Testing of Concurrent and Imperative Software using CLP

Elvira Albert
DSIC, Complutense University of Madrid
28040 Madrid, Spain
elvira@fdi.ucm.es

Puri Arenas
DSIC, Complutense University of Madrid
28040 Madrid, Spain
puri@sip.ucm.es

Miguel Gómez-Zamalloa
DSIC, Complutense University of Madrid
28040 Madrid, Spain
mzamalloa@fdi.ucm.es

ABSTRACT

Testing is a vital part of the software development process. In static testing, instead of executing the program on normal values (e.g., numbers), typically the program is executed on symbolic variables representing arbitrary values. Constraints on the symbolic variables are used to represent the conditions under which the execution paths are taken. Testing tools can uncover issues such as memory leaks, buffer overflows, and also concurrency errors like deadlocks or data races. Due to its inherent symbolic execution mechanism and the availability of constraint solvers, Constraint Logic Programming (CLP) has a big potential in the field of testing. In this talk, we will describe a fully CLP-based framework to testing of a today's imperative language. We will also discuss the extension of this framework to handle actor-based concurrency, used in languages such as Go, ActorFoundry, Erlang, and Scala, among others.

CCS Concepts

•**Computing methodologies** → **Concurrent programming languages;** *Distributed programming languages;* Parallel programming languages; •**Theory of computation** → Operational semantics;

Keywords

Testing; Static Analysis; Dynamic Analysis; Concurrency

1. INTRODUCTION

Testing is the most widely-used methodology for software validation in industry. It typically requires at least half of the total cost of a software project. Still, it remains a mostly manual stage within the software development process. Test Case Generation (TCG) is devoted to the automation of a crucial part of the testing process, the generation of input data for interesting *coverage criteria*. Coverage criteria aim at measuring how well the program is exercised by a test suite. Examples of coverage criteria are: *statement coverage* which requires that each line of the code is executed; *path coverage* which requires that every possible trace through a given part of the code is executed. Among the wide variety of approaches to TCG (see e.g. [41]), our work focuses on *glass-box* testing, where test cases are obtained from the concrete program in contrast to *black-box* testing, where they are deduced from a specification of the program. We will initially focus on *static* testing, where we assume no knowledge about the input data, in contrast to *dynamic* approaches [21, 30] which execute the program to be tested for concrete input values, as will be discussed for the concurrent setting.

The standard approach to generating test cases statically is to perform a *symbolic execution* of the program [32, 17, 29, 34, 35, 19, 18], where the contents of variables are expressions rather than concrete values. Symbolic execution produces a system of *constraints* consisting of the conditions to execute the different paths. This happens, for instance, in branching instructions, like if-then-else, where we might want to generate test cases for the two alternative branches and hence accumulate the conditions for each path as constraints. The symbolic execution approach has been combined with the use of *constraint solvers* [35, 29, 18] in order to handle the constraint systems by solving the feasibility of paths and, afterwards, to instantiate the input variables. For instance, a symbolic JVM machine which integrates several constraint solvers has been designed in [35] for TCG of Java (bytecode) programs. In general, a symbolic machine requires non-trivial extensions w.r.t. a non-symbolic one like the JVM: (1) it needs to execute (imperative) code symbolically as explained above, (2) it must be able to non-deterministically execute multiple paths (as without knowledge about the input data non-determinism usually arises).

We overview our CLP-based approach to TCG of imperative programs which consists of four main ingredients: (i) The imperative program is first translated into an equivalent CLP one, named *CLP-translated program* in what follows. The translation can be performed by partial evaluation [25] or by traditional compilation. (ii) Symbolic execution on the CLP-translated program can be performed by relying on the standard evaluation mechanism of CLP, which provides backtracking and handling of symbolic expressions for free. (iii) The use of dynamic memory requires to define heap-related operations that, during TCG, take care of constructing complex data structures with unbounded data (e.g., recursive data structures). Such operations can be implemented in CLP [?]. (iv) We can guide the TCG process towards specific paths by adding to our CLP-translated programs *trace terms* that track the sequence of calls per-

PPDP '16, September 05-07, 2016, Edinburgh, United Kingdom
© 2016 ACM. ISBN 978-1-4503-4148-6/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2967973.2968593

formed. We can supply fully or partially instantiated traces, thus guiding, completely or partially, the symbolic execution towards specific paths.

Finally, we will describe our work on the field of testing concurrent programs [5, 13, 6, 12]. Concurrent programs are becoming increasingly important as multicore and networked computing systems are omnipresent. Writing correct concurrent programs is more difficult than writing sequential ones, because with concurrency come additional hazards not present in sequential programs such as race conditions, deadlocks, and livelocks. Therefore, testing techniques urge especially in the context of concurrent programming. Due to the non-deterministic interleaving of processes, traditional testing for concurrent programs is not as effective as for sequential programs. In order to ensure that all behaviors of the program are tested, the testing process, in principle, must explore all possible non-deterministic ways in which the processes can interleave. This is known as *systematic testing* [16, 39, 40] in the context of concurrent programs. Such full systematic exploration of all process interleavings produces the well known *state explosion* problem and is often computationally intractable (see, e.g., [40] and its references). We will discuss our recent work [5, 13, 6, 12] on defining strategies and heuristics for pruning redundant state-exploration when testing concurrent systems by reducing the amount of unnecessary non-determinism.

