Resource-driven CLP-based Test Case Generation

Elvira Albert[1], Miguel Gómez-Zamalloa[1], José Miguel Rojas[2]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. Test Data Generation (TDG) aims at automatically obtaining *test inputs* which can then be used by a software testing tool to validate the functional behaviour of the program. In this paper, we propose *resource-aware* TDG, whose purpose is to generate test cases (from which the test inputs are obtained) with associated *resource consumptions*. The framework is parametric w.r.t. the notion of resource (it can measure memory, steps, etc.) and allows using software testing to detect bugs related to non-functional aspects of the program. As a further step, we introduce *resource-driven* TDG whose purpose is to guide the TDG process by taking resource consumption into account. Interestingly, given a *resource policy*, TDG is guided to generate test cases that adhere to the policy and avoid the generation of test cases which violate it.

1 Introduction

Test data generation (TDG) is the process of automatically generating *test inputs* for interesting test *coverage criteria*. Examples of coverage criteria are: *statement coverage* which requires that each line of the code is executed; *loop-k* which limits to a threshold k the number of times we iterate on loops. The standard approach to generate test cases statically is to perform a *symbolic execution* of the program [7,8,12,14,17,18,20], where the contents of variables are expressions rather than concrete values. Symbolic execution produces a system of constraints over the input variables consisting of the conditions to execute the different paths. The conjunction of these constraints represents the equivalence class of inputs that would take this path. In what follows, we use the term *test cases* to refer to such constraints. Concrete instantiations of the test cases that satisfy the constraints are generated to obtain test inputs for the program. Testing tools can later test the *functionality* of an application by executing such test inputs and checking that the output is as expected. The CLP-based approach to TDG of *imperative*[3] programs [4,11] consists of two phases: (1) first, an imperative program is translated into an equivalent CLP program and (2) symbolic execution is performed on the CLP program by relying on the standard CLP's

[3] The application of this approach to TDG of logic programs must consider failure [20] and, to functional programs, should consider laziness, higher-order, etc. [9].

evaluation mechanisms (extended with a special treatment for heap-allocated data [11]) which provide symbolic execution for free.

Non-functional aspects of an application, like its resource consumption, are often more difficult to understand than functional properties. Profiling tools execute a program for concrete inputs to assess the associated resource consumption, a non-functional aspect of the program. Profilers can be parametric w.r.t. the notion of resource which often includes cost models like time, number of instructions, memory consumed, number of invocations to methods, etc. Usually, the purpose of profiling is to find out which parts of a device or software contribute most to its poor performance and find bugs related to the resource consumption.

In this paper, we propose *resource-aware TDG* which strives to build performance into test cases by additionally generating their resource consumption, thus enriching standard TDG with non-functional properties. The main idea is that, during the TDG process, we keep track of the exercised instructions to obtain the test case. Then, in a simple post-process we map each instruction into a corresponding cost, we obtain for each class of inputs a detailed information of its resource consumption (including the resources above). Our approach is not reproducible by first applying TDG, then instantiating the test cases to obtain concrete inputs and, finally, performing profiling on the concrete data. This is because, for some cost criteria, resource-aware TDG is able to generate symbolic (i.e., non-constant) costs. E.g., when measuring memory usage, the amount of memory might depend on an input parameter (e.g., the length of an array to be created is an input argument). The resource consumption of the test case will be a symbolic expression that profilers cannot compute.

A well-known problem of TDG is that it produces a large number of test cases even for medium size programs. This introduces scalability problems as well as complicates human reasoning on them. An interesting aspect of resource-aware TDG is that resources can be taken into account in order to filter out test cases which do not consume more (or less) than a given amount of resources, i.e, one can consider a *resource policy*. This leads to the idea of resource-*driven* TDG, i.e., a new heuristics which aims at guiding the TDG process to generate test cases that adhere to the resource policy. The potential interest is that we can prune the symbolic execution tree and produce, more efficiently, test cases for inputs which otherwise would be very expensive (and even impossible) to obtain.

Our approach to resource-driven CLP-based TDG consists of two phases. First, in a pre-process, we obtain (an over-approximation of) the set of *traces* in the program which lead to test cases that adhere to the resource policy. We sketch several ways of automatically inferring such traces, starting from the simplest one that relies on the call graph of the program to more sophisticated ones that enrich the abstraction to reduce the number of unfeasible paths. An advantage of formalizing our approach in a CLP-based setting is that traces can be partially defined and the TDG engine then completes them. Second, executing standard CLP-based TDG with a (partially) instantiated trace generates a test case that satisfies the resource policy (or it fails if the trace is unfeasible). An interesting aspect is that, if the trace is fully instantiated, TDG becomes deterministic and

solutions can be found very efficiently. Also, since there is no need to backtrack, test cases for the different traces can be computed in parallel.

2 CLP-based Test Case Generation

This section summarizes the CLP-based approach to TDG of [11] and extends it to incorporate *traces* in the CLP programs that will be instrumental later to define the resource-aware framework. CLP-based TDG consists of two main steps: (1) imperative programs are translated into an extended form of equivalent CLP-programs which incorporate built-in operations to handle dynamic data, and, (2) symbolic execution is performed on the CLP-translated programs by relying on the standard evaluation mechanisms of CLP with special operations to treat such built-ins. The next two sections overview these steps.

2.1 CLP-Translation with Traces

The translation of imperative (object-oriented) programs into equivalent CLP program has been subject of previous work (see, e.g., [1,10]). Therefore, we will not go into details of how the transformation is done, but rather simply recap the features of the translated programs in the next definition.

