Symbolic Execution of Concurrent Objects in CLP

Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa

DSIC, Complutense University of Madrid, Spain

Abstract. In the *concurrent objects* model, objects have conceptually dedicated processors and live in a distributed environment with unordered communication by means of asynchronous method calls. Method callers may decide at runtime when to synchronize with the reply from a call. This paper presents a CLP-based approach to *symbolic execution* of concurrent OO programs. Developing a symbolic execution engine for concurrent objects is challenging because it needs to combine the OO features of the language, concurrency and backtracking. Our approach consists in, first, transforming the OO program into an equivalent CLP program which contains calls to specific builtins that handle the concurrency model. The builtins are implemented in CLP and include primitives to handle asynchronous calls synchronization operations and scheduling policies, among others. Interestingly, symbolic execution of the transformed programs then relies simply on the standard sequential execution of CLP. We report on a prototype implementation within the PET system which shows the feasibility of our approach.

1 Introduction

Increasing performance demands, application complexity and multi-core parallelism make distribution and concurrency omnipresent in today's software applications. There is thus a renewed interest in investigating techniques that help in simulating, debugging, testing, verifying, etc., distributed and concurrent programs. The focus of this paper is on developing a CLP-based framework for the symbolic execution of concurrent *object-oriented* (OO) imperative programs. Symbolic execution of a program consists in executing it "a la Prolog", i.e., using as arguments free (logic) variables. It allows thus reasoning about all the inputs that take the same path through the program. Symbolic execution is at the core of software verification [14] and testing tools [15, 18, 23]. In the latter case, by incorporating coverage and termination criteria, symbolic execution allows automatically obtaining test-inputs ensuring a certain degree of code coverage.

Within the OO paradigm, there are two main approaches to concurrency: (1) thread-based concurrency models (like those of Java and C#) are based on threads which share memory and are scheduled preemptively, i.e., they can be suspended or activated at any time. To prevent threads from undesired interleavings, low-level synchronization mechanisms such as locks have to be used.

Experience has shown that software written in the thread-based model is error-prone, difficult to debug, verify and maintain [20]. (2) In order to overcome these problems, the *active-objects* model [6,13,17,20,21] aims at providing programmers with simple language extensions which allow programming concurrent applications with relatively little effort. Active (also called concurrent) objects operate similar to Actors [1] and Erlang processes [5].

In this paper, we consider the imperative OO language ABS [12] which is based on the active-objects concurrency model. A concurrent object, conceptually, has a dedicated processor and it encapsulates a local heap which is not accessible from outside the object. The language supports asynchronous method calls, which trigger activities in other objects without transferring control from the caller. The method caller may decide at runtime when to synchronize with the reply from a call. In general, an object may have many method activations competing to be executed. Among these, at most one process (or *task*) is active and the other processes are suspended in a process pool. Process scheduling is non-deterministic, but controlled by processor release points in a *cooperative* way. Cooperative scheduling means that switching between tasks of the same object happens only at specific scheduling points during program execution, which are explicit in the source code and can be syntactically identified.

The goal of this paper is to design (and implement) a *CLP-based symbolic execution* engine for concurrent ABS programs. This is a challenging problem as one needs to combine the OO and concurrent aspects of the ABS language with the backtracking mechanism required to perform symbolic execution. For *sequential* programs, we have seen in [7–9, 16] that, as symbolic execution is the standard evaluation mechanism of CLP, symbolic execution of imperative programs can be performed in a natural and efficient way by: (1) first, translating the imperative program into an equivalent CLP program and, (2) then, relying on the execution mechanism of CLP which performs symbolic execution natively.

The main contribution of this paper is to lift such CLP-based framework from the sequential to the concurrent OO setting. In particular, we first propose an automatic transformation of concurrent imperative programs into CLP programs which include specific builtin operations to handle the concurrency aspects of the language. The global state is made explicit in the translation as an additional argument of clauses. It includes the set of concurrent objects with their fields values and corresponding queues of pending tasks. We then provide an implementation in CLP of the builtins to treat all concurrency aspects of the language: (a) *asynchronous* calls are handled by adding corresponding pending tasks to the queues of the remote objects on which the calls are performed, (b) *synchronization* operations can be performed to suspend the execution of a task in an object until certain condition holds, (c) *future variables* become part of the state and allow synchronizing with the reply from a call, and (d) different *scheduling policies* can be easily integrated in our symbolic execution engine. We report on a prototype implementation of our proposal within the PET system [8] (a generic platform for CLP-based testing) and evaluate it on a series of small applications which are classical examples of concurrent programming.