2. CLP-BASED STATIC TESTING

This section overviews our CLP-based static testing framework. It was originally proposed for the context of TCG of a simple bytecode language in [14], and later extended to sequential OO programs [**?**] and to concurrent actors [4, 6]. Its implementations for the Java (bytecode) and ABS languages have led to the development of the jPET [15, 8] and aPET [7] tools. The framework takes advantage of the inherent characteristics of CLP, namely, its evaluation mechanism based on backtracking and its constraint solving facilities, for the purpose of symbolic execution. Moreover, it shows that logic programming in general is an adequate paradigm as the basis for reasoning about other programming languages (meta-programming) [10]. The main architecture of the framework is shown in Fig. 1. It consists of three independent phases: (1) First, the program-under-test is translated into an equivalent CLP program. (2) The CLP program is then symbolically executed in CLP relying on CLP's execution mechanism using a termination/coverage criterion. (3) The obtained test-cases are presented to the user in different forms, namely, graphically or as unit tests. This scheme has the important property of being fexible and generic, in the sense that the second and third phases are essentially independent of the concrete target language. Note that most of the concrete features of the target language are translated and uniformly represented in CLP.

2.1 Translation from OO Imperative to CLP programs

The translation of imperative object-oriented programs into equivalent CLP-translated programs has been subject of previous work (see, e.g., [3, 27]). We rely on the so-called *interpretive compilation* by partial evaluation [27]. It consists in compiling the imperative OO program to CLP by partially evaluating an interpreter of the OO imperative language written in CLP w.r.t. the imperative program.

```
int exp(int a,int n) {
  if (n < 0)
    throw new Exception();
  else {
    int r = 1;
    while (n > 0) {
      r = r*a;
      n--;
    }
    return r;
  }
}
```

Figure 2: Imperative code for the exp method

```
exp([A,N],Out,H_in,H_out,EF) :-
    if([A,N],Out,H_in,H_out,EF).
if([A,N],_Out,H_in,H_out,exc(Ref)):-
    N #< 0,
    new_object(H_in,'Exception',Ref,H_out).
if([A,N],Out,H,H,ok) :-
    N #>= 0,
    loop([A,N,1],Out).
loop([_A,N,R],R) :-
    N #=< 0.
loop([A,N,R],Out) :-
    N #> 0,
    R' #= R*A,
    N' #= N-1,
    loop(A,N',R',Out).
```

Figure 3: CLP-translation for the exp method

EXAMPLE 2.1. *Fig. 2 shows the imperative code for method* **exp** *which takes two integer input arguments* a *and* n *and computes* a^n *by successive multiplications. If the value of* n *is less than 0 an exception is thrown. Fig. 3 shows its corresponding (pretty-printed) CLP-translation.*

The main features that can be observed from the translation are: (1) The root predicates for methods (in this case exp/5) include as parameters: the input arguments (as a list), the output argument, the input and output heaps and the exception flag. The rest of the predicates include the required parameters depending on the context. (2) Conditional statements and loops in the source program are transformed into guarded rules and recursion in the CLP program, resp., e.g., rules for while. Mutual exclusion between the rules of a predicate is ensured either by means of mutually exclusive guards, or by information made explicit on the heads of rules, as usual in CLP. (3) The global memory or *heap* is explicitly handled and carried along the execution being used and transformed by the corresponding heap built-ins as a black box. E.g., the new_object/4 operation takes an input heap and a class name, and creates a new object of that class, returning the new heap containing it and its assigned reference. Heaps are therefore represented in the CLP program by means of logic variables (e.g. H_{in} and H_{out}). (4) Exceptional behaviour is handled explicitly in the CLP-translated program by means of the exception flag and exception objects. When an exception is thrown the flag takes the value exp(Ref) being Ref the reference of the corresponding exception object in the heap. Otherwise the value ok is obtained.

2.2 CLP-based Symbolic Execution and TCG

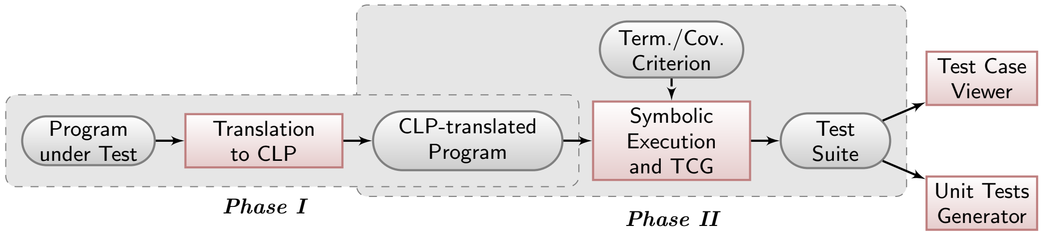


Figure 1: CLP-based static testing framework

The standard CLP execution mechanism, together with a suitable implementation of the heap built-ins, suffices to execute the CLP-translated programs. This can done simply running in a Prolog system a goal with the predicate corresponding to the method under test and fully instantiated input parameters. For instance, we can run the goal $\mathtt{exp}([2,10],\mathtt{Out},[\,],\mathtt{H_{out}},\mathtt{EF})$ to compute 2^{10}. Note that the heap is represented as a list of Reference-Object pairs and therefore $[\,]$ represents an empty heap. As a result the following bounds are obtained: $\mathtt{Out}=\mathtt{1024},\mathtt{H_{out}}=[\,],\mathtt{EF}=\mathtt{ok}$.

