A Framework for Guided Test Case Generation in Constraint Logic Programming

José Miguel Rojas[1] and Miguel Gómez-Zamalloa[2]

[1] Technical University of Madrid, Spain
[2] DSIC, Complutense University of Madrid, Spain

Abstract. Performing test case generation by symbolic execution on large programs becomes quickly impracticable due to the path explosion problem. A common limitation that this problem poses is the generation of unnecessarily large number of possibly irrelevant or redundant test cases even for medium-size programs. Tackling the path explosion problem and selecting high quality test cases are considered major challenges in the software testing community. In this paper we propose a constraint logic programming-based framework to guide symbolic execution and thus test case generation towards a more relevant and potentially smaller subset of paths in the program under test. The framework is realized as a tool and empirical results demonstrate its applicability and effectiveness. We show how the framework can help to obtain high quality test cases and to alleviate the scalability issues that limit most symbolic execution-based test generation approaches.

Keywords: Constraint Logic Programming, Guided Test Case Generation, Software Testing, Symbolic Execution, Trace-abstraction

1 Introduction

Testing remains a mostly manual stage within the software development process [4]. Test Case Generation (TCG) is a research field devoted to the automation of a crucial part of the testing process, the generation of input data. Symbolic Execution is nowadays one of the predominant techniques to automate the generation of input data. It is the underlying technique of several popular testing tools, both in academia and software industry [4].

Symbolic execution [10] executes a program with the contents of variables being symbolic formulas over the input arguments rather than concrete values. The outcome is a set of equivalence classes of inputs, each of them consisting of the constraints that characterize a set of feasible concrete executions of a program that takes the same path. A *test suite* is the set of *test cases* obtained from such path constraints by symbolically executing a program using a particular coverage criterion. Concrete instantiations of the test cases can be generated to obtain actual test inputs for the program, amenable for further validation by testing tools.

In spite of its popularity, it is well known that symbolic execution of large programs can become quickly impracticable due to the large number and the size of paths that need to be explored. This issue is considered a major challenge in the fields of symbolic execution and TCG [12]. Furthermore, a common limitation of TCG by symbolic execution is that it tends to produce an unnecessarily large number of test cases even for medium-size programs.

In previous work [1,8], we have developed a glass-box Constraint Logic Programming (CLP)-based approach to TCG for imperative object-oriented programs, which consists of two phases: First, the imperative program is translated into an equivalent CLP program by means of partial evaluation [7]. Second, symbolic execution is performed on the CLP-translated program, controlled by a termination criterion (in this context also known as coverage criterion), relying on CLP's constraint solving facilities and its standard evaluation mechanism, with extensions for handling dynamic memory allocation.

In this paper we develop a framework to *guide* symbolic execution in CLP-based TCG, dubbed Guided TCG. Guided TCG can serve different purposes. It can be used to discover bugs in a program, to analyze reachability of certain parts of a program, to lead symbolic execution to stress more interesting parts of the program, etc. This paper targets selective and unit testing. Selective testing aims at testing only specific paths of a program. Unit testing is a widely used software engineering methodology, where units of code (e.g. methods) are tested in isolation to validate their correctness. Incorporating the notion of selection criteria in our TCG framework represents one step towards fully supporting both unit and integration testing, a different methodology, where all the pieces of a system must be tested as a single unit.

Our Guided TCG is a heuristics that aims at steering symbolic execution, and thus TCG, towards specific program paths to generate more relevant test cases and filter less interesting ones with respect to a given selection criterion. The goal is 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. In particular, we develop two instances of the framework: one for covering all the local paths of a method, and the other to steer TCG towards a selection of program points in the program under test. Both instances have been implemented and we provide experimental results to substantiate their applicability and effectiveness.

The structure of the paper is as follows. Section 2 conveys the essentials of our CLP-based approach to TCG. Section 3 introduces the framework for guided TCG. Section 4 presents an instantiation of the framework based on trace-abstractions and targeting structural coverage criteria. Section 5 reports on the implementation and empirical evaluation of the approach. Section 6 discusses a complementary strategy to further optimize the framework. Finally, Section 7 situates our work in the existing research space, sketches ongoing and future work and concludes.

2 CLP-based Test Case Generation

Our CLP-based approach to TCG for imperative object-oriented programs essentially consists of two phases: (1) the program is translated into an equivalent CLP counterpart through partial evaluation; and (2) symbolic execution is performed on the CLP-translated program by relying on the CLP standard evaluation mechanisms. Details on the methodology can be found elsewhere [1, 7, 8].

