t_1], s)\}) \cdot \mathtt{ob}(o_1, \_, \_, \mathcal{Q}'{\cup}\{\mathit{tk}(t_1, m_1, l_1, \mathit{body}(m_1))\})\end{array}}$$

$$(\textsc{await}1)\ \frac{t = \mathit{tk}(t, m, l, \langle\mathtt{await}\ y?; s\rangle), l(y) = t_1,\ \mathit{tk}(t_1, \_, \_, s_1) \in \mathtt{Objects}, s_1 = \epsilon(v)}{\mathtt{ob}(o, t, h, \mathcal{Q} \cup \{t\}) \leadsto \mathtt{ob}(o, t, h, \{\mathit{tk}(t, m, l, s)\} \cup \mathcal{Q})}$$

$$(\textsc{await}2)\ \frac{t = \mathit{tk}(t, m, l, \langle\mathtt{await}\ y?; s\rangle), l(y) = t_1,\ \mathit{tk}(t_1, \_, \_, s_1) \in \mathtt{Objects}, s_1 \neq \epsilon(v)}{\mathtt{ob}(o, t, h, \mathcal{Q} \cup \{t\}) \leadsto \mathtt{ob}(o, \bot, h, \{\mathit{tk}(t, m, l, \langle\mathtt{await}\ y?; s\rangle)\} \cup \mathcal{Q})}$$

$$(\textsc{get})\ \frac{t = \mathit{tk}(t, m, l, \langle x = \mathtt{get}.y; s\rangle), l(y) = t_1,\ \mathit{tk}(t_1, \_, \_, s_1) \in \mathtt{Objects}, s_1 = \epsilon(v)}{\mathtt{ob}(o, t, h, \mathcal{Q} \cup \{t\}) \leadsto \mathtt{ob}(o, t, h, \{\mathit{tk}(t, m, l[x \mapsto v], s)\} \cup \mathcal{Q})}$$

$$(\textsc{return})\ \frac{t = \mathit{tk}(t, m, l, \mathtt{return}\ x; s)}{\mathtt{ob}(o, t, h, \mathcal{Q} \cup \{t\}) \leadsto \mathtt{ob}(o, \bot, h, \{\mathit{tk}(t, \_, \_, \epsilon(l(x)))\} \cup \mathcal{Q})}$$

Fig. 17: Summarized Semantics for Distributed and Concurrent Execution

the fields of class D initialized with the values of $l(\bar{y})$. ASYNC spawns a new task (the initial state is created by *buildLocals*) with a fresh task identifier t_1 which is associated to the corresponding future variable y in l. We have assumed that $o \neq o_1$, but the case $o = o_1$ is analogous, the new task t_1 is simply added to $\mathcal{Q}$ of o_1.

The remaining rules define the concurrent execution within each distributed object. In AWAIT1, the future variable we are awaiting for points to a finished task and the `await` can be completed. The finished task t_1 is looked up in all objects in the current state (denoted `Objects`). Otherwise, AWAIT2 yields the lock so that any other task of the same object can take it. GET blocks the object until the task is finished. When RETURN is executed, the return value is stored in v so that it can be obtained by the future variable that points to that task. Besides, the lock is released and will never be taken again by that task. Consequently, that task is *finished* (marked by adding the instruction $\epsilon(v)$) but it does not disappear from the state as its return value may be needed later on in an `await`. In what follows, a *derivation* $S_0 \longrightarrow \cdots \longrightarrow S_n$ from an initial state S_0 of an object system is a sequence of macro-steps (applications of rule MSTEP). Since the execution is non-deterministic, multiple derivations are possible from an initial state.

Example 16. For instance, let us consider the following code corresponding to some method m of some class B.