One of the main advantages of our CLP-translated programs is that they can be symbolically executed using the standard CLP execution mechanism. To do that we simply run a goal with the predicate corresponding to the method under test and free variables for all its arguments. The inherent constraint solving and backtracking mechanisms of CLP allow keeping track of so called *path conditions* (or constraint stores), failing and backtracking when unsatisfiable constraints are hit, hence discarding such execution paths; and succeeding when satisfiable constraints lead to a terminating state in the program, which in the context of symbolic execution implies that a new solution (or test case) is generated.

Example 2.2. *Let us perform a symbolic execution of the* **exp** *method by running the goal* $\mathtt{exp}([\mathtt{A},\mathtt{N}],\mathtt{Out},\mathtt{H_{in}},\mathtt{H_{out}},\mathtt{EF})$. *As a first solution we get*

$$\mathtt{N}<\mathtt{0},\mathtt{H_{out}}=[(\mathtt{R},\mathtt{object}('\mathtt{Exception}',\ldots))|\mathtt{H_{in}}],\mathtt{EF}=\mathtt{exc}(\mathtt{R})$$

which reads as: if $\mathtt{N}<\mathtt{0}$ *the execution ends with an uncaught exception whose associated object is* $\mathtt{R}$. *If we ask for another solution we get*

$$\mathtt{N}=\mathtt{0},\mathtt{Out}=\mathtt{1},\mathtt{H_{out}}=\mathtt{H_{in}},\mathtt{EF}=\mathtt{ok}$$

which reads as: if $\mathtt{N}=\mathtt{0}$ *we get* 1 *as output, regardless of the value of* $\mathtt{A}$, *and the heap does not change. The third solution is:*

$$\mathtt{N}=\mathtt{1},\mathtt{Out}=\mathtt{A},\mathtt{H_{out}}=\mathtt{H_{in}},\mathtt{EF}=\mathtt{ok}$$

It is well-known that the symbolic execution tree (SLD tree in this framework) is in general infinite. This is because iterative constructs such as loops and recursion, whose number of iterations depend on input arguments, usually induce an infinite number of execution paths when executed with symbolic input values. This happens for instance in the symbolic execution of the `exp` method. It is therefore essential to establish a termination criterion. Such a termination criterion can be expressed in different forms. For instance, a computation time budget can be established, or an explicit bound on the depth of the symbolic execution tree can be imposed. In our framework we adopt a more code-oriented termination criterion, which consists in imposing an upper bound on the number of times each loop (or recursive call) is iterated. This can be easily implemented in CLP as a meta-interpreter which controls and limits the number of recursive calls made on each predicate [14].

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

Example 2.3. *The following concrete test-cases are obtained by our framework for the* `exp` *method if we set a limit of at most* 2 *loop iterations and the domain* $-10..10$ *to numeric variables:*

#	*Input (a,n)*	*Output*
1	*(-10, -10)*	*Uncaught Exception*
2	*(-10, 0)*	*1*
3	*(-10, 1)*	*-10*
4	*(-10, 2)*	*100*

As an example, the following JUnit test could be automatically generated for the second test-case:

```
    public void testExp2(){
        int input0 = -10, input1 = 0;
        int output = exp(input0,input1);
        int expected = 1;
        assertEquals(expected,output);
    }
```

Handling Heap-manipulating Programs.

One of the main challenges in symbolic execution is to correctly and efficiently handle heap-manipulating programs [**?**]. This kind of programs often create and use complex and possibly aliased dynamically heap-allocated data structures. Symbolic execution must consider all possible shapes these dynamic data structures can take. In trying to do so, scalability issues arise since many (even an exponential number of) shapes may be built due to the aliasing of references. This lead us to the development of a *heap solver* [9] which enables a more efficient treatment of reference aliasing in symbolic execution by means of disjunctive reasoning, and the use of advanced back-propagation of heap related constraints.

2.3 Guided Testing

It is well known that symbolic execution presents scalability problems when it is applied on realistic programs. Also, in the context of static testing, this complicates human reasoning on the generated test cases. Guided testing [36, 37] aims at steering symbolic execution towards specific program paths in order to efficiently generate more relevant test cases and filter out less interesting ones with respect to a given selection criterion. The goal is thus to improve on scalability and efficiency by achieving a high degree of control over the coverage criterion. This has potential applicability for industrial software testing practices such as unit testing, where units of code (e.g. methods) must be thoroughly tested in isolation, or selective testing, in which only specific paths of a program must be tested. The intuition of guided testing is the following: (1) A heuristics-based trace-generator generates possibly partial traces, i.e., partial descriptions of paths, according to a given selection criterion. This can be done by relying on the control-flow graph of the program. (2) Bounded symbolic execution is guided by the obtained traces. The process is repeated until the selection criterion is satisfied or until no more traces are generated.

We can define a concrete methodology for guided testing in our CLP-based framework as follows:

1. We instrument our CLP-translated programs so that they generate as an additional result so called *trace-terms*. Trace-terms are of the form: $p(K, [T_1, \ldots, T_n])$, where p is the name of the predicate, K a natural number indicating the concrete predicate rule, and, $T_1, \ldots, T_n$ the trace-terms of the calls in the body of the K-th rule of p. Trace-terms are tree-like representations of execution traces and enable us to keep track of the sequence (and order) of rules executed in each derivation.

2. We define a *trace-generator* which generates a set of, possibly partially instantiated, trace-terms according to a given selection criterion.