$$
\begin{array}{rl|rl}
T & ::= B \mid I \mid D \mid D\langle \bar{T} \rangle & A & ::= N \mid T \mid D\langle \bar{A} \rangle \\
Dd & ::= \texttt{data}\ D[\langle \bar{A} \rangle] = Cons[\mid Cons] & Cons & ::= Co[(\bar{A})] \\
F & ::= \texttt{def}\ A\ fn[\langle \bar{A} \rangle](\overline{A\ x}) = e & p & ::= x \mid t \mid Co[(\bar{p})] \\
e & ::= b \mid x \mid t \mid \texttt{this} \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \texttt{case}\ e\ \{\overline{p \Rightarrow e}\} & t & ::= Co[(\bar{t})] \mid \texttt{null} \\
\hline
IF & ::= \texttt{interface}\ I\ [\texttt{extends}\ \overline{I}]\ \{\ \overline{Sg}\ \} & Sg & ::= T\ m\ (\overline{T\ x}) \\
CL & ::= \texttt{class}\ C\ [(\overline{T\ x})]\ [\texttt{implements}\ \overline{I}]\ \{\ \overline{T\ x};\ \overline{M}\} & M & ::= Sg\ \{\overline{T\ x};\ s\ \} \\
s & ::= s\ ;\ s \mid x = rhs \mid \texttt{await}\ g \mid \texttt{return}\ e & g & ::= b \mid e? \mid g \wedge g \\
 & \quad \mid \texttt{if}\ (b)\ \{\ s\ \}\ [\texttt{else}\ \{\ s\ \}] \mid \texttt{while}\ (b)\ \{\ s\ \} \mid \texttt{skip} & & \\
rhs & ::= e \mid \texttt{new}\ C[(\overline{e})] \mid e\ !\ m(\overline{e}) \mid e.m(\overline{e}) \mid x.\texttt{get} & &
\end{array}
$$

Fig. 1. ABS Syntax for Functional (top) and Concurrent Object Level (bottom)

2 An Overview of Concurrent Objects

Our method is presented for the core of the ABS language [12], a successor of Creol [6, 13]. ABS is an OO language for distributed concurrent systems whose concurrency model is based on *concurrent objects*. An ABS *program* defines interfaces, classes, datatypes, and functions, and has a `main` block to configure the initial state. The *functional sub-language* allows abstracting from implementation details: abstract data types are used to specify internal, sequential computations, while concurrency is handled in the imperative part.

Fig. 1 gives the syntax of ABS programs. In the functional level (top), ground types T consist of basic types B (Bool, Int, etc.), names for interfaces I and data types D. In contrast to T, types A may contain type variables named N. Dd stands for data type declarations, where D has at least one constructor $Cons$. Function declarations F consist of a return type A, a function name fn, a list of variable declarations $\overline{x}$ of types $\overline{A}$, and an *expression* e. Expressions e include Boolean expressions b, variables x, (ground) terms t, the special read-only variable **this** which refers to the identifier of the object, constructor expressions of the form $Co[(\overline{e})]$, function applications of the form $fn(\overline{e})$, and case expressions of the form $\texttt{case}\ e\{\overline{p \Rightarrow e}\}$, where p is a *pattern*, as defined in the grammar.

In the *concurrent object level of* ABS (bottom), an interface IF has a name I and method signatures Sg, and it can extend other interfaces $\overline{I}$. A class has a name C, implements a list of interfaces, may contain class parameters and state variables $\overline{x}$ of type $\overline{T}$, and methods $\overline{M}$. The *fields* of the class are both its parameters and state variables. Objects are instances of classes; their declared fields are initialized to arbitrary type-correct values. A method signature Sg declares the return type T of a method m and formal parameters $\overline{x}$ of types $\overline{T}$. M defines a method with signature Sg, a list of local variable declarations $\overline{x}$ of types $\overline{T}$, and a statement s. All methods return a value (Unit plays the role of void in sequential programming). Statements may access fields of the *current class*, locally defined variables, and the method's formal parameters. Right hand side expressions rhs include object creation, method calls, and expressions e. Statements

```
data List⟨A⟩=Nil | Cons(A,List⟨A⟩);
data Set⟨A⟩=EmptyS | Insert(A,Set⟨A⟩);
data Pairs⟨A,B⟩=Pair(A,B);
data Map⟨A,B⟩=EmptyM |
       Assoc(Pairs⟨A,B⟩,Map⟨A,B⟩);
type FN, Packet=String;
type FNs=Set⟨String⟩;
type File=List⟨Packet⟩;
type Catalog=List⟨Pairs⟨Node,FNs⟩⟩;
def B lookup⟨A,B⟩(Map⟨A,B⟩ ms, A k)=
  case ms {Assoc(Pair(k,y),_) ⇒ y;
           Assoc(_,tm) ⇒ lookup(tm,k);}
```

```
def Bool contains⟨A⟩(Set⟨A⟩ s,A e)=
  case s {
    EmptyS ⇒ False;
    Insert(e, _) ⇒ True;
    Insert(_, xs) ⇒ contains(xs, e);}
def Node findServer(FN f,Catalog c)=
  case c {
    Nil ⇒ null;
    Cons(Pair(s, fs), r) ⇒
      case contains(fs, f) {
        True ⇒ s;
        False ⇒ findServer(f, r); };}
```

Fig. 2. (Fragment of) Functional Sequential Part of ABS P2P Network

are standard for assignment $x = rhs$, sequential composition s_1 ; s_2, **skip**, **if**, **while**, and **return** constructs. In **await** g, the guard g controls processor release and consists of Boolean conditions b, return tests $x?$ and conjunctions. If g evaluates to false, the processor is released, the current process is *suspended* and the processor becomes idle. When the processor is idle, any enabled process from the object's pool of suspended processes may be scheduled.