2.1 CLP-translated Programs

All features of the imperative object-oriented program under test are covered by its equivalent *executable* CLP-translated counterpart. Essentially, there exists a one-to-one correspondence between blocks in the control flow of the original program and rules in the CLP counterpart:

Definition 1 (CLP-translated Program). *A CLP-translated program consists of a set of predicates, each of them defined by a set of mutually exclusive rules of the form* $m(I, O, H_i, H_o, E, T) : -[\bar{G},]b_1, \ldots, b_n.$, *such that:*

(i) I *and* O *are the (possibly empty) lists of input and output arguments.*
(ii) H_i *and* H_o *are the input and output heaps.*
(iii) E *is an exception flag indicating whether the execution of* m *ends normally or with an uncaught exception.*
(iv) If predicate m *is defined by multiple rules, the guards in each one contain mutually exclusive conditions. We denote by* m^k *the* $k-th$ *rule defining* m.
(v) $\bar{G}$ *is a sequence of constraints that act as execution guards on the rule.*
(vi) $b_1, \ldots, b_n$ *is a sequence of instructions, including arithmetic operations, calls to other predicates, and built-ins operations to handle the heap.*
(vii) T *is the trace term for* m *of the form* $m(k, \langle T_{c_i}, \ldots, T_{c_m} \rangle)$, *where* k *is the index of the rule and* $T_{c_i}, \ldots, T_{c_m}$ *are free logic variables representing the trace terms associated to the subsequence* $c_i, \ldots, c_m$ *of calls to other predicates in* $b_1, \ldots, b_n$.

Notice that the trace term T is a not a cardinal element in the translated program, but rather a supplementary argument with a central role in this paper.

2.2 Symbolic Execution

CLP-translated programs are symbolically executed using the standard CLP execution mechanism with special support for the use of dynamic memory [8].

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 a set of built-in predicates to handle dynamic memory. The symbolic execution of* m *is the CLP derivation tree, denoted as* $\mathcal{T}_m$, *with root* $m(I, O, H_i, H_o, E, T)$ *and initial constraint store* $\theta = \{\}$ *obtained using* P'.

2.3 Test Case Generation

The symbolic execution tree of programs containing loops or recursion is in general infinite. To guarantee termination of TCG it is therefore essential to impose a *termination criterion* that makes the symbolic execution tree finite:

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* $m(I, O, H_i, H_o, E, T)$ *w.r.t.* $\mathcal{C}$. *Let* B *be the set of the successful (terminating) branches of* $\mathcal{T}_m^{\mathcal{C}}$.
– *A test case for* m *w.r.t.* $\mathcal{C}$ *is a tuple* $\langle \theta, T \rangle$, *where* θ *and* T *are, resp., the constraint store and the trace term associated to one branch* $b \in B$.
– *TCG is the process of generating the set of test cases associated to all branches in* B.

```
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) {
μ       return -1;
    }
}
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)
κ         return a;
    else
        return -a;
}
```

```
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),
        check([A,B,G],[R],_,_,E,V).
check([A,B,G],[R],_,_,E,check(1,[T,V])) :-
        M #= A*B, abs([M],[S],_,_,E,T),
        div([S,G],[R],_,_,E,V).
check([A,B,G],[R],_,_,exc,check(2,[])).
div([A,B],[R],_,_,ok,div(1,[])) :-
        B #\= 0, R #= A/B.
div([A,0],[-1],_,_,exc_caught,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.
```

Fig. 1: Motivating Example: Java (left) and CLP-translated (right) programs.

Each *test case* produced by TCG represents a class of inputs that will follow the same execution path, and its trace is the sequence of rules applied along such path. In a subsequent step, it is possible to produce actual values from the obtained constraint stores (e.g., by using labeling mechanisms in standard *clpfd* domains) therefore obtaining concrete and executable test cases. However, this is not an issue of this paper and we will comply with the above abstract definition of *test case*.

Example 1. Fig. 1 shows a Java program consisting 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. The right side of the figure shows the equivalent CLP-translated program. Observe that each Java method corresponds to a set of CLP rules, e.g., method `lcm` is translated into predicates `lcm`, `cont`, `check` and `div`. 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. For readability, the actual CLP code has been simplified, e.g., input and output heap arguments are not shown, since they do not affect the computation. Our current implementation [2] supports full sequential Java.