3. A set of symbolic executions is performed (possibly in parallel), each of them using as input a different trace-term. Trace-terms allow guiding, completely or partially, the symbolic execution towards specific paths.

EXAMPLE 2.4. *Let us consider selective testing for method* `exp`*. As a selection criterion, e.g., one could be interested in generating a test-case that raises an exception. The challenge is to generate such a test avoiding traversing as much as possible the rest of the paths. Fig. 4 shows the CLP-translated program instrumented with trace-terms (step 1 above). Let us observe the additional trace-term parameter on each program predicate. An effective trace-generator for the above criterion would be able to generate the trace* `exp(1, [if(2, _)])` *(step 2). A symbolic execution using such trace-term as input will generate only the path that raises the exception (step 3).*

In [36] we define concrete trace-generators and guided testing schemes for two different selection criteria, and demonstrate its effectiveness via an experimental evaluation. We also discuss about two central aspects of guided testing, namely *completeness* and *effectiveness*.

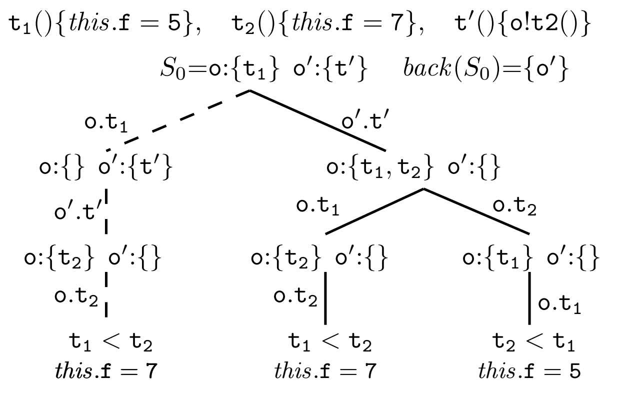
```
exp([A,N],Out,H_in,H_out,EF,exp(1,[T])) :-
    if([A,N],Out,H_in,H_out,EF,T).
if([A,N],_Out,H_in,H_out,exc(Ref),if(1,[~])):-
    N #< 0,
    new_object(H_in,'Exception',Ref,H_out).
if([A,N],Out,H,H,ok,if(2,[T])) :-
    N #>= 0,
    loop([A,N,1],Out,T).
loop([_A,N,R],R,loop(1,[~])) :-
    N #=< 0.
loop([A,N,R],Out,loop(2,[T])) :-
    N #> 0,
    R' #= R*A,
    N' #= N-1,
    loop(A,N',R',Out,T).
```

Figure 4: CLP-translation instrumented with traces for exp

3. TESTING CONCURRENT (ACTOR) PROGRAMS

We consider actor systems [2, 31], a model of concurrent programming that has been gaining popularity and that it is being used in many systems (such as Go, ActorFoundry, Asynchronous Agents, Charm++, E, ABS, Erlang, and Scala). The Actor Model is having also influence on commercial practice, namely Twitter has used actors for scalability, also, Microsoft has used the actor model in the development of its asynchronous agents library.

Actor programs consist of computing entities called actors, each with its own local state and thread of control, that communicate by exchanging messages asynchronously. An actor configuration consists of the local state of the actors and a set of pending *tasks*. In response to receiving a message, an actor can update its local state, send messages, or create new actors. At each step in the computation of an actor system, firstly an actor and secondly a process of its pending tasks are scheduled. We consider a language for distributed and concurrent programming which, in addition to the usual sequential instructions, has two instructions for concurrency: $x = \texttt{new}\ C$ which allows the dynamic creation of an actor x of class C, and $x\ !\ m(\bar{z})$ which spawns a new task or process within the actor x to execute $m(\bar{z})$. As actors do not share their states, in testing one can assume [39] that the evaluation of all statements of a task takes place serially (without interleaving with any other task) until it releases the processor (gets to a return instruction). In this context, transitions correspond to the execution of complete tasks. In particular, a transition or derivation step $S_i \xrightarrow{t} S_{i+1}$ denotes that the task t of an existing actor in S_i has been selected and fully executed, resulting in a modified state S_{i+1}. This corresponds to the notion of macro-step semantics in [39]. Both the selection of which actor executes and which task within the selected component is scheduled is non-deterministic. In order to ensure that all behaviors of the program are tested, the testing process, in principle, must systematically explore all possible ways in which the processes can interleave. This is known as *systematic testing* [16, 39, 40] in the context of concurrent programs. Such full systematic exploration of all process interleavings



$\mathtt{t}_1()\{\mathit{this}.\mathtt{f}=5\}$, $\mathtt{t}_2()\{\mathit{this}.\mathtt{f}=7\}$, $\mathtt{t}'()\{\mathtt{o!t2}()\}$
S_0=o:{t₁} o':{t'} $back(S_0)$={o'}
o.t₁
o'.t'
o:{} o':{t'}
o:{t₁,t₂} o':{}
o'.t'
o.t₁
o.t₂
o:{t₂} o':{}
o:{t₂} o':{}
o:{t₁} o':{}
o.t₂
o.t₂
o.t₁
t₁ < t₂
this.f = 7
t₁ < t₂
this.f = 7
t₂ < t₁
this.f = 5

Figure 5: An execution tree from a given state

produces the well known *state explosion* problem and is often computationally intractable (see, e.g., [40] and its references).