Example 1. Our running example is a peer-to-peer (P2P) distributed application borrowed from [13]. Fig. 2 shows a fragment of the functional program which includes type definitions (*String* and *Int* are predefined) and three functions which are executed using strict evaluation. Fig. 3 shows the most relevant part of the imperative concurrent program (interfaces and the implementation of class Network are not shown). Calls to functions and functional data appear in italics. Function *nth* returns the n-th element of a list and *appr* concatenates two lists. A P2P network is formed by a set of interconnected peers which can act as clients and servers. Peers make the files stored in their database (an object of type DB) available to other peers, without central coordination. The only coordination is by means of an object of class Network. It is enough to know that nodes learn who their neighbors are by invoking getNeighbors implemented in this class. A node acting as client triggers computations with searchFile, which first finds a neighbor node s that can provide the file and then requests the file using reqFile.

Communication in ABS is based on asynchronous method calls, denoted o ! $m(\overline{e})$, and future variables ($Fut\langle\cdot\rangle$). Method calls may be seen as triggers of concurrent activity, spawning new tasks (so-called *processes*) in the called object. After asynchronously calling $x{=}o$! $m(\overline{e})$, the caller may proceed with its execution without blocking on the call. Here x is a future variable, o is an object (typed by an interface), and $\overline{e}$ are expressions. A future variable x refers to a return value which has yet to be computed. There are two operations on future variables, which control external synchronization in ABS. First, a return test $x?$ evaluates to false unless the reply to the call can be retrieved. Second, the return value is retrieved by the expression x.**get**, which blocks

```
class DBImp(Map⟨FN,File⟩ db)
implements DB {
File getFile(FN fId) {
  return lookup(db, fId);}
Int getLength(FN fId) {
  return length(lookup(db,fId));}
Unit storeFile(FN fId, File file) {
  db=Assoc(Pair(fId,file), db);}
FNs listFiles() {
  return keys(db);}
}
class Node(DB db,FN file)
implements Peer {
Catalog cat=Nil;
List⟨Peer⟩ myN=Nil;
Network admin=null;
Unit run() {
  Fut⟨Catalog⟩ c; Fut⟨List⟨Peer⟩⟩ f;
  Server server ;
  await admin != null;
  f=admin ! getNeighbors(this);
  await f?; myN=f.get;
  c=this ! availFiles(myN);
  await c?; cat=c.get;
  server=findServer(file, cat);
  if (server != null) {
    this.reqFile(server,file);}}
Unit setAdmin(Network admin) {
  this.admin=admin;}
FNs enquire() {
  Fut⟨FNs⟩ f; f=db ! listFiles();
  await f?; return f.get;}
Int getLength(FN fId) {
  Fut⟨Int⟩ lth; lth=db ! getLength(fId);
  await lth?; return lth.get;}
Packet getPack(FN fId, Int pNbr) {
  File f=Nil; Fut⟨File⟩ ff;
  ff=db ! getFile(fId);
  await ff?; f=ff.get;
  return nth(f, pNbr);}
Catalog availFiles (List⟨Peer⟩ sL) {
  Catalog cat=Nil; FNs fNs=EmptyS;
  Fut⟨FNs⟩ fN; Catalog catL=Nil;
  Fut⟨Catalog⟩ cL;
  if (sL != Nil) {
    fN=head(sL) ! enquire();
    cL=this ! availFiles(tail(sL));
    await fN? & cL?;
    catL=cL.get; fNs=fN.get;
    cat=appr(catL,Pair(head(sL),fNs));
  }
  return cat;}
Unit reqFile(Server sId, FN fId) {
  Fut⟨Int⟩ l1; Fut⟨Packet⟩ l2;
  l1=sId ! getLength(fId);
  await l1?; Int lth=l1.get;
  while (lth > 0) {
    lth=lth - 1;
    l2=sId ! getPack(fId, lth);
    await l2?; Packet pack=l2.get ;
    file=Cons(pack, file);}
  db ! storeFile(fId, file);}
}
```

Fig. 3. Concurrent Part of ABS Implementation of P2P Network

all execution in the object until the return value is available. A *synchronous call*, abbreviated as $v{=}o.m(\overline{e})$, is internally transformed into the statement sequence $x{=}o$! $m(\overline{e})$; $if(o{==}this)$ **await** $x?$; $v{=}x$.**get**. Observe that checking if $o{==}this$ is necessary to avoid that the execution of the current object blocks when a synchronous local call is performed.

Example 2. The following fragment of code corresponds to a possible `main` method for the P2P example.

$Map\langle FN, File\rangle$ dataBase = $Assoc(Pair(\text{"}file_0\text{"}, Cons(\text{"}a\text{"}, Cons(\text{"}b\text{"}, Cons(\text{"}c\text{"}, Nil)))),$
$Assoc(Pair(\text{"}file_1\text{"}, Cons(\text{"}d\text{"}, Cons(\text{"}e\text{"}, Nil))), EmptyM))$;
DB db_1 = **new** DBImp($EmptyM$); DB db_2 = **new** DBImp(dataBase);
Peer n_1 = **new** Node(db_1,"$file_0$"); Peer n_2 = **new** Node(db_1,"$file_1$");
Peer n_3 = **new** Node(db_2,"$file_1$"); NetWork admin = **new** NetWork($\mathrm{n}_1, \mathrm{n}_2, \mathrm{n}_3$);

n_1 ! setAdmin(admin); n_2 ! setAdmin(admin); n_3 ! setAdmin(admin);
n_1 ! run(); n_2 ! run();

The network configuration consists of three nodes, two databases and one Network object (admin). Nodes n_1 and n_2 are neighbors of n_3. Such six objects become distinct concurrent entities which communicate with each other by means of asynchronous calls and use *future* variables to eventually return/retrieve the results. Any concurrent object has its own heap, its queue of pending tasks and an active task (if any).