Coverage Criteria. By Def. 3, so far we have been interested in covering *all* feasible paths of the program under test w.r.t. a termination criterion. Now, our goal is to improve on efficiency by taking into account a *selection criterion* as well. First, let us define a *coverage criterion* as a pair of two components $\langle TC, SC\rangle$. TC is a *termination criterion* that ensures finiteness of symbolic execution. This can be done either based on execution steps or on loop iterations. In this paper, we 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 steers TCG to determine which paths of the symbolic execution tree will be explored. In other words, SC decides which test cases the TCG must produce. In the rest of the paper we focus on 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. Intuitively, 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.

3 A Generic Framework for Guided TCG

The TCG framework as defined so far has been used in the context of coverage criteria only consisting of a termination criterion. In order to incorporate a selection criterion, one can 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.

In what follows, we develop a methodology where the TCG process is driven towards satisfying the selection criterion, stressing to avoid as much as possible the generation of irrelevant and/or redundant paths. The key idea that allows us to guide the TCG process is to use the trace terms of our CLP-translated program as input arguments. Let us observe also that we could either supply fully or partially instantiated traces, the latter ones represented by including free logic variables within the trace terms. This allows guiding, completely or partially, the symbolic execution towards specific paths.

Definition 4 (trace-guided TCG). *Given a method M, a termination criterion TC, and a (possibly partial) trace π, trace-guided TCG generates the set of test cases with traces, denoted* $\mathsf{tgTCG}(M, TC, \pi)$, *obtained for all successful branches in $\mathcal{T}_m^{TC}$ with root $m(Args_{in}, Args_{out}, H_{in}, H_{out}, E, \pi)$. We also define the* $\mathsf{firstOf\text{-}tgTCG}(M, TC, \pi)$ *to be the set corresponding to the leftmost successful branch in $\mathcal{T}_m^{TC}$.*

Observe that the TCG guided by one trace either generates: (a) exactly one test case if the trace is complete and corresponds to a feasible path, (b) none if it is unfeasible, or, (c) possibly several test cases if it is partial. In this case the traces of all test cases are instantiations of the partial trace.

Now, relying on trace-guided TCG and on the existence of a *trace generator* we define a generic scheme of *guided TCG.*

Definition 5 (guided TCG). *Given a method M; a coverage criterion $CC = \langle TC, SC \rangle$; and a trace generator $TraceGen$, that generates, on demand and one by one, (possibly partial) traces according to CC. Guided TCG is defined as the following algorithm:*

```
Input: M, ⟨TC, SC⟩, TraceGen
TestCases = {}
while TraceGen has more traces and TestCases does not satisfy SC
   Invoke TraceGen to generate a new trace in Trace
   TestCases ← TestCases ∪ firstOf-tgTCG(M, TC, Trace)
Output: TestCases
```

The intuition is as follows: The trace generator generates a trace satisfying SC and TC. 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 possible traces allowed by TC. If the trace generator is complete (see below), this means that SC cannot be satisfied within the limit imposed by TC.

Example 2. Let us consider the TCG for method `lcm` with program-points for points $\textcircled{\mu}$ and $\textcircled{\kappa}$ as selection criterion. Observe the correspondence of these program points in both the Java and CLP code of Fig. 1. Let us assume that the trace generator starts generating the following two traces:

```
t1 : lcm(1,[cont(1,[G,check(1,[A,div(2,[])])])])
t2 : 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 obtained test case is shown in Example 7.

On Soundness, Completeness and Effectiveness: Intuitively, a concrete instantiation of the guided TCG scheme is *sound* if all test cases it generates satisfy the coverage criterion, and *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. Those three features depend solely on the trace generator. We will refer to trace generators as being sound, complete or effective. The intuition is that a trace generator is sound if every trace it generates satisfies the coverage criterion, and complete if it produces an over-approximation of the set of traces satisfying it. Effectiveness is related to the number of unfeasible traces it generates, the larger the number, the less effective the trace generator.

4 Trace Generators for Structural Coverage Criteria

In this section we present a general approach for building sound, complete and effective trace generators for structural coverage criteria by means of program transformations. We then instantiate the approach for the all-local-paths and program-points coverage criteria and proposes Prolog implementations of the guided TCG scheme for both of them. Let us first define the notion of *trace-abstraction* of a program which will be the basis for defining our trace generators.

Definition 6 (trace-abstraction of a program). *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 3. Fig. 2 shows the trace-abstraction of our CLP-translated program of Fig. 1. Let us observe that it essentially corresponds to its control-flow graph.

```
lcm(lcm(1,[T])) :- cont(T).
lcm(lcm(2,[T])) :- cont(T).
cont(cont(1,[T,V])) :- gcd(T), check(V).
check(check(1,[T,V])) :- abs(T), div(V).
check(check(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. 2: Trace-abstraction.

