Recursive Abstractions for Parameterized Systems

JOXAN JAFFAR and ANDREW E. SANTOSA

Department of Computer Science, National University of Singapore
Singapore 117590
{joxan,andrews}comp.nus.edu.sg

Abstract. We consider a language of recursively defined formulas about arrays of variables, suitable for specifying safety properties of parameterized systems. We then present an abstract interpretation framework which translates a paramerized system as a symbolic transition system which propagates such formulas as abstractions of underlying concrete states. The main contribution is a proof method for implications between the formulas, which then provides for an implementation of this abstract interpreter.

1 Introduction

Automation of verification of parameterized systems are an active area of research [1–7]. One essential challenge is to reason about the unbounded parameter n representing the number of processes in the system. This usually entails the provision of an induction hypothesis, a step that is often limited to manual intervention. This challenge adds to the standard one when the domain of discourse of the processes are infinite-state.

In this paper, we present an abstract interpretation [8] approach for the verification of infinite-state parameterized systems.

First, we present a language for the general specification of properties of arrays of variables, each of whom has length equal to the parameter n. The expressive power of this language stems from its ability to specify complex properties on these arrays. In particular, these complex properties are just those that arise from a language which allows *recursive definitions* of properties of interest.

Second, we present a symbolic transition framework for obtaining a symbolic execution tree which (a) is finite, and (b) represents all possible concrete traces of the system. This is achieved, as in standard abstract interpretation, by computing a symbolic execution tree but using a process of abstraction on the symbolic states so that the total number of abstract states encountered is bounded. Verification of a particular (safety) property of the system is then obtained simply by inspection of the tree.

Third, the key step therefore is to compute two things: (a) given an abstract state and a transition of the parameterized system, compute the new abstract state, and (b) determine if a computed abstract state is subsumed by the previously computed abstract states (so that no further action is required on this state). The main contribution of this paper is an algorithm to determine both.

Consider a driving example of a parameterized system of $n \geq 2$ process where each process simply increments the value of shared variable x (see Figure 1 (left)). The idea is to prove, given an initial state where $x = 0$, that $x = n$ at termination.

Figure 1 (right) outlines the steps in generating the symbolic execution tree for this example. The tree is constructed by letting each process proceed from an initial program point $\langle 0 \rangle$ to its final point $\langle 1 \rangle$.

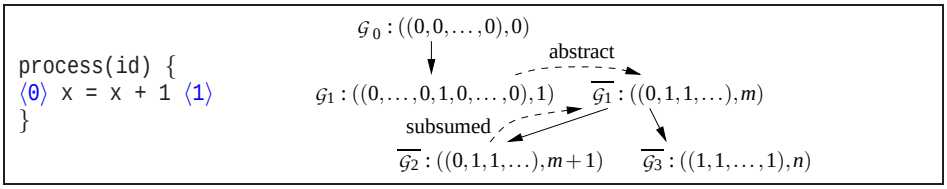

Fig. 1: Abstract Computation Tree of Counting Ones

We start with a program state of $\mathcal{G}_0 = ((0,0,\cdots,0),0)$ where the first element is a sequence of n bits representing the program counter, and the second element represents the value of x. A first transition would bring the system to a state $\mathcal{G}_1 = ((0,\cdots,0,1,0,\cdots,0),1)$, where the position of the "1" is anywhere in the range 1 to n, and the value of x is now 1. At this point, we would like to abstract this state to a state $\overline{\mathcal{G}_1}$ where the counter has, not exactly one numeral 1, but some $1 \leq m < n$ copies of the numeral 1. Further, the value of x is not 1, but instead is equal to m. Let us denote this state $((0,1,1,\cdots),m)$.