3 CLP-Translated Programs

The translation of *sequential* imperative programs into equivalent CLP programs has been subject of previous work (see, e.g., [3, 7]). Intuitively, for each method (or function), the translation represents the method (or function) as well as the intermediate blocks within the method (e.g., loops, conditionals) by means of predicates in the CLP program. The fact that the imperative program works on a global state is simulated by representing the state using additional arguments of all predicates. We will not go into details of how the transformation of the sequential part is formalized (see [3, 7]). Instead, we focus on the syntactic extensions of the ABS translated concurrent programs.

3.1 Syntax of CLP-Translated Programs

An ABS *CLP-translated program* is made up of a set of predicates, each of them defined by one or more *mutually exclusive clauses*, which adhere to the following grammar:

$$
\begin{array}{rl}
\mathit{Clause} ::= & \mathit{Pred}(\mathit{Args}, \mathit{Args}, S, S) : -[\bar{G},]\bar{B}. \\
G ::= & \mathit{Num}^* \ \mathit{Op}_R \ \mathit{Num}^* \mid \mathit{Ref}_1^* \backslash == \mathit{Ref}_2^* \mid \mathit{Var} = \mathit{FTerm}^* \mid \\
& \mathit{diff}(\mathit{Var}, \mathit{FTerm}^*) \mid \mathsf{type}(S, \mathit{Ref}^*, C) \\
B ::= & \mathit{Var} \ \#= \ \mathit{Num}^* \ \mathit{Op}_A \ \mathit{Num}^* \mid \mathit{Pred}(\mathit{Args}, \mathit{Args}, S, S) \mid \mathit{Var}{=}\mathit{FTerm} \mid \\
& \mathsf{new}(C, \mathit{Ref}^*, S, S) \mid \mathsf{getField}(\mathit{Ref}^*, \mathit{FSig}, \mathit{Var}, S) \mid \mathsf{async}(\mathit{Ref}^*, \mathit{Call}, S, S) \mid \\
& \mathsf{setField}(\mathit{Ref}^*, \mathit{FSig}, \mathit{Var}^*, S, S) \mid \mathsf{await}(\mathit{Call}, \mathit{Call}, S, S) \mid \\
& \mathsf{get}(\mathit{Var}, \mathit{Var}, \mathit{Call}, S, S) \mid \mathsf{return}(\mathit{Var}^*, \mathit{Var}, S, S) \mid \mathsf{futAvail}(\mathit{Var}, \mathit{Var})
\end{array}
$$

$$
\begin{array}{rlrl}
\mathit{Call} ::= & \mathit{Pred}(\mathit{Args}, \mathit{Args}) & \mathit{Ref} ::= & \mathit{null} \mid \mathit{Var} \\
\mathit{Pred} ::= & \mathit{BlockN} \mid \mathit{MethodN} \mid \mathit{FuncN} & \mathit{Op}_R ::= & \#> \mid \#< \mid \#>= \mid \#=< \mid \#= \mid \#\backslash = \\
\mathit{Args} ::= & [\,] \mid [\mathit{Data}^* | \mathit{Args}] & \mathit{Op}_A ::= & + \mid - \mid * \mid / \mid \mathit{mod} \\
\mathit{Data} ::= & \mathit{Num} \mid \mathit{Ref} \mid \mathit{FTerm} & S ::= & \mathit{Var}
\end{array}
$$

We use *FuncN*, *MethodN*, *FSig* to denote the set of functions names, methods and field signatures. Clauses can define methods and functions which appear in the original source program (*MethodN*, *FuncN*) and additional predicates which correspond to intermediate blocks in the program (*BlockN*). *Num* is a number, *Var* is a Prolog variable and *FTerm* is a term that represents a corresponding functional data (namely p in Fig. 1). 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 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 create objects and to write and read on object fields, and builtins to handle the concurrency.

We use three different kinds of inequalities in guards, namely, "\==", "=" and *diff* to represent, resp., arithmetic comparisons, comparisons of references and pattern matchings in ABS functions. Virtual method invocations in the OO language are resolved at compile-time and translated into a choice of type builtins followed by the corresponding method invocation for each runtime instance. As expected, the builtin new(C, R, S_1, S_2) creates a new object of class C in state S_1 and returns its assigned reference R and the updated state S_2; getField$(R, FSig, V, S)$ retrieves in variable V the value of field $FSig$ of the object referenced by R in the state S; setField$(R, FSig, V, S_1, S_2)$ sets the field $FSig$ of the object referenced by R in S_1 to V and returns the modified state S_2.

In the translation of concurrent programs, when a concurrency construct appears (namely an asynchronous call, an **await** or **get** statement), we introduce a call to a corresponding builtin predicate that will simulate the concurrent behaviour. Besides, an important point to notice is that, for all **await** and **get** statements, we introduce a *continuation* predicate which allows us to suspend the current task (if needed) and then be able to resume its execution at this precise point. Also, we introduce in the translation *return* statements in order to syntactically identify in the CLP-translated program when the execution of a task finishes and thus another task from the queue can be scheduled.

