Towards Testing Concurrent Objects in CLP*

Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa

DSIC, Complutense University of Madrid (UCM), E-28040 Madrid, Spain
`{elvira,puri}@sip.ucm.es, mzamalloa@fdi.ucm.es`

Abstract

Testing is a vital part of the software development process. It is even more so in the context of concurrent languages, since due to undesired task interleavings and to unexpected behaviours of the underlying task scheduler, errors can go easily undetected. This paper studies the extension of the CLP-based framework for glass-box test data generation of sequential programs to the context of *concurrent objects*, a concurrency model which constitutes a promising solution to concurrency in OO languages. Our framework combines standard termination and coverage criteria used for testing sequential programs with specific criteria which control termination and coverage from the concurrency point of view, e.g., we can limit the number of task interleavings allowed and the number of loop unrollings performed in each parallel component, etc.

1998 ACM Subject Classification D.2.5 Testing and Debugging, D.1.3 Concurrent Programming, D.1.6 Logic Programming, D.1.5 Object-oriented Programming

Keywords and phrases Testing, Glass-box Test Data Generation, Active Objects, Symbolic Execution

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.98

1 Introduction

Due to increasing performance demands, application complexity and multi-core parallelism, concurrency is omnipresent in today's software applications. It is widely recognized that concurrent programs are difficult to develop, debug, test and analyze. This is even more so in the context of concurrent *imperative* languages that use a global memory (called *heap* in what follows) to which the different tasks can access. The focus of this paper is on the development of automated techniques for testing *concurrent objects*. The actor-based paradigm [1] on which concurrent objects are based has lately regained attention as a promising solution to concurrency in OO languages. For many application areas, standard mechanisms like threads and locks are too low-level and have shown to be error-prone and, more importantly, not *modular* enough. The concurrent objects model is based on considering objects as the concurrency units, i.e., each object conceptually has a dedicated processor (and can run in parallel with other objects). Communication is based on asynchronous method calls with standard objects as targets. An essential difference with thread-based concurrency is that task scheduling is *cooperative*, i.e., switching between tasks of the same object happens only at specific scheduling points during the execution, which are explicit in the source code and can be syntactically identified.

* 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 and PRI-AIBDE-2011-0900 projects, by UCM-BSCH-GR35/10-A-910502 grant and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 98–108
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Test data generation (TDG) is the process of automatically generating *test inputs* for interesting *coverage criteria*. The standard approach to (glass-box) TDG is to perform a *symbolic execution* of the program [12, 4, 9, 15, 16, 6, 17], 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 case* to refer to such constraints. The CLP-based approach to glass-box TDG [8] is based on the idea of translating the program to be tested (written in some imperative language) into an equivalent CLP program. The key idea is that test cases can be obtained by executing the CLP-translated program using the standard symbolic execution mechanism of CLP. When the original language includes features which are not supported, or do not have the same behavior, as in CLP, e.g., the use of a heap or primitives for concurrency different from those of CLP, specific *built-in* operations must be implemented entirely in CLP in order to handle them, see [8, 2]. Then, symbolic execution simply consists in executing the translated CLP program together with the predefined built-ins. In particular we leverage typical termination and coverage criteria for sequential programs (e.g., loop-k) to the concurrent setting and, besides, combine them with novel criteria to ensure interesting coverage of the concurrent behaviors (e.g., we can limit and control the number of task switches). We ensure *fairness* in the selection of objects whose tasks are being tested by applying a coverage criterion that limits task switches at the object level.

2 Symbolic Execution of Concurrent Objects

In this section, we summarize symbolic execution of concurrent objects, as presented in [2]. Essentially, the process is formalized in two steps: first the program is translated into a CLP program which contains some built-in predicates to handle the heap and concurrency primitives and, second, an implementation entirely in CLP of the built-ins is provided such that symbolic execution can be then performed by just relying on the standard symbolic execution engine of CLP.

2.1 CLP Translated Programs

The imperative language with concurrent objects we consider is basically the subset of the ABS language [11] which is relevant to define the TDG framework. A *program* consists of a set of classes C, where C is defined as $\mathbf{class}\ \mathsf{C}[(\overline{\mathsf{T}\ \mathsf{x}})]\{\overline{\mathsf{T}\ \mathsf{x}};\overline{\mathsf{M}}\}$. Each “$\mathsf{T_i\ x_i}$” declares a field $\mathsf{x_i}$ of type $\mathsf{T_i}$, and each $\mathsf{M_i}$ is a method definition which takes the form $\mathsf{T\ m(T_1\ x_1,\ldots,T_n\ x_n)}\{\overline{\mathsf{T\ z}};\ \bar{\mathsf{s}}\}$, where T is the type of the return value; $\mathsf{x_1,\ldots,x_n}$ are the formal parameters and $\overline{\mathsf{T\ z}}$ are local variables. Finally, $\bar{\mathsf{s}}$ is a sequence of instructions which adhere to the following grammar:

$$
\begin{array}{lcl}
\mathsf{s} & ::= & \mathsf{x} = \mathsf{rhs} \mid \mathbf{await}\ \mathsf{g} \mid \mathbf{return}\ \mathsf{e} \mid \mathbf{if}\ (\mathsf{b})\,\{\,\mathsf{s}\,\}\,[\,\mathbf{else}\,\{\,\mathsf{s}\,\}] \mid \mathbf{while}\ (\mathsf{b})\,\{\,\mathsf{s}\,\} \mid \mathbf{skip}\\
\mathsf{rhs} & ::= & \mathsf{e} \mid \mathbf{new}\ \mathsf{C}\,[(\bar{\mathsf{e}})] \mid \mathsf{e}\ !\ \mathsf{m}(\bar{\mathsf{e}}) \mid \mathsf{x}.\mathbf{get}\\
e & ::= & \mathbf{null} \mid \mathsf{this.f} \mid \mathsf{x} \mid \mathsf{n} \mid \mathsf{e}+\mathsf{e} \mid \mathsf{e}*\mathsf{e} \mid \mathsf{e}-\mathsf{e}\\
\mathsf{g} & ::= & \mathsf{b} \mid \mathsf{e}? \mid \mathsf{g} \wedge \mathsf{g}
\end{array}
$$

The central concept of the concurrency model is that of *concurrent object*. Conceptually, each object has a dedicated processor and encapsulates a *local heap* which is not accessible from outside this object, i.e., fields are always accessed using the *this* object, and any other object can only access such fields through method calls. Concurrent objects live in a *distributed* environment with asynchronous and unordered communication by means of asynchronous method calls, denoted `o ! m(ē)`. Method calls may be seen as triggers of concurrent activity, spawning new tasks (so-called *processes*) in the called object. After asynchronously calling `x=o ! m(ē)`, the caller may proceed with its execution without blocking on the call. Here `x` is a *future variable* which allows synchronizing with the completion of task `m`. In particular, the instruction `await x?` allows checking whether `m` has finished. In this case, execution of the current task proceeds and `x` can be used for accessing the return value of `m` via the instruction `x.get`. Otherwise, the current task releases the processor to allow another available task to take it.

The translation of an ABS program into an equivalent CLP program has been subject of previous work [2]. An important feature of the translation is that the imperative program works on a global state which contains the set of created objects. This is simulated by representing the state using additional arguments of all predicates. Each object of the state includes the set of fields (which is not accessible outside the object) and its queue of pending tasks. Tasks can be of three types: *call* are asynchronous calls, *await* are tasks suspended due to an await condition and *get* are tasks suspended due to a blocking get instruction. Future variables become *ready(_)* when the corresponding task is completed. The syntax of the state is:

$$
\begin{array}{rcl}
\mathit{State} & ::= & [\,]\mid[(\mathit{Num},\mathit{Object})|\mathit{State}]\\
\mathit{Fut} & ::= & \mathit{ready}(\mathit{Data})|\,\mathit{Var}\\
Q & ::= & [\,]\mid[\mathit{Task}|Q]\\
\mathit{Fields} & ::= & [\,]\mid[\mathit{field}(f,\mathit{Data})|\mathit{Fields}]\\
\mathit{Object} & ::= & \mathit{object}(C,\mathit{Fields},Q)\\
\mathit{Task} & ::= & \mathit{call}(\mathit{Call})\mid\mathit{await}(\mathit{Call},\mathit{Call})\mid\mathit{get}(\mathit{Fut},\mathit{Var},\mathit{Call})
\end{array}
$$

Intuitively, for each class, the CLP translation represents all its methods (as well as the intermediate blocks within the methods for loops, conditionals, etc.) by means of predicates in the CLP program which adhere to the following grammar:

$$
\begin{array}{rcl}
\mathit{Clause} & ::= & \mathit{Pred}(\mathit{Args},\mathit{Args},S,S):-[\bar{G},]\bar{B}.\\
\mathit{Args} & ::= & [\,]\mid[\mathit{Data}^*|\mathit{Args}]\\
S & ::= & \mathit{Var}\\
G & ::= & \mathit{Num}^*\ \mathit{Op}_R\ \mathit{Num}^*\mid\mathit{Ref}_1^*\backslash{==}\mathit{Ref}_2^*\mid\mathit{Var}=\mathit{Data}\\
\mathit{Ref} & ::= & \mathit{null}\mid\mathit{Var}\\
B & ::= & \mathit{Var}\ \#{=}\ \mathit{Num}^*\ \mathit{Op}_A\ \mathit{Num}^*\mid\mathit{Pred}(\mathit{Args},\mathit{Args},S,S)\mid\mathit{Var}{=}\mathit{Data}\mid\\
 & & \mathsf{newObj}(C,\mathit{Ref}^*,S,S)\mid\mathsf{getField}(\mathit{Ref}^*,\mathit{FSig},\mathit{Var},S)\mid\mathsf{async}(\mathit{Ref}^*,\mathit{Call},S,S)\mid\\
 & & \mathsf{setField}(\mathit{Ref}^*,\mathit{FSig},\mathit{Var}^*,S,S)\mid\mathsf{await}(\mathit{Call},\mathit{Call},S,S)\mid\\
 & & \mathsf{get}(\mathit{Var},\mathit{Var},\mathit{Call},S,S)\mid\mathsf{return}(\mathit{Var}^*,\mathit{Var},S,S)\mid\mathsf{futAvail}(\mathit{Var},\mathit{Var})\\
\mathit{Call} & ::= & \mathit{Pred}(\mathit{Args},\mathit{Args})\\
\mathit{Pred} & ::= & \mathit{BlockN}\mid\mathit{MethodN}\\
\mathit{Data} & ::= & \mathit{Num}\mid\mathit{Ref}\mid\mathit{Bool}\\
\mathit{Op}_R & ::= & \#{>}\mid\#{<}\mid\#{>=}\mid\#{=<}\mid\#{=}\mid\#\backslash{=}\\
\mathit{Op}_A & ::= & +\mid-\mid*\mid/\mid\mathit{mod}
\end{array}
$$

Num is a number, *Var* is a Prolog variable and *Bool* can be either *true* or *false*. An asterisk on any element denotes that it can be either as defined by the grammar or a variable. Each clause receives as input a possibly empty list of parameters (1st argument) and a global

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

```
'A.sumFacts'([This, Ob], [R], S1, S2) :-
   getField(This, fSig('A', n), M, S1),
   await(awguard0([This, Ob], _),
         cont0([This, Ob, M], [R], S1, S2).
awguard0([This, Ob], [R], S, S) :-
   getField(This, fSig('A', ft), Ft, S),
   geq([Ft, 0], [R]).
cont0([This, Ob, M], [R], S1, S2) :-
   while([This, Ob, M, 0], [R], S1, S2).
while([Args], [R], S1, S2) :-
   M #=< 0,
   return([Res], [R], S1, S2).
while([Args], [R], S1, S2) :-
   M #> 0,
   getField(This, fSig('A', ft), Ft, S1),
   async(Ob, 'A.fact'([Ob, Ft, This], [F]), S1, S3),
   await(awguard1([F], _),
         cont1([Args, F], [R]), S3, S2).
awguard1([F], [V]) :- futAvail(F, V).
cont1([Args, F], [R], S1, S2) :-
   get(F, A, cont2([Args, A], [R]), S1, S2).
cont2([Args, A], [R], S1, S2) :-
   Res1 #= Res + A,
   getField(This, fSig('A', ft), Ft, S1),
   Ft1 #= Ft + 1,
   setField(This, fSig('A', ft), Ft1, S1, S3),
   M1 #= M − 1, while([Args1], [R], S3, S2).
geq([Ft, Z], [R]) :-
   Ft #< Z, R = false.
geq([Ft, Z], [R]) :-
   Ft #>= Z, R = true.
```

Figure 1 ABS running example (left). CLP translation of sumFacts (right).

state (3rd argument), and returns an output (2nd argument) and a final global state (4th argument). The body of a clause may include a sequence of guards followed by a sequence of instructions, including: arithmetic operations, calls to other predicates, builtins to handle the concurrency (namely await, get, futAvail and return) and builtins to operate on the heap [8]. The latter includes the builtin newObj(C, R, S_1, S_2) which creates a new object of class C in state S_1 and returns its assigned reference R and the updated state S_2; getField$(R, FSig, V, S)$ which retrieves in variable V the value of field $FSig$ of the object referenced by R in the state S and setField$(R, FSig, V, S_1, S_2)$ which sets the field $FSig$ of the object referenced by R in S_1 to V and returns S_2.