Example 3.1. *Consider the program in Fig. 5, where we have a state with two actors* o *and* o', *each of them with a task* ($\mathtt{t}_1$ *and* $\mathtt{t}'$ *respectively) in its queue. The complete execution tree contains three branches with the results this*.f = 7, *this*.f = 7 *and this*.f = 5 *respectively.*

3.1 The Path Explosion Problem

The challenge of systematic testing of concurrent programs in general is to avoid as much as possible the exploration of *redundant* paths which lead to the same configuration. There are two levels of non-determinism:

1. *actor-selection*, the selection of which actor executes, and
2. *task-selection*, the selection of the task within the selected actor.

Such non-determinism might result in different configurations, and they all need to be explored as only some specific interleavings may reveal the bugs.

Partial-order reduction (POR) [23, 20, 24] is a general theory that helps mitigate the state-space explosion problem by formally identifying equivalence classes of redundant explorations. The basic observation that motivates these techniques is that, in general, the set of executions from a state S contains many redundant derivations. Basically, given a derivation where there are two consecutive transitions which are "independent", i.e., whose execution does not interfere with each other, changing their order of execution will not modify their combined effect. More formally, two macro-step transitions t_1, t_2, possibly belonging to different actors, are *independent* if:

1. they do not *enable* each other, i.e., the execution of t_1 does not lead to introducing t_2, or viceversa, and
2. for every state S in which they are both enabled, there is a unique state S_2 such that $S \xrightarrow{t_1} S_1 \xrightarrow{t_2} S_2$ and $S \xrightarrow{t_2} S_1' \xrightarrow{t_1} S_2$, i.e., they can commute.

Conversely, two transitions are *dependent* iff they are not independent. Transition dependencies can thus be categorized in:

- *enabling dependencies*, if one of the transitions enables the other one, and,
- *interacting dependencies*, if they can be both enabled and their combined effect varies with their order.

A complete derivation thus represents an equivalence class of similar derivations that can be obtained by swapping adjacent independent transitions. The so-called *happens-before* relation [22], written $<_d$, is used to characterize these equivalence classes. Given a derivation $S_0 \xrightarrow{t_1} \cdots \xrightarrow{t_n} S_n$, we say that transition t_i *happens-before* transition t_j, written $t_i <_d t_j$, if $i < j$ and t_i is dependent with t_j. The happens-before relation is a partial-order relation, hence the name POR. Furthermore, two derivations are *redundant* if they have the same happens-before relation.

Example 3.2. *Consider the execution tree in Fig. 5. The happens-before relation, or partial order, of the leftmost and middle derivations is* $\{\mathtt{t}' <_d \mathtt{t}_2, \mathtt{t}_1 <_d \mathtt{t}_2\}$. *Thus both derivations compute the same result this*.f = 7. *However the rightmost derivation has as partial order* $\{\mathtt{t}' <_d \mathtt{t}_2, \mathtt{t}_2 <_d \mathtt{t}_1\}$, *computing this*.f = 5 *as result.*

The goal of POR methods is to detect these redundant executions and ideally generate only one representative derivation for each equivalence class and with the minimum number of explored states. Early POR algorithms were based on different static analyses to detect and avoid exploring redundant derivations. The state-of-the-art POR algorithm [22] DPOR (*Dynamic POR*), improves over those approaches by dynamically detecting and avoiding the exploration of redundant derivations on-the-fly. Since the invention of DPOR, there have been several works [1, 16, 39, 40, 38] proposing improvements, variants and extensions in different contexts to the original DPOR algorithm. The most notable one is [1] which proposes an improved DPOR algorithm which further reduces redundant computations ensuring that only one derivation per equivalence class is generated. Some of these works [40, 39, 33] have addressed the application of POR to the context of actor systems from different perspectives. The most recent one [40] presents the TransDPOR algorithm, which extends DPOR to take advantage of a specific property in the dependency relations in pure actor systems, namely transitivity, to explore fewer configurations than DPOR.

Intuitively a DPOR algorithm carries out the exploration of the execution tree using POR. Each node (i.e., state) in the execution tree is evaluated with a backtracking set *back*, which is used to store those actors that must be explored from this node. The backtracking set in the initial state is empty. DPOR algorithms look dynamically at occurring interacting dependencies (i.e., tasks that can be both enabled in a state and their combined effect varies with their execution order) and only backtracks at the those states in which it is possible to reverse them. The TransDPOR algorithm [40] uses an over-approximation $<_d^{\alpha}$ of $<_d$ which considers as dependent those tasks which belong to the same actor. Thus once an actor has been selected the algorithm always tries with the selection of all tasks, since they are in principle considered to be dependent with each other. Instead, at the level of actor selection, the back set is dynamically updated only with the actors that need to be explored. In particular, an actor is added to back only if during the execution the algorithm realizes that it was needed because a new task t of an actor a previously selected appears. This situation might indicate an interacting dependency, and therefore, the algorithm must try to explore the reverse reordering, by updating the back set of the last state S in which a was used. As a simple example, consider a state S in which an actor a_1 with a unique task t_1 is selected. Now, assume that when the execution proceeds, a new task t_2 of a_1 is spawned by the execution of a task t' of an actor a_2 and that t' was in S. This means that it is required to consider also first the execution of t_2 and, next the execution of t_1, since it represents a different partial order between the tasks of a_1. This is accomplished by adding a_2 to the *back* set of S, which allows exploring the execution in which a_2 is selected before a_1 at S, and thus considering the partial order $t_2 <_d^\alpha t_1$. The formal process of updating the *back* sets (and its optimization with *freeze-flags* to avoid further redundancy) can be found at [40].