(a) A x = new A(5,10);
(b) Fut<Int> f;
(c) f = x ! fact(2,x);
(d) `await` f?;
(e) z = f.get;

where class A is that in Ex. 15. We start from the initial state $S_0 = \{\mathtt{ob}(0, 0, \bot, \{tk(0, m, l_0, (a) \cdots (e))\}$. By applying consecutively rules NEWOB, ASYNC and AWAIT2 to (a), (c) and (d) respectively we get:

$$S_1 = \{\mathtt{ob}(0, 0, \bot, \{tk(0, m, l_0, (d) \cdot (e))\}), \mathtt{ob}(1, \bot, h_1, \{tk(2, \mathsf{fact}, l_2, body(\mathsf{fact}))\})\}$$

where $l_0(\mathsf{f}) = 2$, $l_2(\mathsf{k}) = 2, l_2(\mathsf{ob}) = 1$ and $h_1(\mathsf{n}) = 5, h_2(\mathsf{ft}) = 10$. We apply now a macro step on object 1, by reducing task 2. In this case the macro step stops when executing `await` f? of method fact, and the state is modified as follows:

$$\begin{aligned} S_2 = \{\ & \mathtt{ob}(0, 0, \bot, \{tk(0, m, l_0, (d) \cdot (e))\}), \\ & \mathtt{ob}(1, 2, h_1, \{tk(2, \mathsf{fact}, l_2, \mathtt{await}\ \mathsf{f}?; \ldots), tk(3, \mathsf{fact}, l_3, body(\mathsf{fact}))\})\} \end{aligned}$$

where $l_2(\mathsf{f}) = 3$, $l_3(\mathsf{k}) = 1, l_3(\mathsf{ob}) = 1$. Similarly as done before, task with identifier 3 is now reduced, stopping the derivation when we reach `await` f?:

$$\begin{aligned} S_3 = \{\ & \mathtt{ob}(0, 0, \bot,\ \{tk(0, m, l_0, (d) \cdot (e))\}), \\ & \mathtt{ob}(1, 2, h_1, \{tk(2, \mathsf{fact}, l_2, \mathtt{await}\ \mathsf{f}?; \ldots), \\ & \qquad tk(3, \mathsf{fact}, l_3, \mathtt{await}\ \mathsf{f}?; \ldots), tk(4, \mathsf{fact}, l_4, body(\mathsf{fact}))\})\} \end{aligned}$$

where $l_3(\mathsf{f}) = 4$, $l_4(\mathsf{k}) = 0, l_4(\mathsf{ob}) = 1$. Now only task 4 can be reduced and applying rule RETURN we get:

$$\begin{aligned} S_4 = \{\ & \mathtt{ob}(0, 0, \bot,\ \{tk(0, m, l_0, (d) \cdot (e))\}), \\ & \mathtt{ob}(1, 2, h_1, \{tk(2, \mathsf{fact}, l_2, \mathtt{await}\ \mathsf{f}?; \ldots), \\ & \qquad tk(3, \mathsf{fact}, l_3, \mathtt{await}\ \mathsf{f}?; \ldots), tk(4, \bot, l_4, \epsilon(1))\})\} \end{aligned}$$

Now, task 3 can be reduced by applying first AWAIT1 and after RETURN:

$$\begin{aligned} S_5 = \{\ & \mathtt{ob}(0, 0, \bot,\ \{tk(0, m, l_0, (d) \cdot (e))\}), \\ & \mathtt{ob}(1, 2, h_1, \{tk(2, \mathsf{fact}, l_2, \mathtt{await}\ \mathsf{f}?; \ldots), \\ & \qquad tk(3, \bot, l_3, \epsilon(1)), tk(4, \bot, l_4, \epsilon(1))\})\} \end{aligned}$$

Similarly we reduce task 2 and after task 0 from object 0 and we finally get:

$$\begin{aligned} S_6 = \{\ & \mathtt{ob}(0, 0, \bot,\ \{tk(0, m, l_0, \epsilon)\}), \\ & \mathtt{ob}(1, 2, h_1, \{tk(2, \bot, l_2, \epsilon(2)), \\ & \qquad tk(3, \bot, l_3, \epsilon(1)), tk(4, \bot, l_4, \epsilon(1))\})\} \end{aligned}$$

where $l_0(z) = 2$. □

Given an initial state, a naïve exploration of the search space to reach all possible system configurations does not scale. The challenge is then in avoiding the exploration of redundant states which lead to the same configuration. Partial-order reduction (POR) [16,20] is a general theory that helps mitigate the state-space explosion problem by exploring the subset of all possible interleavings which lead to a different configuration. A concrete algorithm (called DPOR) was proposed by Flanagan and Godefroid [18] which maintains for each configuration a backtrack set, which is updated during the execution of the program when it realizes that a non-deterministic choice must be tried. Recently, TransDPOR [45] extends DPOR to take advantage of the transitive dependency relations in actor systems to explore fewer configurations than DPOR. As noticed in [32,45], their effectiveness highly depend on the actor selection order.