There are now two possible transitions: first, corresponding to the case where $\overline{\mathcal{G}_1}$ has at least two 0's, we get a new state $\overline{\mathcal{G}_2}$ whose counter has a mixture of 0's and 1's. But this new state is already covered by $\overline{\mathcal{G}_1}$ and hence need not be considered further. The second case is where $\overline{\mathcal{G}_1}$ has exactly one 0, in which the final transition results in the final state $\overline{\mathcal{G}_3} = ((1,1,\cdots,1),x)$ where the counter has all 1's. Since the value of x in $\overline{\mathcal{G}_3}$ equals the number of 1's, it follows that $x = n$ in this final state.

The key points in this proof are as follow. First, we employed the notion of an abstract state $\overline{\mathcal{G}_1}$ where the counter has $1 \leq m < n$ copies of 1 (the rest are 0), and $x = m$. We then show that the concrete state $\mathcal{G}_1$ emanating from the initial state is in fact an instance of $\overline{\mathcal{G}_1}$. We then showed that the state $\overline{\mathcal{G}_2}$ emanating from $\overline{\mathcal{G}_1}$ is either (a) itself $\overline{\mathcal{G}_1}$ (which therefore requires no further consideration), or (b) the final state $\overline{\mathcal{G}_3} : ((1,1,\cdots,1),x)$, and where $x = n$. Thus the proof that $x = n$ at the end is established.

The main result in this paper, in terms of this example, is first to construct the computation tree, but more importantly to provide an automatic proof of the conditions that make the tree a true representation of all the traces of the underlying parameterized system. In our example, our algorithm proves the entailments $\mathcal{G}_1 \models \overline{\mathcal{G}_1}$ and $\overline{\mathcal{G}_2} \models \overline{\mathcal{G}_1}$. Although not exemplified, all states in discussed here are written in our constraint language using arrays and recursive definitions, which is to be discussed in Section 2. For instance, the state $\mathcal{G}_0$ is represented using n-element array of zeroes which is defined using a recursive definition. We provide an algorithm to prove entailments in verification conditions which involve integer arrays and the recursive definitions.

In summary, our contributions are threefold:

- We present a language for defining recursive abstractions consisting of recursive definitions and integer arrays. Such abstractions are to be used to represent core properties of the parameterized system that are invariant over the parameter n of the system. The provision of these abstractions is generally restricted to be manual.
- Then we provide a symbolic traversal mechanism to construct a symbolic execution tree which exhibits the behavior of the parameterized system, which is exemplified in Figure 1 (left). In constructing the tree we abstract the states encountered using the recursive abstractions. In the above example, this is exemplified with the abstraction of $\mathcal{G}_1$ to $\overline{\mathcal{G}_1}$. Our objective is to produce a closed tree, where all the paths

in the tree reaches the end of the program's execution (the case of $\overline{\mathcal{G}_3}$ above) or ends in a state that is subsumed by some other state in the tree (the case of $\overline{\mathcal{G}_2}$, which is subsumed by $\overline{\mathcal{G}_1}$).

Now, there are two kinds of proofs needed: one is for the correctness of the abstraction step (represented as the *entailment* $\mathcal{G}_1 \models \overline{\mathcal{G}_1}$ of two formulas). Similarly, we need a proof of entailment of formulas defining the subsumption of one state over another (eg. $\overline{\mathcal{G}_2} \models \overline{\mathcal{G}_1}$ above).

- Finally we devise a proof method where the recursive definitions and the arrays work together in the entailment proof. In this way, the *only* manual intervention required is to provide the abstraction of a state (in our example, the provision of the abstraction $\overline{\mathcal{G}_1}$ to abstract $\mathcal{G}_1$). Dispensing with this kind of manual intervention is, in general, clearly as challenging as discovering loop invariants in regular programs. However, it is essentially dependent on knowledge about the *algorithm* underpinning the system, and not about the *proof system* itself.

1.1 Related Work

Central to the present paper is the prior work [9] which presented a general method for the proof of (entailment between) recursively defined predicates. This method is a proof reduction strategy augmented with a principle of *coinduction*, the primary means to obtain a terminating proof. In the present paper, the earlier work is extended first by a symbolic transition system which models the behavior of the underlying parameterized system. A more important extension is the consideration of array formulas. These array formulas are particularly useful for specifying abstract properties of states of a parameterized systems.