Example 3.3. *Figure 5 shows the execution tree explored by TransDPOR using the over-approximation* $<_d^\alpha$. *We distinguish two types of edges: dotted edges that will be removed later when introducing the notion of stable actor in Sec. 3.2, and normal edges which are introduced when an actor has been selected and thus all its tasks must be executed. The unique back set updated by TransDPOR is the one associated to the root of the tree. Concretely,* $\mathtt{o}'$ *is introduced in* $back(S_0)$ *when executing* $\mathtt{o}'.\mathtt{t}'$ *in the left branch, what generates the new state* $\mathtt{o}{:}\{\mathtt{t}_2\}, \mathtt{o}'{:}\{\}$. *At this point, since* $\mathtt{o}.\mathtt{t}_1$ *was previously executed from* S_0, $\mathtt{t}_2$ *does not belong to the queue of* $\mathtt{o}$ *in* S_0 *but* $\mathtt{t}'$ *belongs and introduces* $\mathtt{t}_2$ *in the queue of* $\mathtt{o}$ *in* S_0, *then* $\mathtt{t}'$ *must be added to* $back(S_0)$.

The techniques that we describe below enhance previous approaches with novel strategies to further prune redundant state exploration, and that can be easily integrated within the aforementioned algorithms.

3.2 Stable Objects

In previous DPOR algorithms actors are selected arbitrarily. As noticed in [33], the pruning that can be achieved using DPOR algorithms is highly dependent on the order in which tasks are considered for processing. Consider the execution tree in Figure 5. We can see that the same partial order $\mathtt{t}_1 <_d^\alpha \mathtt{t}_2$ occurs in the executions computing $\mathit{this}.\mathtt{f} = 7$ as result. Hence, the dotted subtree can be removed by considering only the rightmost one.

The notion of *temporal stability* allows us to guide the selection of actors so that the search space can be pruned further and redundant computations avoided. An actor is *stable* if there is no other actor different from it that introduces tasks in its queue. Basically, this means that the actor is autonomous since its execution does not depend on any other actor. In general, it is quite unlikely that an actor is stable in a whole execution. However, if we consider the tasks that have been spawned in a given state, it is often the case that we can find an actor that is temporarily stable w.r.t. the actors in that state.

Example 3.4. *Let us re-consider the exploration of the example in Fig. 5 with our improved actor selection function. Observe that at the root node, actor* $\mathtt{o}$ *is not temporarily stable because in the queue of* $\mathtt{o}'$ *there is a call* $\mathtt{t}'$, *and in the body of method* $\mathtt{t}'$ *there is also a call to method* $\mathtt{t}_2$ *of object* $\mathtt{o}$ *(i.e.,* $\mathtt{o}$ *can possibly be modified by* $\mathtt{o}'$*). However,*

```
int f = 1; int g = 1;
void r0() {this.f++; }
void r1() {this.g = this.g*2; }
void r2() {this.g++; }
```

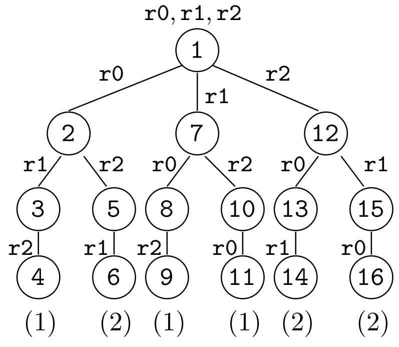


Figure 6: Task independence

actor $\mathtt{o}'$ *is temporarily stable at the root. Our algorithm will therefore select* $\mathtt{o}'$. *The rightmost subtree in Fig. 5 corresponds to the state space explored by our algorithm, in which we can observe that there is no redundant state exploration (with the over-approximation of dependencies used).*

3.3 Independent Tasks

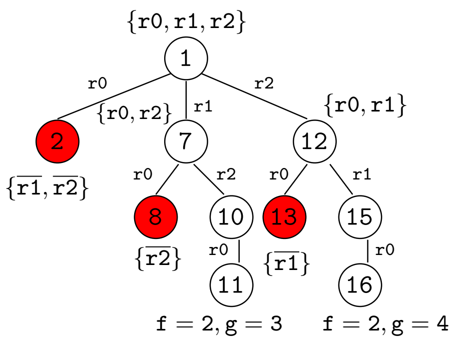
As mentioned in Sec. 3.1, the notion of task dependency has to be over-approximated in order to be used in practice within DPOR. The precision of this over-approximation can be crucial for the effectiveness of DPOR, as the following example shows.

Example 3.5. *Consider the program in Fig. 6, where all methods belong to the same class which contains two fields* $\mathtt{f}$ *and* $\mathtt{g}$, *both of them initialized to* 1. *Using the theoretical definition of task dependency, there are only two equivalence classes of non-redundant executions (branches labeled with (1) and (2)), whereas using the over-approximation of [40] there are six equivalence classes (the complete execution tree).*

The over-approximation of [40] could be improved by looking at shared memory accesses among tasks of the same actor. This way, two tasks belonging to the same actor would have an interacting dependency only if they access a non-disjoint area of their shared memory. In our example, this improved over-approximation would detect task `r0` independent to both `r1` and `r2`, hence allowing DPOR to behave in an optimal way. We call `dep` to this particular relation.