▶ **Example 1.** Fig. 1 (left) shows the implementation of a class A, which contains two integer fields and five methods. Method sumFacts computes $\sum_{k=ft}^{ft+(m-1)} k!$ by asynchronously invoking fact on object ob. The **await** instruction before entering the loop allows releasing the processor if ft is negative. Once it takes a non-negative value, the task can resume its

```
(a)async(Ref,Call,S1,S2) :- addTask(S1,Ref,call(Call),S2).
(b)await(Cond,Cont,S1,S3) :-
    Cond =..[_,[This|_],[Ret]], buildCall(Cond,S1,S2,CondCall), CondCall,
    (Ret = false -> addTask(S1,This,await(Cond,Cont),S2),
                    switchContext(S2,S3)
                    ; buildCall(Cont,S1,S3,ContCall), ContCall).
(c)get (FV,V,Cont,S1,S3) :- Cont =..[_,[This|_],_],
    (var(FV) -> addTask(S1,This,get(FV,V,Cont),S2),
                switchContext(S2,S3)
                ; FV = ready(V), buildCall(Cont,S1,S3,ContCall), ContCall).
(d)return([Ret],[ready(Ret)],S1,S2) :- switchContext(S1,S2).
(e)futAvail(FV,false) :- var(FV), !.           futAvail(ready(_),true).
------------------------------------------------------------------------------
(f)addTask(S1,Ref,T,S2) :- getCell(S1,Ref,object(C,Fs,Q1)),
    insert(Q1,T,Q2), setCell(S1,Ref,object(C,Fs,Q2),S2).
(g)switchContext(S1,S3) :- S1 = [(Ref,_)|_], firstToLast(S1,S2), switchContext_(S2,S3,Ref).
(h)switchContext_(S,S,Ref1) :- S = [(Ref2,object(_,_,[ ]))|_],Ref1 == Ref2.
(i)switchContext_(S1,S3,Ref1) :- \+ (S1 = [(Ref2,object(_,_,[ ])|_], Ref1 == Ref2),
    extractFirst(S1,Task,S2,Answer),
    runTaskOrSwitch(Answer,Task,Ref1,S3,S2).
(j)runTaskOrSwitch(true,Task,_Ref,S1,S3) :- runTask(Task,S1,S3).
(k)runTaskOrSwitch(false,_Task,Ref,S1,S3) :- firstToLast(S1,S2), switchContext_(S2,S3,Ref).
(l)runTask(call(ShortCall),S1,S2) :- buildCall(ShortCall,S1,S2,Call), Call.
(m)runTask(await(Cond,Cont),S1,S2) :- await(Cond,Cont,S1,S2).
(n)runTask(get(FV,V,Cont),S1,S2) :- get (FV,V,Cont,S1,S2).
(o)buildCall(ShortCall,S1,S2,Call) :- ShortCall =..[RN,In,Out], Call =..[RN,In,Out,S1,S2].
```

Figure 2 Implementation of Concurrency Builtins.

execution and enter the loop. Observe that an asynchronous call from sumFacts as follows f = ob ! fact(3, this); will add the task fact(3, this) to the queue of ob. When this task starts to execute it will add the task fact(2, ob) on the object this, which in turn will add the call fact(1, this) on ob and so on, in such a way that the factorial is computed in a distributed way between the two objects. Note that the calls are synchronized on future variables. This means that until the recursive call fact(1, this) is not completed the other tasks are suspended on their `await` conditions. Fig. 1 (right) shows the CLP translation of method sumFacts. We use $\overline{Args}$ and $\overline{Args_1}$ to abbreviate, resp., *This*, *Ob*, *M*, *Res* and *This*, *Ob*, M_1, Res_1. Methods and intermediate blocks (like $cont_0$) are uniformly represented by means of predicates and are not distinguishable in the translated program. The list of input arguments of all rules includes: the *this* reference, the list of input parameters of the corresponding ABS method, and for intermediate blocks, their local variables. The list of output argument is always a unitary list with the return value. Loops in the source program are transformed into guarded rules (e.g., rule *while*). An important point to note is that, for all `await` and `get` statements, we introduce a *continuation* predicate (like $cont_i$, $0 \leq i \leq 2$) which allows us to suspend the current task (if needed) and then resume its execution at this precise point.

2.2 Implementation of Concurrency Builtins

Fig. 2 shows the CLP implementation of the builtins to handle concurrency of [2]. Boxes are used to indicate code that needs to be changed in order to define the TDG framework. *Asynchronous calls* are handled by predicate ⓐ which adds the asynchronous call Call to the queue of tasks of the receiver object Ref producing the updated state S_2. The call to addTask/4 searches the state for the object pointed to by reference Ref by means of getCell/3 [8], adds the task to its queue (using insert/3) and updates the state with the updated object (using setCell/3 [8]).

The fact that objects do not share memory ensures that their execution states are not affected by how distribution (or parallelism) is realized. Namely, *distribution* is implemented as follows: each object executes its scheduled task as far as possible and, when a task finishes or gets blocked, simulation proceeds circularly with the *next* object in the state. In contrast, *concurrency* occurs at the level of objects in the sense that tasks in the object queue are executed concurrently. The concurrency model of our language only specifies that the execution of the current task must proceed until a call to ⓑ, ⓒ, or ⓓ is found. The scheduling policy which decides which task executes next (among those ready for execution) is left unspecified.