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 sound and 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 develop two concrete, and more effective, schemes for the all-local-paths and program-points coverage criteria. In both cases, this is done by taking advantage of the notion of partial traces and the implicit information on the concrete coverage criteria.

4.1 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 7 (slicing for all-local-paths coverage criterion). *Given a trace-abstraction program P and an entry method M:*

1. *Remove from P all the rules that do not belong to method M.*
2. *For all remaining rules in P, remove from their bodies all the 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])) :- check(T).
check(check(1,[A,T])) :- div(T).
check(check(2,[])).
div(div(1,[])).
div(div(2,[])).
```

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

Fig. 3: Slicing of method `lcm` for all-local-paths criterion.

Example 4. Fig. 3 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 Def. 7; `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(M,Trace,TC) in Def. 4, failing if the set is empty, and instantiating on success `TestCase` and `Trace` (in case it was partial). The guided TCG scheme in Def. 5, 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 exit through line (7).

Example 5. 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,[])])]), check(1,[abs(1,[]),div(1,[])])])])`
`{A=B=0,Out=-1}`	`lcm(1,[cont(1,[gcd(1,[loop(1,[abs(1,[])])]), check(1,[abs(1,[]),div(2,[])])])])`
`{B>A}`	`lcm(2,[cont(1,[gcd(1,[loop(1,[abs(1,[])])]), check(1,[abs(1,[]),div(1,[])])])])`

This set of three 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 nine test cases.

4.2 An Instantiation for the program-points Coverage Criterion

Let us first consider a simplified version of the program-points criterion so that only one program point is allowed, denoted as program-point. Starting again from the trace-abstraction program, we apply a syntactic bottom-up program slicing algorithm to filter away all the paths in the program that do not visit the program point of interest.

Definition 8 (slicing for program-point coverage criterion). *Given a trace-abstraction program P, a program point of interest pp, and an entry method M, the sliced program P' is computed as follows:*

1. *Initialize P' to be the empty program, and a set of clauses L with the clause corresponding to pp.*
2. *For each c in L which is not the clause for M, add to L all clauses in P whose body has a call to the predicate of clause c, and iterate until the set L stabilizes.*
3. *Add to P' all clauses in L.*
4. *Remove all calls to rules which are not in P' from the bodies of the rules in P'.*

The obtained sliced program, together with the termination criterion, can be used as a trace generator for the program-point criterion. The generated traces can have free variables representing parts of the execution which are not related (syntactically) to the paths visiting the program point of interest.

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

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

Fig. 4: Slicing for program-point coverage criterion with pp=$\textcircled{\mu}$ from Fig. 1.

Example 6. Fig. 4 shows on the left the sliced trace-abstraction program (using Def. 8) for method `lcm` and program point $\textcircled{\mu}$ from Fig. 1, i.e. the `return` statement within the `catch` block. On the right of the same figure, the traces obtained from such slicing using loop-2 as termination criterion.

Consider again predicates `computeSlicedProgram/2`, `generateTrace/4` and `traceGuidedTCG/4` with the same meaning as in Section 4.1, but being the first two now based on Def. 8 and extended with the program-point argument `PP`. The guided TCG scheme in Def. 5, instantiated for the program-points criterion, can be implemented in Prolog as follows:

```
guidedTCG(M,[],TC) :- !.
guidedTCG(M,[PP|PPs],TC) :-
    computeSlicedProgram(M,PP),
    generateTrace(M,PP,TC,Trace),
    once(traceGuidedTCG(M,Trace,TC,TestCase)), !,
    assert(testCase(M,TestCase,Trace)),
    removeCoveredPoints(PPs,Trace,PPs'),
    guidedTCG(M,PPs',TC).
guidedTCG(M,[PP|_],TC) :- .
```

Intuitively, given the first remaining program point of interest `PP` (line `2`), a trace generator is computed and used to obtain a (possibly partial) trace that exercises `PP` (lines `3`–`4`). Then, if the call in line `5` fails, another trace for `PP` is requested on backtracking. When there are not more traces (i.e., line `4` fails) the process finishes through line `9` reporting that `PP` is not reachable within the imposed `TC`. If the call in line `5` succeeds, the generated test case is asserted with its corresponding trace (now fully instantiated in case it was partial), the remaining program points which are covered by `Trace` are removed obtaining `PPs'` (line `7`), and the process continues with `PPs'`. Note that a new sliced program is computed for each program point in `PPs'`. The process finishes through line `1` when all program points have been covered.

The above implementation is valid for the general case of program-points criteria with any finite set size. The trace generator, instead, has been deliberately defined for just one program point since this way the program slicing can be more aggressive, hence eluding the generation of unfeasible traces.

Example 7. The following test case is obtained for the program-points criterion for method `lcm` and program points ⓜ and ⓚ:

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

This particular case exemplifies specially well how guided TCG can reduce the number of produced test cases through adequate control of the selection criterion.

5 Experimental Evaluation

We have implemented the guided TCG schemes for both all-local-paths and program-points coverage criteria as proposed in Section 4, and integrated them within PET [2, 8], an automatic TCG tool for Java bytecode, which is available at `http://costa.ls.fi.upm.es/pet`. In this section we report on some experimental results which aim at demonstrating the applicability and effectiveness of guided TCG. The experiments have been performed using as benchmarks a selection of classes from the net.datastructures library [9], a well-known library of algorithms and data-structures for Java. In particular, we have used as "methods-under-test" the most relevant public methods of the classes *NodeSequence*, *SortedListPriorityQueue*, *BinarySearchTree* and *HeapPriorityQueue*, abbreviated respectively as *Seq*, *PQ*, *BST* and *HPQ*.

Table 1 aims at demonstrating the effectiveness of the guided TCG scheme for the all-local-paths coverage criterion. This is done by comparing it to standard way of implementing the all-local-paths coverage criterion, i.e., first generating all paths up to the termination criterion using standard TCG by symbolic execution, and then applying a filtering so that only the test cases that are necessary to meet the all-local-paths selection criterion are kept. Each row in the table corresponds to the TCG of one method using standard TCG vs. using guided TCG. For each method we provide: The number of reachable bytecode instructions (**BCs**) and the time of the translation of Java bytecode to CLP (**Tt**), including parsing and loading all reachable classes ; the time of the TCG process (**T**), the number of generated test cases before the filtering (**N**), and the code coverage achieved using standard TCG (**CC**); and the time of the TCG process (**Tg**), the number of generated test cases (**Ng**), the code coverage achieved (**CCg**), and the number of generated/unfeasible

Method Info			Standard TCG			Guided TCG			
Class.Name	**BCs**	**Tt**	**T**	**N**	**CC**	**Tg**	**Ng**	**CCg**	**GT/UT**
Seq.elemAt	98	45	18	24	100%	9	5	100%	6/1
Seq.insertAt	220	85	41	39	100%	14	6	100%	8/2
Seq.removeAt	187	76	35	36	100%	10	4	100%	5/1
Seq.replaceAt	163	66	35	36	100%	9	4	100%	5/1
PQ.insert	357	144	148	109	100%	10	3	100%	4/1
PQ.remove	158	69	8	12	100%	20	7	100%	15/8
BST.addAll	260	125	1491	379	100%	22765	18	100%	151/133
BST.find	228	113	76	62	100%	82	5	100%	7/2
BST.findAll	381	178	1639	330	100%	1266	4	100%	6/2
BST.insert	398	184	2050	970	100%	1979	9	100%	18/9
BST.remove	435	237	741	365	98%	3443	26	98%	204/178
HPQ.insert	322	132	215	43	100%	26	5	100%	6/1
HPQ.remove	394	174	1450	40	100%	100	8	100%	19/11

Table 1: Experimental results for the all-local-paths criterion

traces using guided TCG (**GT/UT**). All times are in milliseconds and are obtained as the arithmetic mean of five runs on an Intel(R) Core(TM) i5-2300 CPU at 2.8GHz with 8GB of RAM, running Linux Kernel 2.6.38. The code coverage measures, given a method, the percentage of its bytecode instructions which are exercised by the obtained test cases. This is a common measure in order to reason about the quality of the obtained test cases. As expected, the code coverage is the same in both approaches, and so is the number of obtained test cases. Otherwise, this would indicate a bug in the implementation.

Let us observe that the gains in time are significant for most benchmarks (column **T** vs. column **Tg**). There are however three notable exceptions for methods PQ.remove, BST.addAll and BST.remove, for which the guided TCG scheme behaves worse than the standard one, especially for BST.addAll. This happens in general when the control-flow of the method is complex, hence causing the trace generator to produce an important number of unfeasible traces (see last column). Interestingly, these cases could be statically detected using a simple syntactic analysis which looks at the control flow of the method. Therefore the system could automatically decide which methodology to apply. Moreover, Section 6 presents a trace-abstraction refinement that will help in improving guided TCG for programs whose control-flow is determine mainly by integer linear constraints. Other classes of programs, e.g. BST.addAll, require a more sophisticated analysis, since their control-flow are strongly determined by object types and dynamic dispatch information. This discussion and further refinement is left out of the scope of this paper.

Table 2 aims at demonstrating the effectiveness of the guided TCG scheme for the program-points coverage criterion. For this aim, we have implemented the support in the standard TCG scheme to check the program-points selection criterion dynamically while the test cases are generated, in such a way that the process terminates when all program points are covered. Note that, in the worst case this will require generating the whole symbolic execution tree, as the standard TCG does. Table 2 compares the effectiveness of this methodology against that of the guided TCG scheme. Again, each row in the table corresponds to the TCG of one method using standard TCG vs. using guided TCG, providing for both schemes the time of the TCG process (**T**