Definition 1 (CLP-translated program). *Given a method m with input arguments $\bar{x}$ and output arguments $\bar{y}$. Its CLP-translation consists of a set of predicates $m, m_1, \ldots, m_n$ such that each of them is defined by a set of rules of the form "$m(\mathtt{I},\mathtt{O},\mathtt{H_{in}},\mathtt{H_{out}})$:- $g, b_1, \ldots, b_n$." where:*

1. *m is the entry predicate (named as the method) and its arguments* `I` *and* `O` *are lists of variables that correspond to $\bar{x}$ and $\bar{y}$.*
2. *For the remaining predicates $m_1, \ldots, m_n$,* `I` *and* `O` *are, resp., the list of input and output arguments of this predicate.*
3. *$\mathtt{H_{in}}$ and $\mathtt{H_{out}}$ are, resp., the input and output heaps to each predicate.*
4. *If a predicate m_i is defined by multiple rules, the guards in each rule contain mutually exclusive conditions. We denote by m_i^k the $k-th$ rule defining m_i.*
5. *$g, b_1, \ldots, b_n$ are CLP-representations of equivalent instructions in the imperative language (as usual, a SSA transformation is performed on the variables), method invocations are replaced by calls to corresponding predicates, and operations that handle data in the heap are translated into built-in predicates (e.g.,* new_object(H,Class,Ref,H'), get_field(H,Ref,Fld,Val)*, etc.).*

Given a rule m_i^k, we denote by $instr(m_i^k)$ the sequence of instructions in the original program that have been translated into rule m_i^k.

As the imperative program is deterministic, the CLP translation is deterministic as well (point 4 in Def. 1). Observe that the global memory (or heap) is explicitly represented in the CLP program by means of logic variables. When a rule is invoked, the input heap $\mathtt{H_{in}}$ is received and, after executing its body,

the heap might be modified, resulting in $H_{\texttt{out}}$. The operations that modify the heap will be shown in the example. 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 TDG 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.

In the next definition, we add a *trace term* as an additional argument to each rule of Def. 1 to keep track of the sequence of rules that are executed.

Definition 2 (CLP-translated program with trace). *Given the rule of Def. 1, its CLP-translation with trace is: "*$m(\texttt{I},\texttt{O},\texttt{H}_{\texttt{in}},\texttt{H}_{\texttt{out}},\texttt{T})\texttt{:-}\ g, b'_1, \ldots, b'_n.$*", where* T *is the trace term for* m *of the form* $\mathsf{m}(\mathsf{k},\mathsf{P},\langle T_{c_i}, \ldots, T_{c_m}\rangle)$*. Here,* P *is the 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* $b_j \equiv p(I_p, O_p, H_{\texttt{in}_\texttt{p}}, H_{\texttt{out}_\texttt{p}})$*, then* $b'_j \equiv p(I_p, O_p, H_{\texttt{in}_\texttt{p}}, H_{\texttt{out}_\texttt{p}}, T_{c_j})$*; otherwise* $b'_j \equiv b_j$*.*

Example 1. Our example in Fig. 1 shows class Vector, that contains a reference to an array of integers elems and two integer fields to keep track of its size and capacity (size and cap). The initial capacity of the array is set by the constructor (method init). The interesting aspect of class Vector is that, when adding an element using method add, if the size has already reached the maximum capacity determined by field cap the size of the array is duplicated (by method realloc) before actually adding the new element. Fig. 2 shows the (simplified and pretty-printed) CLP translation obtained by the PET system [4] from the bytecode associated to class Vector. For brevity, we have omitted the predicates that model the exceptional behavior. Observe that each method is transformed into a set of predicates, some of them defined by several (mutually exclusive) guarded rules. In particular, method add is transformed into predicates add, if and addc. Variable names in the decompiled program correspond to the original names in the Java source. The operations that handle the heap remain as built-in predicates. Heap references are written as terms of the form r(Ref). Function *instr* in Def. 1 keeps the mapping between rules and bytecode instructions. For instance, $instr(\mathsf{init}^1)$=⟨`iload icap, ifgt, aload this, iload icap, newarray int, putfield elems, aload this, aload icap, putfield cap, aload this, iconst 0, putfield size, return`⟩ is the sequence of bytecode instructions that have been translated into rule init. A trace term is made up by the predicate name and number, the set of input arguments on which the cost depends (e.g., rule realloc and its trace parameter NCap) and it recursively includes the trace terms for the predicates it calls.

```
class Vector {
    int[] elems; int size, cap;
    Vector(int iCap) throws Exception{
        if (iCap > 0){
            elems = new int[iCap];
            cap = iCap; size = 0;
        } else
            throw new Exception();
    }
    void add(int x){
        if (size >= cap)
            realloc();
        elems[size++] = x;
    }
    void realloc(){
        int nCap = cap*2;
        int[] nElems = new int[nCap];
        for (int i=0; i<cap; i++) {
            nElems[i] = elems[i];
        }
        cap = nCap; elems = nElems;
    }
}
```

Fig. 1: Java source code example (1).