Recent work by [3] concerns a class of formulas, environment predicates, in a way that systems can be abstracted into a finite number of such formulas. The essence of the formula is a universally quantified expression relating the local variable of a reference process to all other processes. For example, a formula of the form $\forall j \neq i : x[i] < x[j]$ could be used to state that the local variable x of the reference process i is less than the corresponding variable in *all other* processes. A separate method is used to ensure that the relationships inside the quantification fall into a finite set eg. predicate abstraction. An important advantage of these works is the possibility of automatically deriving the abstract formulas from a system.

The *indexed predicates* method [4] is somewhat similar to environment predicates in that the formula describes universally quantified statements over indices which range over all processes. Determining which indexed predicates are appropriate is however not completely automatic. Further, these methods are not accompanied by an abstract transition relation.

The paper [1] presents safety verification technique of parameterized systems using abstraction and constraints. Key ideas include the handling of existentially and universally-quantified transition guards), and the use of *gap-order constraints*. Abstraction is done by weakening the gap-order constraints.

Our method differs from the above three works because we present a general language for the specification of *any* abstraction, and not a restricted class. We further provide a transition relation which can work with the abstraction language in order to generate lemmas sufficient for a correctness proof. The proof method, while not decidable, is general and can dispense with a large class of applications.

Earlier work on counter abstraction [7] clearly is relevant to our abstractions which is centrally concerned with describing abstract properties of program counters. Later works on *invisible invariants* [6] show that by proving properties of systems with a fixed (and small) parameter, that the properties indeed hold when the parameter is not restricted. In both these classes of works, however, the system is assumed to be finite state.

There are some other works using inductive, as opposed to abstraction, methods for example [5]. While these methods address a large class of formulas, they often depend on significant manual intervention.

We finally mention the work of [2] which, in one aspect, is closest in philosophy to our work. The main idea is to represent both the system and the property (including liveness properties) as *logic programs*. In this sense, they are using recursive definitions as we do. The main method involves proving a predicate by a process of folding/unfolding of the logic programs until the proof is obvious from the syntactic structure of the resulting programs. They do not consider array formulas or abstract interpretation.

2 The Language

In this section we provide a short description of constraint language allowed by the underlying constraint solver assumed in all our examples.

2.1 Basic Constraints

We first consider *basic constraints* which are constructed from two kinds of terms: integer terms and *array expressions*. Integer terms are constructed in the usual way, with one addition: the array element. The latter is defined recursively to be of the form $a[i]$ where a is an array expression and i an integer term. An array expression is either an array variable or of the form $\langle a,i,j\rangle$ where a is an array expression and i,j are integer terms.

The meaning of an array expression is simply a map from integers into integers, and the meaning of an array expression $a' = \langle a,i,j\rangle$ is a map just like a except that $a'[i] = j$. The meaning of array elements is governed by the classic McCarthy [10] axioms:

$i = k \;\rightarrow \langle a,i,j\rangle[k] = j$
$i \neq k \;\rightarrow \langle a,i,j\rangle[k] = a[k]$

A basic constraint is either an integer equality or inequality, or an equation between array expressions. The meaning of a constraint is defined in the obvious way.

In what follows, we use constraint to mean either an atomic constraint or a conjunction of constraints. We shall use the symbol ψ or Ψ, with or without subscripts, to denote a constraint.

2.2 Recursive Constraints

We now formalize *recursive constraints* using the framework of Constraint Logic Programming (CLP) [11]. To keep this paper self-contained, we now provide a brief background on CLP.