Method Info	Standard TCG			Guided TCG			
Class.Name	**T**	**N**	**CC**	**Tg**	**Ng**	**CCg**	**GT/UT**
Seq.elemAt	9	10	100%	6	3	100%	3/0
Seq.insertAt	39	36	100%	8	3	100%	3/0
Seq.removeAt	19	16	100%	8	3	100%	3/0
Seq.replaceAt	19	16	100%	8	3	100%	3/0
PQ.insert	149	109	100%	9	3	100%	3/0
PQ.remove	9	12	100%	5	3	100%	3/0
BST.addAll	1501	379	100%	284	2	100%	4/2
BST.find	77	62	100%	10	3	100%	3/0
BST.findAll	1634	330	100%	8	3	100%	3/0
BST.insert	2197	969	100%	35	3	100%	3/0
BST.remove	238	104	98%	61	3	98%	28/25
HPQ.insert	209	43	100%	24	3	100%	3/0
HPQ.remove	1385	38	100%	15	3	100%	3/0

Table 2: Experimental results for the program-points criterion

vs **Tg**), the number of generated test cases (**N** vs **Ng**), the code coverage achieved (**CC** vs **CCg**), and the number of generated/unfeasible traces using guided TCG (**GT/UT**). We have selected three program points for each method with the aim of covering as much code as possible. In all cases, such selection of program points allows obtaining the same code coverage as with the standard TCG even without the selection criterion (i.e. 100% coverage for all methods except 98% for BST.remove because of dead code). Let us observe that the gains in time are huge (column **T** vs. column **Tg**), ranging from one to two orders of magnitude, except for the simplest methods, for which the gain, still being significant, is not so notable. These results are witnessed by the low number of unfeasible traces that are obtained (column **GT/UT**), hence demonstrating the effectiveness of the trace-generator defined in Section 4.2.

Overall, we believe our experimental results support our initial claims about the potential interest of guiding symbolic execution and TCG by means of trace-abstractions. With the exception of some particular cases that deserve further study, our results demonstrate that we can achieve high code coverage without having to explore many unfeasible paths, with the additional advantage of discovering high quality (less in number and better selected) test cases.

6 Trace-Abstraction Refinement

As the above experimental results suggest, there are still cases where the trace-abstraction as defined in Def. 6 may still compromise the effectiveness of the guided TCG, because of the generation of too many unfeasible paths. This section discusses a complementary strategy to further optimize the framework. In particular, we propose a heuristics that aims to refine the trace-abstraction with information taken from the original program that will help reduce the number of unfeasible paths at symbolic execution. The goal is to reach a balanced level of refinement in between the original program (full refinement) and the trace-abstraction (empty refinement). Intuitively, the more information we include, the less unfeasible paths symbolic execution will explore, but the more costly it becomes.

The refinement algorithm consists of two steps: First, in a fixpoint analysis we approximate the instantiation mode of the variables in each predicate of the CLP-translated program. In other words, we infer which variables will be constrained or assigned a concrete value at symbolic execution time. In a second step, by program transformation, the trace-abstraction is enriched with clause arguments corresponding to the inferred variables, and with those goals in which they are involved.

6.1 Approximating instantiation modes

We develop a static analysis, similar to [5,6], to soundly approximate the instantiation mode of the input argument variables in the program at symbolic execution time. The analysis is implemented as a fixpoint computation over the simple abstract domain $\{static, dynamic\}$. Namely, *dynamic* means that nothing was inferred about a variable and it will therefore remain a free unconstrained variable during symbolic execution; and *static* means that the variable will unify with a concrete value or will be constrained during symbolic execution. The analysis's result is a set of assertions in the form $\langle P, \mathcal{V} \rangle$ where P is a predicate name and $\mathcal{V}$ is the set of variables in P, each associated with an abstract value from the domain.

This analysis receives as input a CLP-translated program and a set of initial entries (predicate names). An event queue $\mathcal{Q}$ is initialized with this set of initial entries. The algorithm starts to process the events of $\mathcal{Q}$ until no more events are scheduled. In each iteration, an event p is removed from $\mathcal{Q}$ and processed as follows: Retrieve previously stored information $\psi \equiv \langle p, \mathcal{V} \rangle$ if any exists; else set $\psi \equiv \langle p, \emptyset \rangle$. For each rule r defining p, a new $\mathcal{V}_r$ is obtained by evaluating the body of r. The joint operation on the underlying abstract domain is performed to obtain $\mathcal{V}' \Leftarrow joint(\mathcal{V}, \mathcal{V}_r)$. If $\mathcal{V} \not\equiv \mathcal{V}'$ then set $\mathcal{V} \Leftarrow \mathcal{V}'$ and reschedule every predicate that calls p; else, if $\psi' \equiv \psi$ there is no need to recompute the calling predicates and the algorithm continues. That will ensure backward propagation of approximated instantiation modes. To propagate forward, the evaluation of r will schedule one event per call within its body. The process continues until a fixpoint is reached.

6.2 Constructing the trace-abstraction refinement

This is a syntactic program transformation step of the refinement. It takes as input the original CLP-program and the instantiation information inferred in the first step and outputs a trace-abstraction refinement program. For each rule r of a predicate p in the program, the algorithm retrieves $\langle p, \mathcal{V} \rangle$. We denote $\mathcal{V}_s$ the projection of all variables in $\mathcal{V}$ whose inferred abstract value is *static*. The algorithm adds to the trace-abstraction refinement a new rule r' whose list of arguments is $\mathcal{V}_s$. The body of r' is constructed by traversing the body $b_1, \ldots, b_n$ of r and including 1) all guards and arithmetic operations b_i involving $\mathcal{V}_s$, and 2) all calls to other predicates, with the corresponding projection of $\mathcal{V}_s$ over the arguments of the calls.

Example 8. Consider the Java example of Fig. 5 (left side). Function `power` implements a exponentiation algorithm for positive integer exponents. Its CLP counterpart is shown at the right of the figure. The instantiation modes inferred by the first stage of our algorithm is presented at the right-bottom part of the figure. One can observe that variable `B` (the base of the exponentiation) remains *dynamic* all along the program, because it is never assigned any concrete value nor constrained by any guard. On the other hand, variable `E`'s final abstract value is *static*, since it