Example 3.6. *Let us consider the program in Fig. 6. Assume that* $R(t)$ *and* $W(t)$ *denote the set of fields that are read and written, respectively, in task* t. *We have that* $R(\mathtt{r0}) = W(\mathtt{r0}) = \{\mathtt{f}\}$, $R(\mathtt{r1}) = W(\mathtt{r1}) = \{\mathtt{g}\}$ *and* $R(\mathtt{r2}) = W(\mathtt{r2}) = \{\mathtt{g}\}$. *Thus,* $\mathtt{r1}, \mathtt{r2}$ *are dependent but* $\mathtt{r0}, \mathtt{r1}$ *and* $\mathtt{r0}, \mathtt{r2}$ *are independent.*

In order to use the `dep` relation in testing, we have defined a specialized algorithm at the task selection level which takes full advantage of the over-approximation used in it. The algorithm makes use of marks in the tasks so that the elements in the queues can now be marked or unmarked, written t or



node 1: $\neg\mathsf{dep}(\mathtt{r0},\mathtt{r1})$ $\neg\mathsf{dep}(\mathtt{r0},\mathtt{r2})$
node 7: $\neg\mathsf{dep}(\mathtt{r0},\mathtt{r2})$
node 12: $\neg\mathsf{dep}(\mathtt{r0},\mathtt{r1})$

Figure 7: Exploration performed by DPOR applying marks

$\overline{t}$ respectively. The mark indicates that the task cannot be selected in the next selection step. The intuition of the algorithm is as follows. Let us consider a state S with an actor a whose queue $\mathcal{Q}$ contains a list of unmarked tasks $[t_1, \ldots, t_n]$. The algorithm tries with the selection of all tasks on backtracking. Let us have tasks $t_i, t_j \in \mathcal{Q}$ with $i < j$ such that $\neg\mathsf{dep}(t_i, t_j)$. When a task t_i is selected, the task t_j is marked for the next state. In the following state S' in which actor a is selected, task t_j cannot be chosen, hence avoiding the derivation in which t_i is executed immediately before t_j in a. However, the derivation in which t_j is selected in a immediately before t_i is allowed to be expanded. In this case t_i has not been marked since it does not go in the list after t_j. Also, if at state S', some other task t_k with $\mathsf{dep}(t_k, t_j)$ is selected, then t_j gets unmarked therefore allowing expanding the derivations with partial orders $t_i <_d^{\alpha} t_k, t_k <_d^{\alpha} t_j$ and $t_j <_d^{\alpha} t_k, t_k <_d^{\alpha} t_i$ which are not redundant.

EXAMPLE 3.7. *Consider again the execution of the program in Fig. 6. Figure 7 shows the actions performed by our specialized task selection algorithm from state 1. Initially all tasks are unmarked. Thus, task* r0 *is selected from* [r0, r1, r2] *and no tasks are unmarked. Then, our task selection algorithm marks tasks* r1 *and* r2 *since* $\neg\mathsf{dep}(\mathtt{r0},\mathtt{r1})$ *and* $\neg\mathsf{dep}(\mathtt{r0},\mathtt{r2})$. *This derivation is therefore cut at state 2. Afterwards, the selection of* r1 *at state 1 does not mark any task. However, when selecting* r0 *at state 7, task* r2 *is marked since* $\neg\mathsf{dep}(\mathtt{r0},\mathtt{r2})$. *Hence, at node 8 the derivation is cut since all tasks are marked. Similarly, at node 13 the derivation is cut because* $\neg\mathsf{dep}(\mathtt{r0},\mathtt{r1})$. *The only expanded derivations are those ending in nodes 11 and 16 which correspond to the partial orders* $\mathtt{r1} <_d^{\alpha} \mathtt{r2}$ *and* $\mathtt{r2} <_d^{\alpha} \mathtt{r1}$, *respectively.*

4. CONCLUSIONS AND FUTURE WORK

We have presented a generic TCG framework that uses CLP as enabling technology, it allows an efficient handling of heap-manipulating programs, as well as guiding the process. We have also discussed the challenges of extending such basic framework to handle concurrent actor systems. As future work, we plan to define new notions of dependency that can be used to improve the detection of independent tasks. Also, we have recently worked on the extension of our framework for concurrent testing so that it can be driven towards paths that potentially lead to deadlock [12].

Acknowledgments

This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu), by the Spanish MINECO projects TIN2012-38137 and TIN2015-69175-C4-2-R, and by the CM project S2013/ICE-3006.

5. REFERENCES

[1] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas. Optimal Dynamic Partial Order Reduction. In *Proc. of POPL'14*, pages 373–384. ACM, 2014.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Procs. of ESOP'07*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.

[4] Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Symbolic Execution of Concurrent Objects in CLP. In *Proc. of PADL'12*, LNCS, pages 123–137. Springer, 2012.

[5] Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Actor- and Task-Selection Strategies for Pruning Redundant State-Exploration in Testing. In *Proc. of FORTE'14*, volume 8461 of *LNCS*, pages 49–65. Springer, 2014.

[6] Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Test Case Generation of Actor Systems. In *Proc. of ATVA'15*, volume 9364 of *LNCS*, pages 259–275. Springer, 2015.

[7] Elvira Albert, Puri Arenas, Miguel Gómez-Zamalloa, and Peter Y.H. Wong. aPET: A Test Case Generation Tool for Concurrent Objects. In *Proc. of ESEC/FSE'13*, pages 595–598. ACM, 2013.