2.2 Symbolic Execution

When the imperative language does not use dynamic memory, CLP-translated programs can be executed by using the standard CLP's execution mechanism with all arguments being free variables. However, in order to generate arbitrary heap-allocated data structures, it is required to define heap-related operations which build the heap associated with a given path by using only the constraints induced by the visited code. We rely in the CLP-implementation presented in [11], where operations to create, read and modify heap-allocated data structures are presented in detail. Briefly, at symbolic execution-time, 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 one represented as a pair of the form *(signature, content)*. An array contains its type, its length and its list of elements. Note that our CLP translated programs manipulate the heap as a black-box through its associated operations. For instance, a new object is created through a call to predicate $\mathsf{new_object}(\mathsf{H}_{\mathsf{In}},\mathsf{Class},\mathsf{Ref},\mathsf{H}_{\mathsf{Out}})$, 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 $\mathsf{H}_{\mathsf{Out}}$ is the new heap after allocating the object. Read-only operations do

```
add([r(This),X],[],H,H1,add(1,[],[T])) :- get_field(H,This,size,Size),
   get_field(H,This,cap,Cap), if([Size,Cap,r(This),X],[],H,H1,T).
if1([Size,Cap,r(This),X],[],H,H1,if(1,[],[T])) :- Size #< Cap,
   addc([r(This),X],[],H,H1,T).
if2([Size,Cap,r(This),X],[],H,H2,if(2,[],[T1,T2])) :- Size #>= Cap,
   realloc([r(This)],[],H,H1,T1), addc([r(This),X],[],H1,H2,T2).
addc([r(This),X],[],H,H2,addc(1,[],[])) :- get_field(H,This,elems,r(Es)),
   get_field(H,This,size,Size), set_array(H,Es,Size,X,H1),
   NSize #= Size+1, set_field(H1,This,size,NSize,H2).
---
realloc([r(This)],[],H,H2,realloc(1,[NCap],[T])) :-
   get_field(H,This,cap,Cap), NCap #= Cap*2, new_array(H,int,NCap,NEs,H1),
   loop([r(This),NCap,r(NEs),0],[],H1,H2,T).
loop([r(This),NCap,r(NEs),I],[],H,H1,loop(1,[],[T])) :-
   get_field(H,This,cap,Cap), cond([Cap,I,r(This),NCap,r(NEs)],[],H,H1,T).
cond1([Cap,I,r(This),NCap,r(NEs)],[],H,H2,cond(1,[],[])) :- I #>= Cap,
   set_field(H,This,cap,NCap,H1), set_field(H1,This,elems,r(NEs),H2).
cond2([Cap,I,r(This),NCap,r(NEs)],[],H,H2,cond(2,[],[T])) :- I #< Cap,
   get_field(H,This,elems,r(Es)), get_array(H,Es,I,E), set_array(H,NEs,I,E,H1),
   NI #= I+1, loop([r(This),NCap,r(NEs),NI],[],H1,H2,T).
---
init1([r(This),ICap],[],H,H4,init(1,[ICap],[])) :- ICap #> 0,
   new_array(H,int,ICap,E,H1),set_field(H1,This,elems,r(E),H2),
   set_field(H2,This,cap,ICap,H3),set_field(H3,This,size,0,H4).
init2([r(This),ICap],[],H,H1,init(2,[ICap],[])) :- ICap #=< 0,
   new_object(H,'Exception',E,H1).
```

Fig. 2: CLP translation.

not produce any output heap (e.g. get_field(H,Ref,Field,Value)). The remaining operations are implemented likewise.

It is well-known that the execution tree to be traversed in symbolic execution is in general infinite. This is because iterative constructs such as loops and recursion whose number of iterations depends on the input values usually induce an infinite number of execution paths when executed with unknown input values. It is therefore essential to establish a *termination criterion*. In the context of TDG, termination is usually ensured by the *coverage criterion* which guarantees that the set of paths generated produces test cases which meet certain degree of code coverage and the process terminates. In what follows, we denote by $\mathcal{T}_m^C$ the finite symbolic execution tree of method m obtained using coverage criterion C.

Definition 3 (test case with trace and TDG). *Given a method m, a coverage criterion C and a successful branch b in $\mathcal{T}_m^C$ with root $m(Args_{in},Args_{out},H_{in},H_{out},T)$, a* test case with trace *for m w.r.t. C is a 6-tuple of the form: $\langle\sigma(Args_{in}),\sigma(Args_{out}),\sigma(H_{in}),\sigma(H_{out}),\sigma(T),\theta\rangle$, where σ and θ are the set of bindings and constraint store, resp., associated to b. TDG generates the set of test cases with traces obtained for all successful branches in $\mathcal{T}_m^C$.*

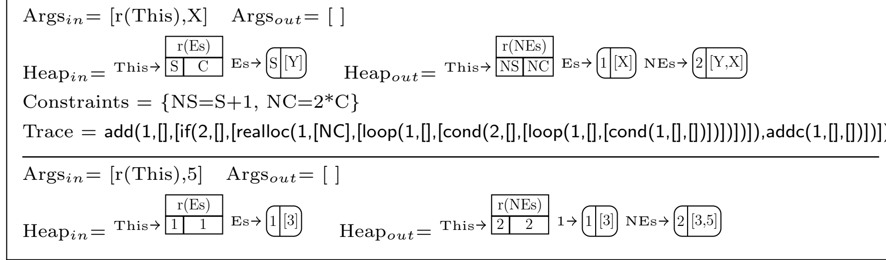


Fig. 3: Example of test case (up) and test input (down) for add with loop-1

The root of the execution tree is a call to method m with its associated arguments. Calls to methods are inlined in this scheme (one could use compositional TDG [5] as well). Each test case represents a class of inputs that will follow the same execution path, and its trace the sequence of rules applied along such path.

Example 2. Let us consider loop-1 as coverage criterion. In our example, loop-1 forces the array in the input vector to be at most of length 1. Note that we include the reference to the This object as an explicit input argument in the CLP translated program. The symbolic execution tree of `add(Args_in,Args_out,H_in,H_out,T)` will contain the following two successful derivations (ignoring exceptions):

1. If the size of the Vector object is less than its capacity, then the argument X is directly inserted in elems.
2. If the size of the Vector object is greater than or equal to its capacity, then method realloc is invoked before inserting X.