An *atom* is of the form $p(\tilde{t})$ where p is a user-defined predicate symbol and $\tilde{t}$ a tuple of terms, written in the language of an underlying constraint solver. A *rule* is of the form $A :\!\text{-}\, \Psi, \tilde{B}$ where the atom A is the *head* of the rule, and the sequence of atoms

```
sys(N,K,X) :- 1 ≤ Id ≤ N, K[Id] = 0, K′=<K,Id,1>, X′=X+1, sys(N,K′,X′).
```

Fig. 2: Transitions of Counting Ones

$\tilde{B}$ and constraint Ψ constitute the *body* of the rule. The body of the rule represents a conjunction of the atoms and constraints within. The constraint Ψ is also written in the language of the underlying constraint solver, which is assumed to be able to decide (at least reasonably frequently) whether Ψ is satisfiable or not. A rule represents implication with the body as antecedent and the head as the conclusion. A *program* is a finite set of rules, which represents a conjunction of those rules. The semantics of a program is the smallest set of (variable-free) atoms that satisfy the program. Given a CLP program, recursive constraints are constructed using recursive predicates defined in the program.

Example 1 (Count Ones). The following program formalizes the states described in the "counting ones" example (note that _ denotes "any" value). In the predicates below, the number `N` represents the parameter, the array `K` represents the counter, and `X` represents the shared variable. `Allzeroes(N,K,X)` holds for any `N`, `K`, and `X` when `K` is an array of length `N` with all elements zero and `X` is zero. `Allones(N,K,X)` holds when all elements of `K` are one, and `X=N`. Finally, the meaning of `abs(N, K, M)` is that `K` is a bit vector and `M` is the number of 1's in `K`.

```
allzeroes(0, _, 0).
allzeroes(N, ⟨K,N,0⟩, 0) :- N > 0, allzeroes(N-1, K, 0).
allones(0, _, 0).
allones(N, ⟨K,N,1⟩, N) :- N > 0, allones(N-1, K, N-1).
bit(0).
bit(1).
abs(0, _, 0).
abs(N, ⟨K,N,B⟩, M+B) :- N > 0, bit(B), abs(N-1, K, M).
```

3 Formalization of a Parameterized System

We now formalize a parameterized system as a transition system. We assume interleaving execution of the concurrent processes, where a transition that is executed by a process is considered atomic, that is, no other process can observe the system state when another process is executing a transition. Similar to the definition of recursive constraints in the previous section, the transition systems here are also defined using CLP, where a CLP rule models a state transition of the system.

3.1 Abstract Computation Trees

Before proceeding, we require a few more definitions on CLP. A *substitution* θ simultaneously replaces each variable in a term or constraint e into some expression, and we write $e\theta$ to denote the result. We sometimes write θ more specificly as $[e_1/t_1, \ldots, e_n/t_n]$ to denote substitution of t_i by e_i for $1 \leq i \leq n$. A *renaming* is a substitution which maps each variable in the expression into a variable distinct from other variables. A *grounding* is a substitution which maps each integer or array variable into its intended universe of discourse: an integer or an array. Where Ψ is a constraint, a grounding of Ψ results in *true* or *false* in the usual way.

A *grounding* θ of an atom $p(\tilde{t})$ is an object of the form $p(\tilde{t}\theta)$ having no variables. A grounding of a goal $\mathcal{G} \equiv (p(\tilde{t}), \Psi)$ is a grounding θ of $p(\tilde{t})$ where $\Psi\theta$ is *true*. We write $[\![\mathcal{G}]\!]$ to denote the set of groundings of $\mathcal{G}$. We say that a goal $\mathcal{G}$ *entails* another goal $\mathcal{G}'$, written $\mathcal{G} \models \mathcal{G}'$, if $[\![\mathcal{G}]\!] \subseteq [\![\mathcal{G}']\!]$.

From now on we speak about *goals* which have exactly the same format as the body of a rule. A goal that contains only constraints and no atoms is called *final*.