Example 3. The following code shows the CLP-translated program for method reqFile of class Node.

$$
\begin{array}{l}
\text{'}Node.reqFile\text{'}([This, SId, FId], [Out], S_1, S_2) \text{ :-} \\
\quad async(SId,\text{'}Node.getLength\text{'}([SId, FId], [L_1]), S_1, S_3), \\
\quad await(awguard_1([L_1], [\_]), cont_1([This, SId, FId, L_1], [Out]), S_3, S_2). \\
awguard_1([L_1], [V]) \text{ :- } futAvail(L_1, V). \\
cont_1([This, SId, FId, L_1], [Out], S_1, S_2) \text{ :-} \\
\quad get(L_1, Lth, cont_2([This, SId, FId, Lth], [Out]), S_1, S_2). \\
cont_2([This, SId, FId, Lth], [Out], S_1, S_2) \text{ :- } File = \text{'}Nil\text{'}, \\
\quad while([This, SId, FId, File, Lth], [Out], S_1, S_2). \\
while([This, SId, FId, File, Lth], [Out], S_1, S_2) \text{ :- } \# <= (Lth, 0), \\
\quad getField(This,\text{'}Node.db\text{'}, Db, S_1), \\
\quad async(Db,\text{'}DBImp.storeFile\text{'}([Db, FId, File], [\_], S_1, S_3), \\
\quad return([\text{'}Unit\text{'}], [Out], S_3, S_2). \\
while([This, SId, FId, File, Lth], [Out], S_1, S_2) \text{ :- } \# > (Lth, 0), \# = (Lth_1, Lth - 1), \\
\quad async(SId,\text{'}Node.getPack\text{'}([SId, FId, Lth_1], [L_2]), S_1, S_3), \\
\quad await(awguard_2([L_2], \_), cont_3([This, SId, FId, File, L_2, Lth_1], [Out]), S_3, S_2). \\
awguard_2([L_2], [V]) \text{ :- } futAvail(L_2, V). \\
cont_3([This, SId, FId, File, L_2, Lth], [Out], S_1, S_2) \text{ :-} \\
\quad get(L_2, Pack, cont_4([This, SId, FId, File, Pack, Lth], [Out], S_1, S_2). \\
cont_4([This, SId, FId, File, Pack, Lth], [Out], S_1, S_2) \text{ :- } File_1 = \text{'}Cons\text{'}(Pack, File), \\
\quad while([This, SId, FId, File_1, Lth], [Out], S_1, S_2).
\end{array}
$$

The main features that can be observed from the translation are: **(1)** Methods (like reqFile), intermediate blocks (like $cont_1$) and functions are uniformly represented by means of predicates and are not distinguishable in the translated program. The input arguments list of all rules includes: the *this* reference, the list of input parameters of the ABS method from which the rule originates, and, in the case of predicates corresponding to intermediate blocks, their local variables. The output arguments list is always a unitary list with the return value. **(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*. **(3)** Additional rules are produced for the continuations after await and get statements. The calls to such continuation rules are included within the arguments of the await and get builtins (see e.g. rules '*Node.reqFile*' for the case of await or $cont_1$ for get). This allows the symbolic execution engine to suspend the execution at this point and resume it later. **(4)** A global state is explicitly handled. Observe that each rule includes as arguments an input and an output state. The state is carried along the execution being used and transformed by the corresponding builtins as a black box, therefore it is always a variable in the CLP program.

3.2 The Global State

In a sequential OO language, the global state carried along by the CLP-translated program only contains the data stored in the heap. Instead, in our concurrent setting, it has to include the set of existing concurrent objects, each of them with its associated internal state. The internal state of an object includes two pieces of information: (1) its heap (set of fields) which is not accessible from outside the object and (2) the queue of pending tasks. Formally, the syntax of the global state is as follows:

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

The state is represented as a list of pairs, where *Num* is a unique reference to the object *Object*. Each object is a term which includes its class C, a list of fields *Fields* and a queue Q of pending tasks. Each element in *Fields* is a term containing a field name and its associated data. The meaning of the different kinds of tasks *Task* and the syntax of future variables *Fut* is related to the symbolic execution of the translated programs and will be explained in detail in the next section.