Fig. 3 shows in detail the second test case. Heaps are graphically represented by using rounded boxes for arrays (the array length appears to the left and the array itself to the right) and square boxes for Vector objects (field elems appears at the top, fields size and cap to the left and right bottom of the square, resp.). The trace-term `T` contains the rules that were executed along the derivation. At the bottom of the figure, an (executable) instantiation of this test case is shown.

3 Resource-Aware Test Case Generation

In this section, we present the extension of the TDG framework of Sec. 2 to build resource consumption into the test cases. First, in Sec. 3.1 we describe the cost models that we will consider in the present work. Then, Sec. 3.2 presents our approach to resource-aware TDG.

3.1 Cost Models

A cost model defines how much the execution of an instruction costs. Hence, the resource consumption of a test case can be measured by applying the selected cost model to each of the instructions exercised to obtain it.

Number of Instructions. The most traditional model, denoted $\mathcal{M}_{ins}$, is used to estimate the number of instructions executed. In our examples, since the input to our system is the bytecode of the Java program, we count the number of bytecode instructions. All instructions are assigned cost 1.

Memory Consumption. Memory consumption can be estimated by counting the actual size of all objects and arrays created along an execution [3].

$$\mathcal{M}_{mem}(b) = \begin{cases} size(Class) & if\ b \equiv \mathsf{new}\ Class \\ S_{ref} * Length & if\ b \equiv \mathsf{anewarray}\ Class\ Length \\ S_{prim} * Length & if\ b \equiv \mathsf{newarray}\ PrimType\ Length \\ 0 & otherwise \end{cases}$$

We denote by S_{prim} and S_{ref}, resp., the size of primitive types and references. In the examples, by assuming a standard JVM implementation, we set both values to 4 bytes. The size of a class is the sum of the sizes of the fields it defines. Note that, if one wants to consider garbage collection when assessing memory consumption, then the behaviour of the garbage collection should be simulated during the generation of test cases. In this paper, we assume that no garbage collection is performed.

Number of calls. This cost model, $\mathcal{M}_{call}$, counts the number of invocations to methods. It can be specialized to $\mathcal{M}^{m}_{call}$ to count calls to a specific method m which, for instance, can be one that triggers a billable event (e.g. send SMS).

Example 3. The application of the cost models $\mathcal{M}_{ins}$, $\mathcal{M}_{mem}$ and $\mathcal{M}_{call}$ to the sequence of instructions in rule init (i.e., $instr(\mathsf{init})$ of Ex. 1) results in, resp., 14 bytecode instructions, $4 * ICap$ bytes and 0 calls.

3.2 Resource-Aware TDG

Given the test cases with trace obtained in Def. 3, the associated cost can be obtained as a simple post-process in which we apply the selected cost models to all instructions associated to the rules in its trace.

Definition 4 (test case with cost). *Consider a test case with trace* $Test \equiv \langle Args_{in}, Args_{out}, H_{in}, H_{out}, \mathrm{Trace}, \theta\rangle$*, obtained in Def. 3 for method* m *w.r.t.* C*. Given a cost model* $\mathcal{M}$*, the cost of* $Test$ *w.r.t.* $\mathcal{M}$*, is defined as:*

$$C(Test, \mathcal{M}) = \mathtt{cost}(\mathrm{Trace}, \mathcal{M})$$

where function `cost` *is recursively defined as:*

$$\mathtt{cost}(\mathtt{m}(\mathtt{k},\mathtt{P},\mathtt{L}),\mathcal{M}) = \begin{cases} \sum_{\forall i \in instr(m^k)} \mathcal{M}(i) & if\ \mathtt{L} = [\ ] \\ \sum_{\forall i \in instr(m^k)} \mathcal{M}(i) + \sum_{\forall l \in \mathtt{L}} \mathtt{cost}(l, \mathcal{M}) & otherwise \end{cases}$$

For the cost models in Sec. 3.1, we define the test case with cost *as a tuple of the form* $\langle Test, C(Test, \mathcal{M}_{ins}), C(Test, \mathcal{M}_{mem}), C(Test, \mathcal{M}_{call})\rangle$.

```java
static Vector[] multiples(int[] ns, int div, int icap){
    Vector v = new Vector(icap);
    for (int i=0; i<ns.length; i++)
        if (ns[i]%div == 0)
            v.add(ns[i]);
    return r;
}
```

Fig. 4: Java source code example (2).

$\text{Args}_{in} = [\text{r(Ns)},\text{Div},1]$ $\text{Args}_{out} = \text{r}(0)$

$\text{Heap}_{in} =$ Ns→ (4 | [E_1,E_2,E_3,E_4]) $\text{Heap}_{out} =$ Ns→ (4 | [E_1,E_2,E_3,E_4]) 1→ [r(4) | 4 | 4] 2→ (1 | [E_1]) 3→ (2 | [E_1,E_2]) 4→ (4 | [E_1,E_2,E_3,E_4])

Constraints $= \{\text{Div}\neq 0,\ E_1 \bmod \text{Div}=0,\ E_2 \bmod \text{Div}=0,\ E_3 \bmod \text{Div}=0,\ E_4 \bmod \text{Div}=0\}$

$\mathcal{M}_{mem}=2{*}S_{PrimType}+S_{Ref}+S_{PrimType}{*}1+S_{PrimType}{*}2+S_{PrimType}{*}4=40$ bytes

$\mathcal{M}_{ins} = 270$ bytecode instructions $\mathcal{M}^{\mathsf{realloc}}_{call} = 2$

$\text{Args}_{in} = [\text{r(Ns)},\text{Div},\text{ICap}]$ $\text{Args}_{out} = \text{r}(0)$