In our semantics in Fig. 16, functions *selectObject* and *selectTask* can be implemented with novel strategies and heuristics to further prune redundant state exploration, and they can be easily integrated within the aforementioned algorithms. For instance, *selectObject* could try to find a *stable object*, i.e., an object to which no other actor will post messages. Basically, this means that the object is autonomous since its execution does not depend on any other actor and thus no backtracking is required from that point. Furthermore, when temporal stability of any object cannot be proved, it is possible to look for heuristics that assign a weight to the messages according to the error that the object-selection strategy may make when proving stability w.r.t. them. Finally, function *selectTask* can be defined to select independent tasks according to the independence notion defined in [8], which basically establishes that two tasks are independent if they access disjoint parts of the shared memory. Note that this would avoid non determinism reordering among tasks.

4.2 Coverage and Termination Criteria for Concurrent Objects

As commented in Sec. 2.1, an important problem in symbolic execution is that, since the input data is unknown, the execution tree to be traversed is in general infinite. Hence it is required to integrate a *termination criterion* which guarantees that the length of the paths traversed remains finite while at the same time an interesting set of test cases is generated, i.e., certain code *coverage* is achieved.

Task-Level coverage and Termination Criteria. Given a task executing on an object, we aim at ensuring its local termination by leveraging existing Coverage Criteria (CC for short) developed in the sequential setting to the context of concurrent objects. We focus on the *loop-k* coverage criteria [26] described in Sec. 2.1, which limits the number of times we iterate on loops to a threshold $\mathsf{K_l}$ (other existing criteria would pose similar problems and solutions). However applying the task-level CC to all tasks *does not guarantee termination.* This is because we can switch from one task to another an infinite number of times. For example, consider the symbolic execution of $\mathsf{ob_1}$! $\mathsf{fact(n, ob_2)}$, where method fact is defined in Ex. 15. We circularly switch from object $\mathsf{ob_1}$ to object $\mathsf{ob_2}$ an infinite number of times because each asynchronous call in one object adds another call on the other object (see Ex. 16). This is not detected by the task-level CC because each method invocation is a new task. Intuitively, we get the following situation, where we show in each state the value of the queues in both objects. In each step the corresponding call to fibo is always selected.

$$
\begin{array}{ll}
\{\mathsf{ob}_1, \mathsf{ob}_2\},\ Q_{\mathsf{ob}_1}{=}\{\mathsf{fact}(\mathsf{n}, \mathsf{ob}_2)\}, Q_{\mathsf{ob}_2} = \{\} & \longrightarrow \\
\{\mathsf{ob}_1, \mathsf{ob}_2\},\ Q_{\mathsf{ob}_1}{=}\{\texttt{await f?}; \ldots\}, Q_{\mathsf{ob}_2} = \{\mathsf{fact}(\mathsf{n}_1, \mathsf{ob}_1)\} & \xrightarrow{n_1=n-1} \\
\{\mathsf{ob}_1, \mathsf{ob}_2\},\ Q_{\mathsf{ob}_1}{=}\{\mathsf{fact}(\mathsf{n}_2, \mathsf{ob}_2), \texttt{await f?}; \ldots\}, Q_{\mathsf{ob}_2} = \{\texttt{await f?}; \ldots\} & \xrightarrow{n_2=n_1-1} \\
\ldots \qquad \ldots & \longrightarrow \ldots
\end{array}
$$

The same problem can happen even with a single object, e.g., in method sumFacts when executing `await (ft >= 0)`, there is an infinite branch in the evaluation tree, corresponding to the case `ft < 0` which can be re-tried forever. I.e., we can apply infinitely the rule AWAIT2 in Fig. 17 on the task `await (ft >= 0)`, whose effect is to extract the task from the queue, to prove that the task does not hold, and to put again the task in the queue.

Task-Switching Coverage and Termination Criteria. In both examples above we can observe that the problem, in presence of concurrency relies, not only on loops, but also on the number of task switches allowed per object. Thus, the number of task switches can be limited by simply allowing a fixed and global number of task switching. However, it might happen that, due to excessive task switching in certain objects, others are not properly tested (i.e., their tasks exercised) because the *global* number of allowed task switches has been exceeded. For example, suppose that we add the instructions B ob_2 = new B(); ob_2 ! q(); before the return in method sumFacts, where B is a class that implements method q but whose code is not relevant. Then, as the evaluation for the *while* loop generates an infinite number of task switches (because of the `await` instruction in the loop), the evaluation of the call ob_2 ! p(); is not reached. Thus, in order to have fairness in the process and guarantee proper coverage from the concurrency point of view, we propose to *limit the number of task switches* per *object* (i.e., per concurrency unit).