Rule ⓖ is used when the execution of the current task can no longer proceed (hence it *releases* the processor). The implementation gives the turn of execution to the first task (according to the scheduling policy) of the following object (the next one in the state). This is implemented by always keeping the current object in the head of the state, and moving it to the last position when its current task finishes or gets blocked. If the current object has some pending task in its queue ⓙ, predicate extractFirst/4 bounds Answer to true. Otherwise, it is bound to false and the following object is tried ⓚ. The execution of the whole application finishes when there is no pending task in any object ⓗ.

Await ⓑ first checks its condition Cond by means of the meta-call CondCall. If the condition holds (Ret gets instantiated to true), a meta-call to the continuation Cont is made (meta-call ContCall). Otherwise (Ret is false), an await task is added to the queue of the current object and we switch context. Predicate ⓞ builds a *full* call from a call without states and two states. The evaluation of await conditions can involve return tests on future variables. This is represented in our CLP programs by a call to ⓔ. We use the special term ready(V) to know whether the execution has finished. Rule ⓔ checks whether the future variable is a CLP variable or is instantiated to ready(_) and returns, resp., false or true. When a method finishes its execution, we reach a return statement ⓓ which instantiates the future variable V associated to the current task to ready(V). This allows that, if the task that requested the execution of this one was blocked awaiting on this future variable, it can proceed its execution when it is re-scheduled. Namely, ⓒ first checks if the task can resume execution because its future variable has become instantiated. In such case, the continuation of this get is executed (meta-call ContCall). Otherwise, the current task is added to the queue and context is switched.

3 From Symbolic Execution to TDG

Having a CLP symbolic execution engine for concurrent objects is an important piece when defining the CLP-based TDG framework, but there are still many other missing pieces. Firstly, we need to define a TDG engine which incorporates relevant *coverage criteria* (CC). An important problem in symbolic execution is that, since the input data is unknown, the execution tree to be traversed is in general infinite. Hence it is required to integrate a *termination criterion* which guarantees that the length of the paths traversed remains finite while at the same time an interesting set of test cases is generated, i.e., certain code *coverage* is achieved. The challenge when developing the TDG framework is integrating CC on the CLP-translated programs which achieve the desired degree of coverage on the original ABS.

3.1 Task-Level Coverage and Termination Criteria

Given a task executing on an object, we aim at ensuring its local termination by leveraging existing CC developed in the sequential setting to the context of concurrent objects. We focus on the loop-count criteria [10] which limits the number of times we iterate on loops to a threshold $\mathsf{K_l}$ (other existing criteria would pose similar problems and solutions). If we focus on a single task, this task-level CC can be integrated, as in the sequential CLP-based approach [8], by keeping track of the *ancestor sequences* for every call unfolded in the task. The main idea is that loop iterations are detected because recursive calls are performed. However, in order to distinguish a recursive call from an independent call to the same (recursive) predicate, we need to track the ancestors of each call. This can be implemented by using a global ancestor stack for the task such that each time an atom A is unfolded using a rule $H{:-}B_1, \ldots, B_n$, the predicate name of A (F/N where N is the arity) is pushed on the ancestor stack. Additionally, a $'\mathsf{\$pop\$}'$ mark is added to the new goal $B_1, \ldots, B_n, '\mathsf{\$pop\$}'$ to delimit the scope of the predecessors of F/N such that, once those atoms are evaluated, we find the mark $'\mathsf{\$pop\$}'$ and can remove F/N from the ancestor stack. This way, the ancestor stack, at each stage of the computation, contains the ancestors of the next atom to be selected for resolution.

Due to the coexistence of multiple tasks in the concurrent setting, the problem is more complicated and we need to construct the list of ancestor predicates for each available task and besides, as tasks can suspend their execution, be able to recover this information when they resume. Thus, the new syntax for tasks is:

$$Task ::= call(Call) \mid await(Call, Call, AncSt) \mid get(Fut, Var, Call, AncSt)$$

where $AncSt$ is a list of elements of the form F/N. Additionally, we introduce atoms of the form taskSuspendMark to indicate to the TDG engine that a task is going to suspend and hence its ancestor stack needs to be stored. This is achieved by replacing the framed code in ⓑ and ⓒ in Fig. 2, resp., by:

$(await)$ $\mathsf{taskSuspendMark(AncSt), addTask(S_1,This,await(Cond,Cont,AncSt),S_2),}$
(get) $\mathsf{taskSuspendMark(AncSt), addTask(S_1,This,get(FV,V,Cont,AncSt),S_2),}$