$\text{Heap}_{in} =$ Ns→ (2 | [E_1,E_2]) $\text{Heap}_{out} =$ Ns→ (4 | [E_1,E_2]) 1→ [r(2) | 4 | ICap] 2→ (ICap | [$E_1,E_2,\ldots$])

Constraints $= \{\text{ICap}\geq 4,\ \text{Div}\neq 0,\ E_1 \bmod \text{Div}=0,\ E_2 \bmod \text{Div}\neq 0\}$

Trace = multiples(1,[],[init(1,[ICap],[]),mloop(1,[],[mcond(2,[],[mif(2,[],[add(1,[],[if(1,[],[addc(1,[],[])])])]),
mloop(1,[],[mcond(2,[],[mif(1,[],[mloop(1,[],[mcond(1,[],[])])])])])])])])])

$\mathcal{M}_{mem} = 2{*}S_{PrimType}+S_{Ref}+S_{PrimType}{*}\text{ICap} = 12+4{*}\text{ICap}$

$\mathcal{M}_{ins} = 86$ bytecode instructions $\mathcal{M}^{\mathsf{realloc}}_{call} = 0$

$\text{Args}_{in}=[\text{r(Ns)},\text{Div},3]$ $\text{Args}_{out}=\text{r}(0)$

$\text{Heap}_{in} =$ Ns→ (4 | [E_1,E_2,E_3,E_4]) $\text{Heap}_{out} =$ Ns→ (4 | [E_1,E_2,E_3,E_4]) 0→ [r(2) | 4 | 6] 1→ (3 | [E_1,E_2,E_3]) 2→ (6 | [$E_1,E_2,E_3,E_4,0,0$])

Constraints $= \{\text{Div}\neq 0,\ E_1 \bmod \text{Div}=0,\ E_2 \bmod \text{Div}=0,\ E_3 \bmod \text{Div}=0,\ E_4 \bmod \text{Div}=0\}$

$\mathcal{M}_{mem} = (2{*}S_{PrimType}+S_{Ref}+S_{PrimType}{*}3+S_{PrimType}{*}6) = 48$ bytes

$\mathcal{M}_{ins} = 247$ bytecode instructions $\mathcal{M}^{\mathsf{realloc}}_{call} = 1$

Fig. 5: Selected test cases with cost for method multiples with loop-4

This could also be done by profiling the resource consumption of the execution of the test-case. However, observe that our approach goes beyond the capabilities of TDG + profiling, as it can also obtain *symbolic* (non-constant) resource usage estimations while profilers cannot. Besides, it saves us from the non trivial implementation effort of developing a profiler for the language.

Example 4. We use a slightly more complex example from now on. Fig. 4 shows method multiples, which receives an input array of integers ns and outputs an object of type Vector, created with initial capacity icap, containing all the elements of ns that are multiples of the second input argument div. Let us consider the TDG of the CLP translation of method multiples with loop-4 as coverage criterion. We get 54 test cases, which correspond to all possible executions for input

arrays of length not greater than 4, i.e., at most 4 iterations of the for loop. Fig. 5 shows three test cases. The upper one corresponds to the test case that executes the highest number of instructions, in which method realloc is executed 2 times (worst case for $\mathcal{M}_{call}^{\mathsf{realloc}}$ as well). The one in the middle corresponds to one of the paths with the highest parametric memory consumption (for brevity, only the trace for this case is shown), and the one at the bottom corresponds to that with the highest constant memory consumption. In the middle one, the input array Ns is of length 2, both elements in the array are multiples of Div, and the initial capacity is constrained by ICap ≥ 4. With such input configuration, the array is fully traversed and its two elements are inserted in the resulting Vector object. By applying the cost model $\mathcal{M}_{mem}$ to the trace in the figure, we obtain a symbolic heap consumption which is parametric on ICap (observe that ICap is a parameter of the second and third calls in the trace). Importantly, this parameter remains as a variable because method realloc is not executed. Symbolic execution of realloc would give a concrete value to ICap when determining the number of iterations of its loop. The test case at the bottom in contrast executes realloc once, as the vector runs out of capacity at the fourth iteration of the loop. Hence, its capacity is duplicated.

Resource-aware TDG has interesting applications. It can clearly be useful to detect, early within the software development process, bugs related to an excessive consumption of resources. Additionally, one of the well-known problems of TDG is that, even for small programs, it produces a large set of test cases which complicate the software testing process which, among other things, requires reasoning on the correctness of the program by verifying that the obtained test cases lead to the expected result. Resource-aware TDG can be used in combination with a *resource policy* in order to filter out test cases which do not adhere to the policy. The resource policy can state that the resource consumption of the test cases must be larger (or smaller) than a given threshold so that one can focus on the (potentially problematic) test cases which consume a certain amount of resources.

Example 5. Let us recall that in Ex. 4 we had obtained 54 test cases. By using a resource policy to focus on those cases that consume more than 48 bytes, we filter out 23 test cases. In a realistic scenario, the user must provide the testing framework with resource consumption parameters. For instance, by setting the amount of memory available in the resource policy, TDG could help us detect (potentially buggy) behaviours of the program under test which exceed the memory limit.

Furthermore, one can display to the user the test cases ordered according to the amount of resources they consume. For instance, for the cost model $\mathcal{M}_{mem}$, the test cases in Ex. 4 would be shown first. It is easy to infer the condition ICap > 9, which determines when the parametric test case is the most expensive one. Besides, one can implement a *worst-case* resource policy which shows to the user only the test case that consumes more resources among those obtained by the TDG process (e.g., the one at the top together with the previous condition for

$\mathcal{M}_{mem}$), or display the n test cases with highest resource consumption (e.g., the two cases in Fig. 5 for $n = 2$).