Example 4. Consider an execution of the main method in Ex. 2 which starts from an initial state [ ]. After creating the objects of type DBImp, the state takes the form $[o_{db_1}, o_{db_2}]$, where o_{db_1}=$(1, object('DBImp', [field(db,'EmptyM')], [\,]))$ and o_{db_2}=$(2, object('DBImp', [field(db, dataBase)], [\,]))$. Here, *1* and *2* are the references for db_1 and db_2, respectively. Similarly, the next three new instructions add three new elements to the state, resulting in $[o_{db_1}, o_{db_2}, o_{n_1}, o_{n_2}, o_{n_3}]$, where:

```
async(Ref,Call,S1,S2) :- addTask(S1,Ref,call(Call),S2).
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).
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).
return([Ret],[ready(Ret)],S1,S2) :- switchContext(S1,S2).
futAvail(FV,'False') :- var(FV), !.
futAvail(ready(_),'True').
```

```
addTask(S1,Ref,T,S2) :- getCell(S1,Ref,object(C,Fs,Q1)),
                        insert(Q1,T,Q2), setCell(S1,Ref,object(C,Fs,Q2),S2).
switchContext(S1,S3) :- S1 = [(Ref,_)|_], firstToLast(S1,S2),
                        switchContext_(S2,S3,Ref).
switchContext_(S,S,Ref1) :- S = [(Ref2,object(_,_,[ ]))|_], Ref1 == Ref2, !.
switchContext_(S1,S3,Ref) :-
    (extractTask(S1,Task,S2) -> runTask(Task,S2,S3)
                                firstToLast(S1,S2), switchContext_(S2,S3,Ref)).
runTask(call(ShortCall),S1,S2) :- buildCall(ShortCall,S1,S2,Call), Call.
runTask(await(Cond,Cont),S1,S2) :- await(Cond,Cont,S1,S2).
runTask(get(FV,V,Cont),S1,S2) :- get(FV,V,Cont,S1,S2).
buildCall(ShortCall,S1,S2,Call) :- ShortCall =..[RN,In,Out], Call =..[RN,In,Out,S1,S2].
```

Fig. 4. Implementation of Concurrency builtins

$$o_{n_1} = (3, object('Node', [\, field(db, 1), field(file, "file_0"), field(cat, 'Nil'), \\ field(myN, 'Nil'), field(admin, null)], [\,]))$$

and o_{n_2}, o_{n_3} are similar to o_{n_1} except for the object identifiers (4 and 5 respectively) and the value of field *file* (which is "$file_1$" in both objects). Field db has value *1* for o_{n_2}, and value 2 for o_{n_3}.

4 Symbolic Execution of Concurrent Objects

In dynamic (or concrete) execution, the initial state must be a ground term (e.g., if execution starts from a `main`, it is an empty list). Objects must be created using new/4 before their fields can be read or written. In symbolic execution, the intuitive idea proposed in [8] is that the state contains two parts: the known part (beginning of the list) with the objects that have been explicitly created during symbolic execution, and the unknown part which is a logic variable (tail of the list) in which new data can be added by producing the corresponding bindings. Therefore, the state starts being a free variable, and the implementation of getField/4 and setField/5 invokes predicates getCell/3 and setCell/4 which, if the object whose fields are going to be read or written is not in the known part,

they instantiate the unknown part of the heap to be able to assume the previous allocation of the object and access its fields. Figure 4 shows the CLP implementation of the builtins to handle concurrency. They rely on the above getCell/3 and setCell/4 operations (whose implementation is in [8]) to symbolically access the heap. The following sections explain the behavior of the different builtins.

4.1 Asynchronous Calls

Predicate async(Ref,Call,S_1,S_2), given the current state S_1 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, adds the task to its queue and updates the state with the updated object. As explained above, if the object pointed to by Ref is not in the known part of the state, getCell/3 produces a corresponding instantiation on the unknown part so that after this operation the object is in the state.

Example 5. Let us consider the symbolic execution of method reqFile, i.e., we run in CLP the goal $'Node.reqFile'(In, Out, S_0, S_1)$. After the first call to async/4 the following instantiations are produced:

S_0=[$(SId, object('Node', [field('Node.db', DB), \ldots]), [\,])$]
S_1=[$(SId, object('Node', [field('Node.db', DB), \ldots]), [call('Node.getLength'(\ldots))])$]

Observe that, as expected, asynchronous calls do not transfer control from the caller, i.e., they are not executed when they occur but rather added as pending tasks on the receiver objects that will eventually schedule them for execution.

4.2 Implementation of Distribution and Concurrency

The fact that objects do not share memory ensures that their execution states (and thus the global state) are not affected by how distribution is realized. Therefore, symbolic execution can simulate distribution in any convenient way. We implement it in the following specific way: 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 (which could be running in parallel in an actual deployment configuration). In contrast, *concurrency* occurs at the level of object in the sense that tasks in the object queue are executed concurrently. Cooperative scheduling of the ABS language only specifies that the execution of the current task must proceed until a call to return/4, await/4 or get/5 is found. The scheduling policy which decides the task that executes next (among those ready for execution) is left unspecified.

Predicate switchContext/2 is used when the execution of the current task can no longer proceed. It 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, as it

can be observed in the implementation of switchContext/2. If the current object has some pending task in its queue, the task is run (calling runTask/3). Otherwise (predicate extractTask/3 fails), the following object is tried. The execution of the whole application finishes when there is no pending task in any object (see first rule of switchContext_/3). Observe that there are three different types of tasks, call, await and get, whose behaviour is explained below.

One can implement different scheduling policies by providing concrete implementations of predicates insert/3 and extractTask/3. For instance, a FIFO scheduling policy is implemented by 1) inserting at the end of the queue, and 2) extracting always the first task. One can also use priority queues. The implementation becomes parametric on the scheduling policy by just asserting the selected policy and adding a parameter to predicates insert and extractTask to apply the selected policy. Furthermore, the language allows that different objects apply different scheduling policies. Thus, one can also select the desired policy per object. In this case, when scheduling a new task, we first read the asserted information which indicates the scheduling policy at the object level and, then, invoke the appropriate implementation of insert and extractTask for the current object. Having parametric scheduling policies is interesting in the application of symbolic execution to regression testing, as one then wants to save the selected policy within the test-cases in order to be able to replay them.