[8] Elvira Albert, Israel Cabañas, Antonio Flores-Montoya, Miguel Gómez-Zamalloa, and Sergio Gutiérrez. jPET: an Automatic Test-Case Generator for Java. In *Proc. of WCRE'11*, pages 441–442. IEEE Computer Society, 2011.

[9] Elvira Albert, María García de la Banda, Miguel Gómez-Zamalloa, José Miguel Rojas, and Peter Stuckey. A CLP Heap Solver for Test Case Generation. *Theory and Practice of Logic Programming*, 13(4-5):721–735, July 2013.

[10] Elvira Albert, Miguel Gómez-Zamalloa, Laurent Hubert, and Germán Puebla. Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In *Proc. of PADL'07*, number 4354 in LNCS, pages 124–139. Springer, 2007.

[11] Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. Combining Static Analysis and Testing for Deadlock Detection. In *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, volume 9681 of

Lecture Notes in Computer Science, pages 409–424. Springer, 2015.

[12] Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. Combining Static Analysis and Testing for Deadlock Detection. In *Proc. of IFM'16*, volume 9681 of *LNCS*, pages 409–424. Springer, 2016.

[13] Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. SYCO: A Systematic Testing Tool for Concurrent Objects. In *Proc. of CC'16*, pages 269–270. ACM, 2016.

[14] Elvira Albert, Miguel Gómez-Zamalloa, and Germán Puebla. Test Data Generation of Bytecode by CLP Partial Evaluation. In *Proc. of LOPSTR'08*, number 5438 in LNCS, pages 4–23. Springer, 2009.

[15] Elvira Albert, Miguel Gómez-Zamalloa, and Germán Puebla. PET: A Partial Evaluation-based Test Case Generation Tool for Java Bytecode. In *Proc. of PEPM'10*, pages 25–28. ACM Press, 2010.

[16] Maria Christakis, Alkis Gotovos, and Konstantinos F. Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *Proc. of ICST'13*, pages 154–163. IEEE Computer Society, 2013.

[17] Lori A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[18] F. Degrave, T. Schrijvers, and W. Vanhoof. Towards a Framework for Constraint-Based Test Case Generation. In *Proc. of LOPSTR'09*, volume 6037 of *LNCS*, pages 128–142. Springer, 2010.

[19] C. Engel and R. Hähnle. Generating Unit Tests from Formal Proofs. In *Proc. of TAP'07*, volume 4454 of *LNCS*, pages 169–188. Springer, 2007.

[20] J. Esparza. Model checking using net unfoldings. *Sci. Comput. Program.*, 23(2-3):151–195, 1994.

[21] Roger Ferguson and Bogdan Korel. The Chaining Approach for Software Test Data Generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[22] C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *Proc. of POPL'05*, pages 110–121. ACM, 2005.

[23] P. Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In *Proc. of CAV'91*, volume 531 of *LNCS*, pages 176–185. Springer, 1991.

[24] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.

[25] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *JIST*, 51:1409–1427, October 2009.

[26] 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.

[27] Miguel Gómez-Zamalloa, Elvira Albert, and Germán Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology*, 51(10):1409–1427, October 2009.

[28] Miguel Gómez-Zamalloa, Elvira Albert, and Germán Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *Theory and Practice of Logic Programming, 26th Int'l. Conference on Logic Programming (ICLP'10) Special Issue*, 10(4-6):659–674, July 2010.

[29] A. Gotlieb, B. Botella, and M. Rueher. A CLP Framework for Computing Structural Test Data. In *Proc. CL'00*, LNAI 1861, pp. 399–413. Springer, 2000.

[30] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Generating Test Data for Branch Coverage. In *Proc. of ASE'00*, pages 219–228. IEEE Computer Society, 2000.

[31] P. Haller and M. Odersky. Scala Actors: Unifying Thread-based and Event-based Programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[32] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[33] Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. Evaluating Ordering Heuristics for Dynamic Partial-Order Reduction Techniques. In *Proc. of FASE'10*, volume 6013 of *LNCS*, pages 308–322. Springer, 2010.

[34] Christophe Meudec. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.

[35] Roger A. Müller, Christoph Lembeck, and Herbert Kuchen. A Symbolic Java Virtual Machine for Test Case Generation. In *Proc. of IASTEDSE'04*, pages 365–371. ACTA Press, 2004.

[36] J. M. Rojas and M. Gómez-Zamalloa. A framework for guided test case generation in constraint logic programming. In *LOPTR 2012*, volume 7844 of *LNCS*. Springer, 2013.

[37] N. Rungta, E.G. Mercer, and W. Visser. Efficient Testing of Concurrent Programs with Abstraction-Guided Symbolic Execution. In *SPIN'09*, LNCS 5578. Springer, 2009.

[38] Olli Saarikivi, Kari Kahkonen, and Keijo Heljanko. Improving Dynamic Partial Order Reductions for Concolic Testing. In *Proc. of ACSD'12*, pages 132–141. IEEE Computer Society, 2012.

[39] Koushik Sen and Gul Agha. Automated Systematic Testing of Open Distributed Programs. In *Proc. of FASE'06*, volume 3922 of *LNCS*, pages 339–356. Springer, 2006.

[40] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In *Proc. of FMOODS/FORTE'12*, volume 7273 of *LNCS*, pages 219–234. Springer, 2012.

[41] H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.