4 Resource-driven TDG

This section introduces *resource-driven TDG*, a novel heuristics to guide the symbolic execution process which improves, in terms of scalability, over the resource-aware approach, especially in those cases where restrictive resource policies are supplied. The main idea is to try to avoid, as much as possible, the generation of paths during symbolic execution that do not satisfy the policy. If the resource policy imposes a maximum threshold, then symbolic execution can stop an execution path as soon as the resource consumption exceeds it. However, it is often more useful to establish resource policies that impose a minimum threshold. In such case, it cannot be decided if a test case adheres to the policy until it is completely generated. Our heuristics to avoid the unnecessary generation of test cases that violate the resource policy is based on this idea: 1) in a pre-process, we look for traces corresponding to potential paths (or sub-paths) that adhere to the policy, and 2) we use such traces to guide the symbolic execution.

An advantage of relying on a CLP-based TDG approach is that the trace argument of our CLP-transformed programs can be used, not only as an output, but also as an input argument. 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 5 (guided TDG). *Given a method m, a coverage criterion C, and a (possibly partial) trace π, guided TDG generates the set of test cases with traces, denoted $gTDG(m, C, \pi)$, obtained for all successful branches in $\mathcal{T}_m^C$.*

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

Example 6. Let us consider the partial trace multiples(1, [ ], [init(1, [ICap], [ ]), mloop(1,[ ], [mcond(2, [ ], [mif(2, [ ], [A1, mloop(1, [ ], [mcond(2,[ ], [mif(2,[ ], [A2, mloop(1,[ ], [mcond(2, [ ], [mif(2, [ ], [A3, mloop(1, [ ], [mcond(2,[ ], [mif(2,[ ], [A4, mloop(1,[ ], [mcond(1, [ ], [ ])...]), which represents the paths that iterate four times in the *for* loop of method multiples (rules mloop^1 and mcond^2 in the CLP translated program), always following the *then* branch of the *if* statement (rule mif^2), i.e. invoking method add. The trace is partial since it does not specify where the execution goes after method add is called (in other words, whether method realloc is executed or not). This is expressed by the free variables (A1, A2, A3 and A4) in the trace-term arguments. The symbolic execution guided by such trace produces four test cases which differ on the constraint on ICap, which is resp. ICap=1, ICap=2, ICap=3 and ICap$\geq$4. The first and the third test

cases are the ones shown at the top and at the bottom resp. of Fig 5. All the executions represented by this partial trace finish with the evaluation to false of the loop condition (rule mcond^1).

By relying on an oracle O that provides the traces, we now define resource-driven TDG as follows.

Definition 6 (resource-driven TDG). *Given a method m, a coverage criterion C and a resource-policy R, resource-driven TDG generates the set of test cases with traces defined by*

$$\bigcup_{i=1}^{n} gTDG(m, C, \pi_i)i$$

where $\{\pi_1, \ldots, \pi_n\}$ is the set of traces computed by an oracle O w.r.t R and C.

In the context of symbolic execution, there is an inherent need of carrying out a constraint store over the input variables of the program. When the constraint store becomes unsatisfiable, symbolic execution must discard the current execution path and backtrack to the last branching point in the execution tree. Therefore, in general it is not possible to parallelize the process. This is precisely what we gain with deterministic resource-guided TDG. Because the test cases are computed as the union of independent executions, they can be parallelized. Experimenting on a parallel infrastructure remains as future work.

This definition relies on a generic oracle. We will now sketch different techniques for defining specific oracles. Ideally, an oracle should be *sound*, *complete* and *effective*. An oracle is sound if every trace it generates satisfies the resource policy. It is complete if it generates all traces that satisfy the policy. Effectiveness is related to the number of unfeasible traces it generates. The larger the number, the less effective the oracle and the less efficient the TDG process. For instance, assuming a worst-case resource policy, one can think of an oracle that relies on the results of a static cost analyzer [1] to detect the methods with highest cost. It can then generate partial traces that force the execution go through such costly methods (combined with a terminating criterion). Such oracle can produce a trace as the one in Ex. 6 with the aim of trying to maximize the number of times method `add` (the potentially most costly one) is called. This kind of oracle can be quite effective though it will be in general unsound and incomplete.

4.1 On Soundness and Completeness of Oracles

In the following we develop a concrete scheme of an oracle which is sound, complete, and parametric w.r.t. both the cost model and the resource policy. Intuitively, an oracle is complete if, given a resource policy and a coverage criterion, it produces an over-approximation of the set of traces (obtained as in Def. 3) satisfying the resource policy and coverage criterion. We first propose a naive way of generating such an over-approximation which is later improved.

Definition 7 (trace-abstraction 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).*

The trace-abstraction of a program corresponds to its control-flow graph, and can be directly used as a trace-generator that produces a superset of the (usually infinite) set of traces of the program. The coverage criterion is applied in order to obtain a concrete and finite set of traces. Note that this is possible as long as the coverage criterion is structural, i.e., it only depends in the program structure (like loop-k). The resource policy can then be applied over the finite set: (1) in a post-processing where the traces that do not satisfy the policy are filtered out or (2) depending on the policy, by using a specialized search method.

As regards soundness, the intuition is that an oracle is sound if the resource consumption for the selected cost model is *observable* from the traces, i.e, it can be computed and it is equal to the one computed after the guided TDG.