4.3 Synchronization: future variables, await, get and return

Await. Predicate await(Cond,Cont,S_1,S_3) 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 involved object and we switch context. Let us observe that the calls wrapped within asyncs, awaits and gets as well as those stored in object queues, do not include states but just input and output arguments (see grammars in Sect. 3). This is because when a task is to be executed the current state must be used (and not the one that was current when the task was first created). Predicate buildCall/4 builds a *full* call from a call without states and the two states involved.

Future variables. The evaluation of await conditions can involve return tests on future variables. This is represented in our CLP programs by a call to the futAvail/2 builtin. Future variables occur in the global state in the output arguments of call tasks, and are available when they get instantiated. Since, in the context of symbolic execution, the return value of a method can be a variable V, we use the special term ready(V) to know whether the execution has finished (see the global state grammar in Sect. 3.2). Predicate futAvail/2 then just has to check whether the future variable is a CLP variable or is instantiated to ready(_) and returns, resp., 'False' or 'True'.

Example 6. Let us continue with the symbolic execution of method reqFile right after the execution of the first async (see Ex. 5). The call to await first produces a call to $awguard_1$ which checks whether the return value L_1 (future

Benchmark	D=50			D=75			D=100		
	#I	#S	T	#I	#S	T	#I	#S	T
ProducerImpl.loop	1175	29	30	8028	134	140	35291	437	630
ConsumerImpl.loop	35	2	10	159	4	20	254	5	20
BoundedBuffer.append	2751	77	10	10494	198	30	24840	360	40
DistHT.lookupNode	319	11	20	697	17	10	1219	23	10
DistHT.getAllData	6	1	10	1406	21	40	9466	111	130
DistHT.getAllKeysAux	96	3	10	849	14	60	15622	173	360
DistHT.getAllKeys	22	1	11	160	3	30	1177	14	119
DistHT.putData	2220	50	10	14608	242	30	47532	612	70
DBImp.getLength	9108	253	61	30940	595	160	78208	1128	359
Node.run	0	0	10	51241	720	240	14219536	148466	45640
Node.getLength	3731	91	40	20475	351	150	55081	741	360
Node.getPack	1736	42	20	9919	169	40	26961	361	60
Node.reqFile	0	0	10	1988	28	110	16530	190	390
SessionImp.order	0	0	30	0	0	110	5647	59	320
AgentImp.free	616	22	10	1435	35	10	2491	47	10
DBImp.confirmOrder	95568	2167	599	4863238	71277	21230	-	-	-

Table 1. Statistics about the Analysis Process

variable) of the call to *getLength* is already available (by means of the call to futAvail/2). Since it is not the case (i.e, a '*False*' is returned) the execution of the current task cannot proceed, therefore the await task is added to the current object (so that it is re-tried later on) and context is switched (see the calls to addTask/4 and switchContext/2). This, in turn, produces a call to runTask(call('Node.getLength'(...)),S_2,S_3) where the current state is now

S_2=[(SId, $object$('$Node$', [$field$('$Node.db$', DB_1), ...]), [ ]),
($This$, $object$('$Node$', [$field$('$Node.db$', DB_2), ...]), [$await$($awguard_1$(...), $cont_1$(...))])]

Return. 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.

Get. Predicate get first checks if the task can resume execution because the future variable that is blocking it has become instantiated. In such case, the continuation of the get is executed (meta-call ContCall). Otherwise, the current task is added to the queue and context is switched.

5 Experimental Results in aPET

PET [8] is a test-case generation tool which aims at being a generic platform for CLP-based test-case generation of different languages. This work implements

the core part of aPET, an extension of PET to generate test-cases from concurrent ABS programs. Currently, we have implemented the automatic translation of ABS programs into CLP equivalent programs and extended the symbolic execution engine of PET with the concurrency primitives of ABS described along the paper. Experimental evaluation has been carried out using several typical concurrent applications: BBuffer, a classical bounded-buffer for communicating several producers and consumers, DistHT which implements a distributed hash-table, PeerToPeer, our running example; BookShop, which implements a web shop client-server application. The code of the examples can be found in `http://costa.ls.fi.upm.es/pet/apet`.

Table 1 summarizes our experiments. Each set of rows contains the results of symbolically executing methods which belong to the above benchmarks. Symbolic execution for all methods works properly but, in the table, we have only showed the results for the methods which have more complex code and whose symbolic execution takes longer. As methods contain loops or recursion, symbolic execution does not terminate unless we introduce some termination criteria. In our case, we limit the length of the branches of the symbolic execution tree to a constant **D** (i.e., the depth of the tree to **D**). For each experiment, we show three sets of columns with the results of setting **D** to 50, 75 and 100 steps. Then, column #**I** shows the total number of instructions that have been executed including all branches, #**S** shows the number of solutions (branches) in the resulting symbolic execution tree, and **T** the total time (in milliseconds) required to build the tree. Experiments have been performed on an Intel Core i5 at 3.2GHz with 3.1GB of RAM, running Linux. All times have been computed as the average of 5 runs. When time is negligible, the system gives **T** = **10**. As expected, when allowing larger values for the depth of the tree, the number of branches grows exponentially and thus the total time. This is not a problem related to our approach, but rather inherent to symbolic execution. Methods Node.run and DBImp.confirmOrder have larger times (and number of instructions) because the size of the code reachable from them is much larger (they contain many calls to other methods). For the last one, no result is computed in a reasonable time for **D**=**100**. In order to alleviate this problem, testing tools often limit the number of iterations on loops to a small number. Otherwise, the process can become quite expensive and too many test-cases can be obtained, as it can be observed from the large number of solutions obtained.