Let $\mathcal{G} \equiv (B_1, \cdots, B_n, \Psi)$ and P denote a goal and program respectively. Let $R \equiv A$:- $\Psi_1, C_1, \cdots, C_m$ denote a rule in P, written so that none of its variables appear in $\mathcal{G}$. Let the equation $A = B$ be shorthand for the pairwise equation of the corresponding arguments of A and B. A *reduct* of $\mathcal{G}$ using a rule R which head matches an atom B_i in $\mathcal{G}$, denoted $\text{REDUCT}_{B_i}(\mathcal{G}, R)$, is of the form

$(B_1, \cdots, B_{i-1}, C_1, \cdots, C_m, B_{i+1}, \cdots, B_n, (B_i = A), \Psi, \Psi_1)$

provided the constraint $(B_i = A) \wedge \Psi \wedge \Psi_1$ is satisfiable.

Definition 1 (Unfold). *Given a program P and a goal $\mathcal{G}$, $\text{UNFOLD}_B(\mathcal{G})$ is $\{\mathcal{G}' | \exists R \in P : \mathcal{G}' = \text{REDUCT}_B(\mathcal{G}, R)\}$.* □

A *derivation sequence* for a goal $\mathcal{G}_0$ is a possibly infinite sequence of goals $\mathcal{G}_0$, $\mathcal{G}_1$, $\cdots$ where $\mathcal{G}_i$, $i > 0$ is a reduct of $\mathcal{G}_{i-1}$. If the last goal $\mathcal{G}_n$ is a final (hence no rule R of the program can be applied to generate a reduct of $\mathcal{G}_n$), we say that the derivation is *successful*. Since a goal can be unfolded to a number of other goals (reducts), we can identify the *derivation tree* of a goal.

Definition 2 (Abstract Computation Tree). *An* abstract computation tree *is defined just like a derivation tree with one exception: the use of a derivation step may produce not the reduct goal $\mathcal{G}$ as originally defined, but a* generalization $\overline{\mathcal{G}}$ *of this reduct goal. Whenever such a generalization is performed in the tree construction, we say that an* abstraction step *is performed on $\mathcal{G}$ obtaining $\overline{\mathcal{G}}$.* □

Our concern in this paper is primarily to compute an abstract computation tree which represents all the concrete traces of the underlying parameterized system. The following property of abstract trees ensures this.

Definition 3 (Closure). *An abstract computation tree is* closed *if each leaf node represents a goal $\mathcal{G}$ which is either terminal, ie. no transition is possible from $\mathcal{G}$, or which is entailed by a goal labelling another node in the tree.* □

3.2 Symbolic Transitions

Next we describe how to represent a parameterized system as a CLP program. In doing so, we inherit a framework of abstract computation trees of parameterized systems. More specifically, the safety property that we seek can then be obtained by inspection of a closed abstract computation tree that we can generate from the system.

We start with a predicate of the form **sys**(N, K, T, X) where the number N represents the parameter, the N-element array K represents the program counter, the N-element array T represents each *local* variable of each process, and finally, X represents a shared/global variable. Multiple copies of T and/or X may be used as appropriate.

We then write symbolic transitions of a parameterized systems using the following general format:

```
sys(N, K, T, X) :- K[Id] = α, K′ = <K, Id, β>,
    Ψ(N, K, T, X, K′, T′, X′), sys(N, K′, T′, X′).
```

This describes a transition from a program point α to a point β in a process. The variable Id symbolically represents a (nondeterministic) choice of which process is being executed. We call such variables *index* variables. The formula Ψ denotes a (basic or recursive) constraint relating the current values K,T,X and future values K',T',X' of the key variables.

Consider again the example of Figure 1, and consider its transition system in Figure 2. The transition system consists of transitions from program counter $\langle 0 \rangle$ to $\langle 1 \rangle$ of a parameterized system, where each process simply increments its local variable X and terminates[1]. The system terminates when the program counter contains only 1's, ie. when all processes are at point $\langle 1 \rangle$.