Definition 8 (resource observability). *Given a method m, a coverage criterion C and a cost-model $\mathcal{M}$, we say that $\mathcal{M}$ is* observable *in the trace-abstraction for m, if for every feasible trace π generated from the trace-abstraction using C, we have that $\texttt{cost}(\pi, \mathcal{M}) = \texttt{cost}(\pi', \mathcal{M})$, where π' is a corresponding trace obtained for $gTDG(m, C, \pi)$.*

Observe that π can only have variables in trace parameters (second argument of a trace-term). This means that the only difference between π and π' can be made by means of instantiations (or associated constraints) performed during the symbolic execution on those variables. Trivially, $\mathcal{M}_{ins}$ and $\mathcal{M}_{call}$ are observable since they do not depend on such trace parameters. Instead, $\mathcal{M}_{mem}$ can depend on trace parameters and is therefore non-observable in principle on this trace-abstraction, as we will discuss later in more detail.

Enhancing trace-abstractions. Unfortunately the oracle proposed so far is in general very far from being effective since trace-abstractions can produce a huge amount of unfeasible traces. To solve this problem, we propose to enhance the trace-abstraction with information (constraints and arguments) taken from the original program. This can be done at many degrees of precision, from the empty enhancement (the one we have seen) to the full one, where we have the original program (hence the original resource-aware TDG). The more information we include, the less unfeasible traces we get, but the more costly the process is. The goal is thus to find heuristics that enrich sufficiently the abstraction so that many unfeasible traces are avoided and with the minimum possible information.

A quite effective heuristic is based on the idea of adding to the abstraction those program variables (input arguments, local variables or object fields) which get instantiated during symbolic execution (e.g., field `size` in our example). The idea is to enhance the trace-abstraction as follows. Let us start with a set of variables V initialized with those variables (this can be soundly approximated by

```
multiples(ICap,multiples(1,[],[T1,T2])):- init(ICap,Size,Cap,T1),
                                          mloop(Size,Cap,T2).
mloop(Size,Cap,mloop(1,[],[T])) :- mcond(Size,Cap,T).
mcond1(_,_,mcond(1,[],[])).
mcond2(Size,Cap,mcond(2,[],[T])) :- mif(Size,Cap,T).
mif1(Size,Cap,mif(1,[],[T])) :- mloop(Size,Cap,T).
mif2(Size,Cap,mif(2,[],[T1,T2])) :- add(Size,Cap,NSize,NCap,T1),
                                    mloop(NSize,NCap,T2).
add(Size,Cap,NSize,NCap,add(1,[],[T])) :- if(Size,Cap,NSize,NCap,T).
if1(Size,Cap,NSize,Cap,if(1,[],[T])) :- Size #\= Cap, addc(Size,NSize,T).
if2(Size,Cap,NSize,NCap,if(2,[],[T1,T2])) :- Size #= Cap,
                              realloc(Cap,NCap,T1), addc(Size,NSize,T2).
addc(Size,NSize,addc(1,[],[])) :- NSize #= Size+1.
realloc(Cap,NCap,realloc(1,[NCap],[T])) :- NCap #= Cap*2, loop(Cap,0,T).
loop(Cap,I,loop(1,[],[T])) :- cond(Cap,I,T).
cond1(Cap,I,cond(1,[],[])) :- I #>= Cap.
cond2(Cap,I,cond(2,[],[T])) :- I #< Cap, NI #= I+1, loop(Cap,NI,T).
init1(ICap,0,ICap,init(1,[ICap],[])).
init2(ICap,0,ICap,init(2,[ICap],[])).
```

Fig. 6: Enhanced trace-abstraction program.

means of static analysis). For every $v \in V$, we add to the program all occurrences of v and the guards and arithmetic operations in which v is involved. The remaining variables involved in those guards are added to V and the process is repeated until a fixpoint is reached. Fig. 6 shows the trace-abstraction with the proposed enhancement for our working example, in which variables Size and Cap (fields), ICap (input argument) and I (local variable) are added.

Resource Observability for $\mathcal{M}_{mem}$. As already mentioned, $\mathcal{M}_{mem}$ is in general non-observable in trace-abstractions. The problem is that the memory consumed by the creation of arrays depends on dynamic values which might be not present in the trace-abstraction. Again, this problem can be solved by enhancing the trace-abstraction with the appropriate information. In particular, the enhancement must ensure that the variables involved in the creation of new arrays (and those on which they depend) are added to the abstraction. This information can be statically approximated [2,15,16].

Instances of Resource-driven TDG. The resource-driven scheme has been deliberately defined as generic as possible and hence it could be instantiated in different ways for particular resource policies and cost-models producing more effective versions of it. For instance, for a worst-case resource policy, the oracle must generate all traces in order to know which is the one with maximal cost. Instead of starting a guided symbolic execution for all of them, we can try them

one by one (or k by k in parallel) ordered from higher to lower cost, so that as soon as a trace is feasible the process stops. By correctness of the oracle, the trace will necessarily correspond to the feasible path with highest cost.

Theorem 1 (correctness of trace-driven TDG). *Given a cost model $\mathcal{M}$, a method m, a coverage criterion C and a sound oracle O on which $\mathcal{M}$ is observable, resource-driven TDG for m w.r.t. C using O generates the same test cases as resource-aware TDG w.r.t. C for the cost model $\mathcal{M}$.*

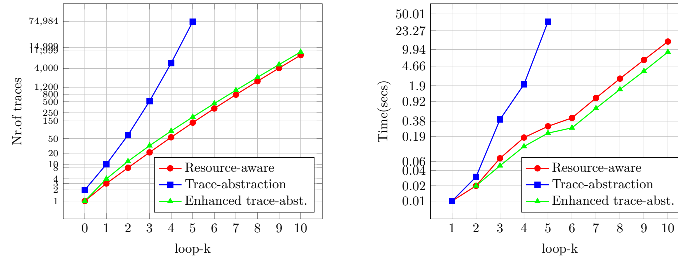
Soundness is trivially entailed by the features of the oracle.