```
void arraypower(int a[],int e) {
    int i=0;
    int n=a.length;
    for (i=0; i<n; i++)
        if (i%2==0)
            a[i]=power(a[i],e);
}
int power(int b, int e) {
    if (e >= 0) {
        int pow = 1;
        while (i <= e) {
            pow *= b;
            i++;
        }
        return pow;
    } else return -1;
}
```

```
power([B,E],[R],_,_,F,power(1,[T])) :-
   if([B,E],[R],_,_,F,T).
if([B,E],[-1],_,_,F,if(1,[])) :-
   E #< 0.
if([B,E],[R],_,_,F,if(2,[T])) :-
   E #>= 0), loop([B,E,1,1],[R],_,_,F,T).
loop([B,E,I,P],[P],_,_,ok,loop(1,[])) :-
   I #> E.
loop([B,E,I,P],[R],_,_,F,loop(2,[T])) :-
   I #=< E, Pp #= P*B, Ip #= I+1,
   loop([B,E,Ip,Pp],[R],_,_,F,T).
```

Inferred instantiation modes:
⟨`power`, {`B`= *dynamic*,`E`= *static*}⟩
⟨`if`, {`B`= *dynamic*,`E`= *static*}⟩
⟨`loop`, {`B`= *dynamic*,`E`= *static*,
`I`= *static*,`P`= *dynamic*}⟩

Fig. 5: Trace-abstraction refinement.

is constrained by `0` and the also *static* variable `I` in rules `if` and `loop`. The following is the refined trace-abstraction that our algorithm constructs:

```
power([E],power(1,[T])) :- if([E],T).
if([E],if(1,[])) :- E #< 0.
if([E],if(2,[T])) :- E #>= 0, loop([E,1],T).
loop([E,I],loop(1,[])) :- I #> E.
loop([E,I],loop(2,[T])) :- I #=< E, Ip #= I+1, loop([E,Ip],T).
```

To illustrate how the trace-abstraction refinement can improve on effectiveness of the guided TCG, let us observe method `arraypower`. It iterates over the elements of an input array `a` and calls function `power` to update all even positions of the array by raising their values to the power of the integer input argument `e`. We report on the following performance results for this example and a coverage criterion ⟨loop-2, {}⟩:

– Standard non-guided TCG (i.e., full refinement) generates 11 test cases.
– Trace-abstraction guided TCG with the empty refinement generates 497 possibly (un)feasible traces.
– Trace-abstraction guided TCG with our trace-abstraction refinement reduces the number of possibly (un)feasible traces to 161.

These preliminary yet promising results, suggest the potential integrability of the trace-abstraction refinement algorithm presented in this section within the general guided TCG framework developed in this paper. The refinement is complementary to the slicings schemes presented in Section 4 without any modification. Unfortunately, the slicings could produce a loss of important information added by the refinement. This could be however improved by means of simple syntactic analyses on the sliced parts of the program. A deeper study of these issues remains as future work.

7 Related Work and Conclusions

Previous work also uses abstractions to guide symbolic execution and TCG by several means and for different purposes. Fundamentally, abstraction aims to reduce large data domains of a program to smaller domains [11]. One of the most relevant to ours is [3], where predicate abstraction, model checking and SAT-solving are combined to produce abstractions and generate test cases for C programs, with good code coverage, but depending highly on an initial set of predicates to avoid infeasible program paths. Rugta *et al.* [13] also proposes to use an abstraction of the program in order to guide symbolic execution and prune the execution tree as a way to scale up. Their abstraction is an under-approximation which tries to reduce the number of test cases that are generated in the context of concurrent programming, where the state explosion is in general problematic.

The main contribution of this paper is the development of a methodology for Guided TCG that allows to guide the process of test generation towards achieving more selective and interesting structural coverage. Implicit is the improvement in the scalability of TCG by guiding symbolic execution by means of trace-abstractions, since we gain more control over the symbolic execution state space to be explored. Moreover, whereas the main goal of our CLP-based TCG framework has been the exhaustive testing of programs, our new Guided TCG framework unveil new potential applications areas. Namely, the all-local-paths and program-points Guided TCG schemes we have presented in this paper, enable us to explore on the automation of other interesting software testing practices, such as selective and unit testing, goal-oriented testing and bug detection.

The effectiveness and applicability of Guided TCG is substantiated by an implementation within the PET system (`http://costa.ls.fi.upm.es/pet`), and encouraging experimental results. Nevertheless, our current and future work involves a more thorough experimental evaluation of the framework and the exploration of the new application areas in software testing. In a different line, a particularly challenging goal has been triggered which consists in developing static analysis techniques to achieve optimal refinement levels of the trace-abstraction programs. Last but not least, we plan to further study the generalization and integration of other interesting coverage criteria to our Guided TCG framework.

Acknowledgments. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN2008-05624, TIN2012-38137 and PRI-AIBDE-2011-0900 projects, by UCM-BSCH-GR35/10-A-910502 grant and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

References

1. E. Albert, M. Gómez-Zamalloa, and G. Puebla. Test Data Generation of Bytecode by CLP Partial Evaluation. In *Proc. of LOPSTR'08*, volume 5438 of *LNCS*. Springer-Verlag, 2009.
2. E. Albert, I. Cabañas, A. Flores-Montoya, M. Gómez-Zamalloa, and S. Gutiérrez. jPET: an Automatic Test-Case Generator for Java. In *Proc. of WCRE'11*. IEEE Computer Society, 2011.

3. T. Ball. Abstraction-guided test generation: A case study. Technical Report MSR-TR-2003-86, Microsoft Research, 2003.
4. C. Cadar, P. Godefroid, S. Khurshid, C. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proc. of ICSE'11*. ACM, 2011.
5. S.-J. Craig, J. P. Gallagher, M. Leuschel, and K. S. Henriksen. Fully Automatic Binding-Time Analysis for Prolog. In *Proc. of LOPSTR'04*, volume 3573 of *LNCS*. Springer, 2004.
6. S. K. Debray. Static inference of modes and data dependencies in logic programs. *ACM Trans. Program. Lang. Syst.*, 11(3):418–450, July 1989.
7. 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.
8. 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), 2010.
9. M. Goodrich, R. Tamassia, and E. Zamore. The net.datastructures package. http://net3.datastructures.net.
10. J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
11. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Proc. of TACAS'01*, volume 2031 of *LNCS*. Springer-Verlag, 2001.
12. 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.
13. N. Rungta, E.G. Mercer, and W. Visser. Efficient testing of concurrent programs with abstraction-guided symbolic execution. In *Proc. of SPIN'09*, volume 5578 of *LNCS*. Springer, 2009.