6 Conclusions and Related Work

We have presented the first CLP-based approach to symbolic execution of concurrent objects. The main idea is that concurrent distributed imperative programs can be translated into equivalent CLP programs which contain calls to builtin operations that simulate the concurrent behavior of the active objects paradigm. A unique feature of our approach is that, as the builtin operations can be fully implemented in logic programming, symbolic execution boils down to standard sequential execution of the CLP transformed program.

Process scheduling in concurrent objects has some similarities with the *dynamic scheduling* available in Prolog systems. However, the behavior is not the same and it cannot be directly used. This is because synchronization using dynamic scheduling can resume the execution of a task as soon as the await condition is satisfied, while cooperative scheduling only allows switching between tasks at specific scheduling points. As concurrent objects do not share memory, one could think of using Prolog's parallelism [11] to simulate the distributed execution by running each object as a parallel task. However, there is no support to simulate the fact that one object receives requests from another one by means of asynchronous calls. Some systems, like SWI, implement parallelism using threads with associated queues and synchronization is achieved by means of asserted variables. Indeed, for concrete execution, we have a working implementation using SWI Prolog parallelism in which tasks communicate by means of global variables (asserted in Prolog's database). However, the use of impure features does not allow the backtracking required in symbolic execution. Recent years are witnessing a wealth of research in testing concurrent programs. Symbolic execution is the central part of most static test-case generation tools, which typically obtain the test-cases from the branches of the symbolic execution tree. There is previous related work on using Creol for modeling and testing systems against specifications [2], though the problem of symbolic execution is not studied there. Later, [10] studies dynamic symbolic execution of Creol programs which combines concrete and symbolic execution. A fundamental difference with our approach is that they use an interpreter of Creol to perform symbolic execution, while in our case, we transform the ABS program into an equivalent CLP which does not require any interpretation layer, rather it is executed natively in CLP. Simulation tools for ABS programs that perform concrete execution [4] are only tangentially related to our work. This is because dynamic execution does not require backtracking and hence the use of CLP has less interest.

Recent work on testing thread-based languages studies ways to improve scalability [19] which could also be adapted to our context. Likewise, [22] proposes new coverage criteria in the context of concurrent languages that could be studied in our CLP-based setting. As future work, we plan to integrate our symbolic execution mechanism within a test-case generation tool in order to generate unit tests for ABS programs in a fully automatic way.

Acknowledgments. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, the UCM-BSCH-GR35/10-A-910502 *GPD* Research Group, the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project and by the Spanish Ministry of Science (MICINN) under the TIN-2008-05624 *DOVES* project.

References

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

2. B. K. Aichernig, A. Griesmayer, R. Schlatte, and A. Stam. Modeling and Testing Multi-Threaded Asynchronous Systems with Creol. *ENTCS*, 243:3–14, 2009.
3. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.
4. E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Simulating Concurrent Behaviors with Worst-Case Cost Bounds. In *Proc. of FM'2011*, vol. 6664 of *LNCS*, pages 353–368. Springer, 2011.
5. J. Armstrong, R. Virding, C. Wistrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.
6. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.
7. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *JIST*, 51:1409–1427, 2009.
8. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *TPLP, ICLP'10 Special Issue*, 2010.
9. A. Gotlieb, B. Botella, and M. Rueher. A CLP Framework for Computing Structural Test Data. In *Computational Logic*, 2000.
10. A. Griesmayer, B. K. Aichernig, E. B. Johnsen, and R. Schlatte. Dynamic Symbolic Execution of Distributed Concurrent Objects. In *Proc. of FMOODS/FORTE'2009*, volume 5522 of *LNCS*, pages 225–230. Springer, 2009.
11. G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM TOPLAS*, 23(4):472–602, July 2001.
12. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. of FMCO'2010*, *LNCS*. Springer, 2011. To appear.
13. E. B. Johnsen and O. Owe. An Asynchronous Communication Model for Distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, 2007.
14. S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proc of TACAS*, pages 553–568, 2003.
15. J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.
16. C. Meudec. Atgen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.
17. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1997.
18. Roger A. Müller, Christoph Lembeck, and Herbert Kuchen. A Symbolic Java Virtual Machine for Test Case Generation. In *IASTED Conf. on Software Engineering*, pages 365–371, 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. J. Schäfer and A. Poetzsch-Heffter. Jcobox: Generalizing Active Objects to Concurrent Components. In *Proc. of ECOOP'10*, volume 6183 of *LNCS*, pages 275–299. Springer, 2010.
21. S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proc. of ECOOP'08*, volume 5142 of *LNCS*, pages 104–128. Springer, 2008.
22. J. Takahashi, H. Kojima, and Z. Furukawa. Coverage based Testing for Concurrent Software. In *ICDCS Workshops*, pages 533–538. IEEE Computer Society, 2008.
23. Nikolai Tillmann and Jonathan de Halleux. Pex-white Box Test Generation for .NET. In *TAP*, pages 134–153, 2008.