3.2 Task-Switching Coverage and Termination Criteria

Applying the task-level CC to all tasks does not guarantee termination. This is because we can switch from one task to another an infinite number of times. For example, consider the symbolic execution of $\mathsf{ob_1}$! $\mathsf{fact(n, ob_2)}$. We circularly switch from object $\mathsf{ob_1}$ to object $\mathsf{ob_2}$ an infinite number of times because each asynchronous call in one object adds another call on the other object (see Ex. 1). This is not detected by the task-level CC because each method invocation is a new task that has no ancestors. The same problem can happen even with a single object, e.g., in method sumFacts when executing `await` (ft >= 0), there is an infinite branch in the evaluation tree, corresponding to the case ft < 0 which is re-tried forever.

The number of task switches can be limited by simply allowing $\mathsf{K_s}$ executions of predicate (g) (Fig. 2). However, it might happen that, due to excessive task switching in certain objects, others are not properly tested (i.e., their tasks exercised) because the *global* number of allowed task switches has been exceeded. For example, suppose that we add the instructions $\mathsf{B}\ \mathsf{ob_2} = \mathsf{new}\ \mathsf{B}();\ \mathsf{ob_2}\ !\ \mathsf{q}();$ before the return in method sumFacts, where B is a class that implements method q but whose code is not relevant. Then, as the evaluation tree for the *while* loop generates an infinite number of task switches, the evaluation of the call $\mathsf{ob_2}\ !\ \mathsf{m}();$ is not reached. In order to have fairness in the process and guarantee proper coverage from the concurrency point of view, we propose to limit the number of task switches *per* object (i.e., per concurrency unit). For this purpose, objects are now of the form:

$$Object ::= object(C, Fields, Q, NT)$$

where NT is the number of tasks which have been extracted from its queue. Besides, similarly to the treatment of the task-level CC, we introduce special markers by replacing the framed code of rule (j) by:

$$\mathsf{taskStartMark(S_1,Task),\ incNumTasks(S_1,S_2),\ runTask(Task,S_2,S_3),}$$

which allows the TDG engine to realize that there has been a task switching and hence the limit needs to be checked. Predicate incNumTasks adds 1 to the number of task switches NT of the first object in $\mathsf{S_1}$, i.e., the object selected by extractFirst.

3.3 A CLP-based TDG Engine for Concurrent Objects

Fig. 3 presents a TDG engine, named unfold, which receives as input parameters the method call to be tested Root (last parameter), a list of atoms to be evaluated (initially Root), two constants $\mathsf{K_l}$ and $\mathsf{K_s}$ to limit, resp., the number of loop iterations and the number of task switches per object and the ancestor stack AncSt of the current task (initially empty). Rule (1) corresponds to the end of a successful derivation, it stores (using storeTestCase/1) the computed test case, namely the initial call Root instantiated with the bindings for the input/output parameters and the states, and the constraint store. The task-level CC is handled in rules (2), (4) and (7). Essentially, rule (7) checks if the number of iterations has not been exceeded (checkIter traverses the list of ancestors AncSt) and, if not, it adds the '\$pop\$' mark as explained in Sec. 3.1. Later, when this mark is reached in rule (2), the top of the stack is popped. In rule (4), when a task is suspending, the argument of taskSuspendMark gets unified with the current stack (fourth argument of unfold) to be later recovered, and execution proceeds. The treatment of the task switching criteria is captured by rule (3) which detects the mark introduced in Sec. 3.2 and invokes checkNtasks to check if the number of task switches in the current object exceeds $\mathsf{K_s}$. If the task to be started now is an await or get, predicate $\mathsf{recoverAncStack(Task, AncSt_p)}$ recovers its ancestor stack; if it is a call, initializes $\mathsf{AncSt_p}$ to empty. As we have seen in Sec. 3.2, after finding such mark, there is a call to incNumTasks.