4.3 Task Interleavings in TCG

An important problem in TCG of concurrent languages is that, when a task t suspends, there could be other tasks on the same object whose execution at this point could interleave with t and modify the information stored in the heap. It is essential to consider such task interleavings in order not to lose any important path. For example, let us consider a class C with two fields int n, f, and a method p in C defined as: int p(){n = 0; `await` (f > 0); if(n>=0) `return` 1; else `return` 2; }. Suppose a call of the form x = o ! p(); `await` x?; y = x.`get`. The symbolic execution of p, will in principle consider just one path (the one that goes through the if branch), giving as result always y = 1. There can be however another task (suspended in the queue of the object o) which executes when p suspends in `await` (f > 0) and writes a negative value on n. This would exercise the else branch when p resumes, giving as result y = 2. For example, suppose that the method void set(){n = −1; } belongs to class C and that set() is in the queue when executing `await` (f > 0), and that is executed before f > 0 holds. Then the execution of p() will try the else branch.

The questions that we solve in this section are: (a) is it possible to consider all interleavings that affect the method's coverage? (b) do we have means to discard useless interleavings? (i.e., those which do not add new paths). As regards (a), it is not enough to assume that there is one instance of each method call in the queue as further coverage is possible by introducing multiple instances of the same method. Even though termination is guaranteed by the limit imposed on the number of task switches in Sec. 4.2 (i.e., the length of the queue is finite), it is more appropriate to define an additional coverage criteria in this new dimension by fixing the maximum length of a queue in order to achieve a more meaningful coverage.

In order to answer question (b), we start by characterizing the notion of useless interleaving. Starting from the set of all methods in the class of the method under test, we propose a sequence of *prunings* which ensure that only useless interleavings are eliminated. The objective is to over-approximate, for each method m, the set related(m), which contains all methods whose interleaved execution with m can lead to a solution not considered before. The remaining ones are useless interleavings. Starting from the set of all methods in the class of the method under test, we propose a sequence of *prunings* which ensure that only useless interleavings are eliminated.

(Pruning 1) The first refinement is to discard methods which do not modify the heap, i.e., *pure* methods. Purity can be syntactically proved by checking that the method does not contain any instruction of the form $this.f = x$ and that methods (transitively) invoked from it are pure. Using this pruning on Ex. 15, we get $\mathsf{related}(\mathsf{sumFacts}) = \{\mathsf{sumFacts}, \mathsf{set}, \mathsf{setN}, \mathsf{setFt}\}$.

(Pruning 2) The second pruning amounts to considering only *directly impure* methods (ignoring transitive calls), i.e., those which write directly on fields. Let p be the method under test, m be a directly impure method and q a method that invokes m. The intuition is that by considering m alone, we execute it from a more general context, while its execution from q will be just more specific (since q will have added additional constraints). Hence, it will not add additional local traces for p. With this pruning, $\mathsf{related}(\mathsf{sumFacts}) = \{\mathsf{sumFacts}, \mathsf{setN}, \mathsf{setFt}\}$.

(Pruning 3) The third pruning consists in considering only the interleavings with those methods that write (directly) on fields which are used (read or written) before an `await` , *and* read after an `await`. These sets are easily computed by just looking for instructions $this.f = x$ and $x = this.f$ in the corresponding program fragments. Given a field f, the intuition for this condition is that, if f has not been accessed before the `await` then there is no information about the field. Thus, $\mathsf{related}(\mathsf{sumFacts}) = \{\mathsf{sumFacts}, \mathsf{setFt}\}$.