4.2 Performance of Trace-Driven TDG

We have performed some preliminary experiments on our running example using different values for k for the loop-k coverage criterion (X axis) and using a worst-case resource policy for the $\mathcal{M}_{ins}$ cost model. Our aim is to compare resource-aware TDG with the two instances of resource-driven TDG, the one that uses the naive trace-abstraction and the enhanced one. Fig. 7a depicts the number of traces which will be explored in each case. It can be observed that the naive trace-abstraction generates a huge number of unfeasible traces and the growth is larger as k increases. Indeed, from $k = 6$ on, the system runs out of memory when computing them. The enhanced trace-abstraction reduces drastically the number of unfeasible traces and besides the difference w.r.t. this number in resource-aware is a (small) constant. Fig. 7b shows the time to obtain the worst-case test case in each case. The important point to note is that resource-driven TDG outperforms resource-aware TDG in all cases, taking in average half the time w.r.t. the latter. We believe our results are promising and suggest that the larger the symbolic execution tree is (i.e., the more exhaustive TDG aims to be), the larger the efficiency gains of resource-driven TDG are. Furthermore, in a real system, the different test cases for resource-driven TDG could be computed in parallel and hence the benefits would be potentially larger.

5 Conclusions and Related work

In this paper, we have proposed resource-aware TDG, an extension of standard TDG with resources, whose purpose is to build resource consumption into the test cases. Resource-aware TDG can be lined up in the scope of performance engineering, an emerging software engineering practice that strives to build performance into the design and architecture of systems. Resource-aware TDG can serve different purposes. It can be used to test that a program meets performance criteria up to a certain degree of code coverage. It can compare two systems to find which one performs better in each test case. It could even help finding out what parts of the program consume more resources and can cause the system to perform badly. In general, the later a defect is detected, the higher the cost



(a) Number of traces. (b) Time.

Fig. 7: Preliminary experimental results.

of remediation. Our approach allows thus that performance test efforts begin at the inception of the development project and extend through to deployment.

Previous work also considers extensions of standard TDG to generate resource consumption estimations for several purposes (see [6, 13, 21] and their references). However, none of those approaches can generate symbolic resource estimations, as our approach does, neither take advantage of a resource policy to guide the TDG process. The most related work to our resource-driven approach is [19], which 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. An important difference is that our trace-based abstraction is an over-approximation of the actual paths which allows us to select the most expensive paths. In contrast, 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 can be problematic. Besides, our extension to infer the resources from the trace-abstraction and the idea to use it as a heuristics to guide the symbolic execution is new.

Acknowledgements

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 TIN-2008-05624 *DOVES* project, the UCM-BSCH-GR35/10-A-910502 *GPD* Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

References

1. 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*. Springer, 2007.

2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Removing useless variables in cost analysis of java bytecode. In *Proc. of SAC'08*. ACM, 2008.
3. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *Proc. of ISMM '07*. ACM Press, 2007.
4. E. Albert, M. Gómez-Zamalloa, and G. Puebla. PET: A Partial Evaluation-based Test Case Generation Tool for Java Bytecode. In *Proc. of. PEPM'10*. ACM Press, 2010.
5. E. Albert, M. Gómez-Zamalloa, J.M. Rojas, and G. Puebla. Compositional clp-based test data generation for imperative languages. In *LOPSTR 2010 Revised Selected Papers*, volume 6564 of *LNCS*. Springer-Verlag, 2011.
6. J. Antunes, N. F. Neves, and P. Veríssimo. Detection and prediction of resource-exhaustion vulnerabilities. In *Proc. of ISSRE'08*. IEEE Computer Society, 2008.
7. L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.
8. C. Engel and R. Hähnle. Generating unit tests from formal proofs. In *Proc. of TAP'07*, volume 4454 of *LNCS*. Springer, 2007.
9. S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Proc. of PPDP'07*. ACM, 2007.
10. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *JIST*, 51:1409–1427, October 2009.
11. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *TPLP, ICLP'10 Special Issue*, 2010.
12. A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *Computational Logic*, pages 399–413, 2000.
13. A. Holzer, V. Januzaj, and S. Kugele. Towards resource consumption-aware programming. In *Proc. of ICSEA'09*. IEEE Computer Society, 2009.
14. J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
15. M. Leuschel and M.H. Sørensen. Redundant argument filtering of logic programs. In *Proc. of LOPSTR'96*. Springer-Verlag, 1996.
16. M. Leuschel and G. Vidal. Forward Slicing by Conjunctive Partial Deduction and Argument Filtering. In *Proc. of ESOP'05*. Springer-Verlag, 2005.
17. C. Meudec. Atgen: Automatic test data generation using constraint logic programming and symbolic execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.
18. R. A. Müller, C. Lembeck, and H. Kuchen. A symbolic java virtual machine for test case generation. In *IASTED Conf. on Software Engineering*, 2004.
19. N. Rungta, E.G. Mercer, and W. Visser. Efficient testing of concurrent programs with abstraction-guided symbolic execution. In *Proc. of SPIN'09*. Springer, 2009.
20. T. Schrijvers, F. Degrave, and W. Vanhoof. Towards a framework for constraint-based test case generation. In *Proc. of LOPSTR'09*, 2009.
21. J. Zhang and S.C. Cheung. Automated test case generation for the stress testing of multimedia systems. *Softw., Pract. Exper.*, 32(15):1411–1435, 2002.