The two remaining rules treat the builtins and the constraints. In particular, rule (6) handles the ABS builtin predicates in Fig. 2 which make callbacks to the program. They are treated differently from rule (7) because the loop-count criteria does not have to be applied on them. Rule (5) covers external predicates, i.e., constraints and the auxiliary predicates in Fig. 2. The difference w.r.t. rule (6) is that here we execute them (making call(A)) since the

```
① unfold([ ], _K_l, _K_s, _AncSt, Root) :- storeTestCase(Root).
② unfold(['$pop$'|R], K_l, K_s, [_|AncSt], Root) :- !, unfold(R, K_l, K_s, AncSt, Root).
③ unfold([taskStartMark(S, Task)|R], K_l, K_s, AncSt, Root) :- !,
       checkNtasks(S, K_s),
       recoverAncStack(Task, AncSt_p),
       unfold(R, K_l, K_s, AncSt_p, Root).
④ unfold([taskSuspendMark(AncSt)|R], K_l, K_s, AncSt, Rt) :- !,
       unfold(R, K_l, K_s, AncSt, Rt).
⑤ unfold([A|R], K_l, K_s, AncSt, Rt) :- isExternal(A), !,
       call(A), unfold(R, K_l, K_s, AncSt, Rt).
⑥ unfold([A|R], K_l, K_s, AncSt, Root) :- functor(A, F, Ar), isAbsBuiltin(F/Ar), !,
       clause(A, B), append(B, R, NG),
       unfold(NG, K_l, K_s, AncSt, Root).
⑦ unfold([A|R], K_l, K_s, AncSt, Root) :- functor(A, F, Ar), checkIter(AncSt, F, Ar, K_l),
       clause(A, B), append(B, ['$pop$'|R], NG),
       unfold(NG, K_l, K_s, [F/Ar|AncSt], Root).
checkIter([ ], _, _, K) :- K > 0.
checkIter([F/Ar|As], F, Ar, K) :- !, K > 1,K_1 is K−1,checkIter(As, F, Ar, K_1).
checkIter([_|As], F, Ar, K) : −checkIter(As, F, Ar, K).
checkNtasks([(_, object(_, _, _, NTs))|_], K) :- NTs < K.
incNumTasks(H, H_p) :- H = [(Ref, object(CN, Fs, Q, K))|RH],
       K_p is K + 1,
       H_p = [(Ref, object(CN, Fs, Q, K_p))|RH].
recoverAncStack(await(_, _, AncSt), AncSt).
recoverAncStack(get(_, _, _, AncSt), AncSt).
recoverAncStack(call(_), [ ]).
```

Figure 3 Implementation of TDG engine.

predicate is not part of the CLP program. The execution of unfold($[Root], K_l, K_s, [\,], Root$), where $Root{=}'C.m'(In, Out, S_1, S_2)$, computes an *incomplete* derivation tree for method m of class C, the different branches of the tree are obtained by backtracking. Successful branches are obtained in ① and incomplete branches by ③ and ⑦ when the termination tests stop the derivation. $\langle G\diamond\theta\rangle$ denotes a state with goal G and computed *constraint store* θ. Then, given the set of branches (derivations) for the derivation tree $\mathcal{T}$ associated to $\langle$unfold$([Root], K_l, K_s, [\,], Root)\diamond\{\}\rangle$, where $Root{=}'C.m'(In, Out, S_1, S_2)$, the *test cases* for m are the set of constraint stores θ associated to each output state $\langle\epsilon\diamond\theta\rangle$ of a successful branch in $\mathcal{T}$ (computed in ①), where ϵ is an empty goal.

▶ **Example 2.** Let us obtain the test cases for method sumFacts with K_l=1, K_s=2. The execution of unfold($[Root], 1, 2, [\,], Root$) with $Root{=}'A.sumFacts'(In, Out, S_1, S_2)$, first applies rule ⑦ which bounds In to the list $[This, Ob]$ and Out to $[R]$. Predicate checkIter succeeds (the ancestor stack is empty). The first instruction in $'A.sumFacts'$ (see Fig. 1 right) is a getField which bounds S_1 to $[(Id_1, object(A, [field(n,\_)|\_], [\,], 0))|\_]$. Afterward, we find a call to await which is handled by rule ⑥, which in turns executes the rule ⓑ of await in Fig. 2. Here, the condition $awguard_0$ in the await adds to the list of fields of S_1 the literal $field(ft, \_)$. The execution of the guard returns *false* and addTask inserts the task in the queue of object Id_1 with the annotation taskSuspendMark. Next, switchContext will take the task and annotate it with taskStartMark. Now, rule (3) is applied and checkNtasks fails since the number of task switches (incremented by switchContex) for Id_1 is greater than 2. By backtracking, we generate the branch in which $Ft \geq 0$ which leads to executing $cont_0$ and, after unfolding the first clause of predicate *while*, the first test case is computed. In this test case, $S_1/(Id_1, object('A', [field(n, N_1), field(ft, Ft_1)|\_], [\,], 0))|\_]$, $R/0$, the constraint store contains $N_1 \leq 0$, $Ft_1 \geq 0$, and $S_1 \equiv S_2$. Again, by backtracking the second clause for *while* is tried. At this point, the ancestor stack is $[cont_0/4, sumFacts/4]$. The async call introduces a new object $(Id_2, object(A, \_F, [call('A.fact')], 0))$ in the queue of Id_1. The execution of the await spawns the task $'A.fact'$ which returns 1. Thus, the second test case is computed $S_1/[(Id_1, object('A', [field(n, 1), field(ft, \ 0)|\_], \ [\,], 0)), (Id_2, object('A', \_, [\,], 0))|\_]$, $S_2/[(Id_1, object('A', [field(n, 1), field(ft, 1)|\_], [\ ], 1)), \ (Id_2, object('A', \_, [\ ], 1))|\_]$ and $R/1$. Note that the number of task switches for both objects Id_1 and Id_2 changes from 0 in the initial state S_1 to 1 in the final state S_2. No more solutions are computed since the execution of fact is stopped after two task switches coming from the await in its body and checkIter fails when evaluating again predicate *while* as the stack of ancestors contains already $[\ldots, while, \ldots]$. Therefore, the two criteria are needed to ensure termination: $\mathsf{K_s}$ to limit the number of task switches between the two objects and $\mathsf{K_l}$ to limit the number of loop iterations in the *while* loop.