4.4 Related Work on TCG of Thread-based Concurrency

As it happens with actor-based systems, the main difficulties in TCG of thread-based systems are related to the scalability when considering thread interleavings. In thread-based systems, this problem is exacerbated. In [37], a symbolic execution framework which combines symbolic execution with model checking is presented to detect safety violations. Safety properties are represented by using logical formalisms understood by the model checker or that can be inserted in the code as annotations. The model checker, when doing symbolic execution, is able to report counterexamples which violate the correctness safety criterion. Furthermore, when generating test cases, the model checker generates the paths that fulfill the safety property. To reduce the number of thread interleavings, the model checker uses partial order reduction techniques [20] as we do. An advantage on this technique is the possibility of handling native calls through mixed concrete-symbolic solving. The main drawback of this framework is that satisfiability of constraints is checked at the end of each branch of the symbolic tree, what it might be unfeasible. Thus, they use preconditions on the symbolic input values in order to avoid the exploration of branches which violate the precondition. In contrast to [37], our CLP-approach is able to discard a branch in the symbolic tree once the associated constraint are unsatisfiable.

Other approaches that use techniques different from ours are [29,43,44]. The work [29] combines dynamic symbolic execution (concolic testing) with unfoldings. The unfolding approach allows constructing a compact representation of the interleavings and thus the new testing algorithm may use this information to guide the symbolic execution, avoiding irrelevant interleavings. This new approach achieves in some cases an exponential gain when compared with existing dynamic partial-order reduction based approaches [18,45]. Basically, the point is that in the previous approaches, the number of explored interleavings depends on the order in which processes are executed, but in this new approach it does not, since interleavings are computed a priory.

In [43,44], a runtime algorithm to monitor executions for multithreaded Java and possibly detect safety violations is presented. From a concrete execution, they automatically extract a partial order causality from a sequence of read-/write events on shared variables. Basically they extract, for a shared variable, the sequence of write/reads/write to that variable in the execution. Thus any permutation of these events can be considered an execution of the program if and only if it does not contradict the partial order. The main drawbacks is the state explosion since a large number of unreachable branches may be explored.

As an improvement of the previous work, in [42], a novel approach uses concolic execution (a combination of symbolic and concrete execution) to test shared-memory in multithreaded programs by using an algorithm based on race-detection and flipping. From a concrete execution, they determine the partial order relation or the exact race conditions between the processes in the execution path. Afterwards, such processes involved in races are flipped by generating new thread schedules and generating new test inputs. Hence, differently to the previous conservative approaches, in this work they explore one path from each partial order, avoiding possible warnings that could never occur in a real execution.

5 Conclusions

This tutorial summarizes the basic principles used in TCG by symbolic execution. It first discusses the main challenges that TCG currently poses: the efficient handling of heap-manipulating programs, compositionallity, and guiding the process. It then overviews a particular instantiation of the generic TCG framework that uses CLP as enabling technology. We will review the main features, advantages and implementation of this CLP-approach. Finally, we discuss the extension of the basic framework to handle concurrent actor systems.

Acknowledgments. This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu), by the Spanish projects TIN2008-05624 and TIN2012-38137, and by the CM project S2013/ICE-3006.

References

1. G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.
3. E. Albert, I. Cabañas, A. Flores-Montoya, M. Gómez-Zamalloa, and S. Gutiérrez. jPET: an Automatic Test-Case Generator for Java. In *WCRE'11*, pages 441–442. IEEE Computer Society, 2011.
4. Elvira Albert, María García de la Banda, Miguel Gómez-Zamalloa, José Miguel Rojas, and Peter Stuckey. A CLP Heap Solver for Test Case Generation. *Theory and Practice of Logic Programming*, 13(4-5):721–735, July 2013.
5. Elvira Albert, Miguel Gómez-Zamalloa, José Miguel Rojas, and Germán Puebla. Compositional CLP-based Test Data Generation for Imperative Languages. In María Alpuente, editor, *LOPSTR 2010 Revised Selected Papers*, volume 6564 of *Lecture Notes in Computer Science*, pages 99–116. Springer-Verlag, 2011.
6. S. Anand, E. K. Burke, T. Y. Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.
7. S. Anand, P. Godefroid, and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *TACAS'08*, LNCS 4963. Springer, 2008.
8. G. R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991.
9. C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, February 2013.
10. Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *ICSE'11*, pages 1066–1071. ACM, 2011.
11. L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

12. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.
13. F. Degrave, T. Schrijvers, and W. Vanhoof. Towards a Framework for Constraint-Based Test Case Generation. In *LOPSTR'09*, LNCS 6037, pages 128–142. Springer, 2010.
14. Agostino Dovier, Andrea Formisano, and Enrico Pontelli. A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems. In *Logic Programming*, LNCS 3668, pages 67–82. Springer, 2005.
15. C. Engel and R. Hähnle. Generating Unit Tests from Formal Proofs. In *TAP'07*, LNCS 4454, pages 169–188. Springer, 2007.
16. Javier Esparza. Model checking using net unfoldings. *Sci. Comput. Program.*, 23(2-3):151–195, 1994.
17. R. Ferguson and B. Korel. The Chaining Approach for Software Test Data Generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.
18. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121. ACM, 2005.
19. P. Godefroid. Compositional Dynamic Test Generation. In *POPL'07*, pages 47–54. ACM, 2007.
20. Patrice Godefroid. Using partial orders to improve automatic verification methods. In *CAV*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer, 1991.
21. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology*, 51(10):1409–1427, October 2009.
22. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *Theory and Practice of Logic Programming, ICLP'10 Special Issue*, 10(4–6):659–674, 2010.
23. A. Gotlieb, B. Botella, and M. Rueher. A CLP Framework for Computing Structural Test Data. In *CL'00*, LNAI 1861, pages 399–413. Springer, 2000.
24. N. Gupta, A. P. Mathur, and M. L. Soffa. Generating Test Data for Branch Coverage. In *ASE'00*, pages 219–228. IEEE Computer Society, 2000.
25. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.
26. W.E. Howden. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.
27. J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
28. The Java Modelling Language homepage. `http://www.eecs.ucf.edu/~leavens/JML//index.shtml`, 2013.
29. Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. Using unfoldings in automated testing of multithreaded programs. In Michael Goedicke, Tim Menzies, and Motoshi Saeki, editors, *ASE*, pages 150–159. ACM, 2012.
30. S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *TACAS'03*, LNCS 2619, pages 553–568. Springer, 2003.
31. J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.
32. Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. Evaluating Ordering Heuristics for Dynamic Partial-Order Reduction Techniques. In *FASE*

2010, volume 6013 of *Lecture Notes in Computer Science*, pages 308–322. Springer, 2010.
33. J.W. Lloyd. *Foundations of Logic Programming.* Springer, 2nd Ext. Ed., 1987.
34. K. Marriott and P. J. Stuckey. *Programming with Constraints: an Introduction.* MIT Press, 1998.
35. C. Meudec. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.
36. R. A. Müller, C. Lembeck, and H. Kuchen. A Symbolic Java Virtual Machine for Test Case Generation. In *IASTEDSE'04*, pages 365–371. ACTA Press, 2004.
37. Corina S. Pasareanu, Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehlitz, and Neha Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.
38. C. S. Păsăreanu and W. Visser. A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, 2009.
39. José Miguel Rojas and Miguel Gómez-Zamalloa. A Framework for Guided Test Case Generation in Constraint Logic Programming. In *LOPSTR 2012*, volume 7844 of *Lecture Notes in Computer Science*, pages 176–193. Springer, 2013.
40. José Miguel Rojas and Corina S. Păsăreanu. Compositional Symbolic Execution through Program Specialization. In *BYTECODE 2013, 8th Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, March 2013.
41. Koushik Sen and Gul Agha. Automated Systematic Testing of Open Distributed Programs. In *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2006.
42. Koushik Sen and Gul Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In Eyal Bin, Avi Ziv, and Shmuel Ur, editors, *Haifa Verification Conference*, volume 4383 of *Lecture Notes in Computer Science*, pages 166–182. Springer, 2006.
43. Koushik Sen, Grigore Rosu, and Gul Agha. Online efficient predictive safety analysis of multithreaded programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 123–138. Springer, 2004.
44. Koushik Sen, Grigore Rosu, and Gul Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Formal Methods for Open Object-Based Distributed Systems, 7th IFIP WG 6.1 International Conference, FMOODS 2005*, volume 3535 of *Lecture Notes in Computer Science*, pages 211–226. Springer, 2005.
45. S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In *FMOODS/FORTE*, volume 7273 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 2012.
46. N. Tillmann and J. de Halleux. Pex: White Box Test Generation for .NET. In *TAP'08*, LNCS 4966, pages 134–153. Springer, 2008.
47. Markus Triska. The Finite Domain Constraint Solver of SWI-Prolog. In *FLOPS*, LNCS 7294, pages 307–316, 2012.
48. J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.
49. H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.