4 Conclusions, Related and Future Work

We have presented a novel approach to automate test case generation for concurrent objects, entirely implemented in CLP, which ensures *completeness* of the test cases w.r.t. several interesting criteria. The coverage criteria prune the tree in several dimensions: (1) limiting the number of iterations of loops at the level of tasks, (2) limiting the length of the queue of tasks of the objects such that the number of task interleavings that are tried remains finite, (3) limiting the number of task switches allowed in each concurrency unit. The technique is complete on the orderings in which tasks can be selected for execution, even allowing that different policies are applied on different objects. We argue that our CLP-based framework is at the same time practical and highly flexible and constitutes thus a promising approach to TDG of concurrent languages.

In future work, we plan to study the application of our framework to a thread-based concurrency model like Java [13, 5, 18]. The main conceptual difference with the actor-based model is that task scheduling is preemptive. Therefore, at any point, the current task can be suspended and interleaved with another one. Specific coverage criteria should be defined to control such interleavings in a way that the size of the symbolic execution tree remains reasonable and at the same time interesting test cases can be obtained. It seems that the combination with dynamic analysis is useful for this purpose [3]. We also want to investigate the application of further coverage criteria [14, 18, 7] to detect bugs related, for instance, to happen-before relations.

References

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

2 E. Albert, P. Arenas, and M. Gómez-Zamalloa. Symbolic Execution of Concurrent Objects in CLP. In *Practical Aspects of Declarative Languages (PADL'12)*, volume 7149 of *LNCS*, pages 123–137. Springer, January 2012.

3 Jun Chen and Steve MacDonald. Towards a better Collaboration of Static and Dynamic Analyses for Testing Concurrent Programs. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD'08)*, page 8. ACM, 2008.

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

5 O. Edelstein, E. Farchi, E. Goldin, Y. Nir, Ratsaby G, and S. Ur. Framework for Testing Multi-Threaded Java Programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

6 Christian Engel and Reiner Hähnle. Generating Unit Tests from Formal Proofs. In *Tests and Proofs, First International Conference (TAP'07)*, volume 4454 of *LNCS*, pages 169–188. Springer, 2007.

7 M. Factor, E. Farchi, and Y. Malka Y. Lichtenstein. Testing Concurrent Programs: A Formal Evaluation of Coverage Criteria. In *Seventh Israeli Conference on Computer-Based Systems and Software Engineering (ICCSSE '96)*, pages 119–126, 1996.

8 M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *Theory and Practice of Logic Programming, ICLP'10 Special Issue*, 10 (4–6), 2010.

9 A. Gotlieb, B. Botella, and M. Rueher. A CLP Framework for Computing Structural Test Data. In *Computational Logic*, pages 399–413, 2000.

10 W.E. Howden. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.

11 E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Formal Methods for Components and Objects (FMCO 2010, Revised Papers)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.

12 J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

13 B. Long, D. Hoffman, and P. A. Strooper. Tool Support for Testing Concurrent Java Components. *IEEE Trans. Software Eng.*, 29(6):555–566, 2003.

14 Shan Lu, Weihang Jiang, and Yuanyuan Zhou. A Study of Interleaving Coverage Criteria. In *ESEC/SIGSOFT FSE*, pages 533–536. ACM, 2007.

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

16 R. A. Müller, C. Lembeck, and H. Kuchen. A Symbolic Java Virtual Machine for Test Case Generation. In *IASTED Conf. on Software Engineering*. IASTED/ACTA Press, 2004.

17 T. Schrijvers, F. Degrave, and W. Vanhoof. Towards a Framework for Constraint-Based Test Case Generation. In *International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'09)*, volume 6037 of *LNCS*, pages 128–142. Springer, 2010.

18 Juichi Takahashi, Hideharu Kojima, and Zengo Furukawa. Coverage Based Testing for Concurrent Software. In *IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2008)*, pages 533–538. IEEE Computer Society, 2008.