3.3 The Top-Level Verification Process

In this section we outline the verification process. The process starts with a goal representing the initial state of the system. Reduction and abstraction steps are then successively applied to the goal resulting in a number of verification conditions (obligations), which are proved using our proof method.

We now exemplify using the Counting Ones example of Section 1. This goal representing the initial state is $\mathcal{G}_0$ in Figure 1. Recall that we formalize the transitions of the Counting Ones example in Figure 2. In our formalization, we represent the goal $\mathcal{G}_0$ as follows $\mathbf{sys}(N,K,X), allzeroes(N,K,X)$ denoting a state where all the elements of the array K are zero.

We apply the transition of Figure 2 by reducing $\mathcal{G}_0$ into the goal $\mathcal{G}_1$, which in our formalism is the goal $\mathbf{sys}(N,K',X'), allzeroes(N,K,X), 1 \leq Id_1 \leq N, K[Id_1] = 0, K' = \langle K, Id_1, 1\rangle, X' = X+1$. The goal represents a state where only one of the elements of the array K is set to 1. Note that this reduction step is akin to strongest postcondition propagation [12] since given the precondition $allzeroes(N,K,X)$, the postcondition is exactly $(\exists K,X,Id_1 : allzeroes(N,K,X), 1 \leq Id_1 \leq N, K[Id_1] = 0, K' = \langle K, Id_1, 1\rangle, X' = X+1)[K/K', X/X']$.

We now abstract the goal $\mathcal{G}_1$ into $\overline{\mathcal{G}_1}$, which in our formalism is represented as $\mathbf{sys}(N,K',X'), abs(N,K',X')$. Here one verification condition in the form of an entailment is generated:

$$allzeroes(N,K,X), 1 \leq Id_1 \leq N, K[Id_1] = 0, K' = \langle K, Id_1, 1\rangle, X' = X+1 \models abs(N,K',X').$$

The proof this obligation guarantess that the abstraction is an over approximation.

Now, the propagation from $\overline{\mathcal{G}_1}$ to $\overline{\mathcal{G}_2}$ is again done by applying unfold (reduction) to the predicate **sys** based on its definition (Figure 2). As the result, we obtain the goal $\overline{\mathcal{G}_2}$ as follows:

$$\mathbf{sys}(N,K'',X''), abs(N,K',X'), 1 \leq Id_2 \leq N, K'[Id_2] = 0, K'' = \langle K', Id_2, 1\rangle, X'' = X'+1$$

Proving of subsumption of $\overline{\mathcal{G}_2}$ by $\overline{\mathcal{G}_1}$ is now equivalent to the proof of the verification condition

$$abs(N,K',X'), 1 \leq Id_2 \leq N, K'[Id_2] = 0, K'' = \langle K', Id_2, 1\rangle, X'' = X'+1 \models abs(N,K',X')[K''/K', X''/X'].$$

The purpose of renaming in the above example is to match the system variables of $\overline{\mathcal{G}_2}$ with those of $\overline{\mathcal{G}_1}$.

[1] *Termination* here means that no further execution is defined.

4 The Proof Method

In this key section, we consider proof obligations of the form $\mathcal{G} \models \mathcal{H}$ for goals $\mathcal{G}$ and $\mathcal{H}$ possibly containing recursive constraints.

Intuitively, we proceed as follows: unfold the recursive predicates in $\mathcal{G}$ *completely* a finite number of steps in order to obtain a "frontier" containing the goals $\mathcal{G}_1, \ldots, \mathcal{G}_n$. We note that "completely" here means that $\{\mathcal{G}_1, \ldots, \mathcal{G}_n\} = \text{UNFOLD}_A(\mathcal{G})$. We then unfold $\mathcal{H}$ obtaining goals $\mathcal{H}_1, \ldots, \mathcal{H}_m$, but this time not necessarily completely, that is, we only require that $\{\mathcal{H}_1, \ldots, \mathcal{H}_m\} \subseteq \text{UNFOLD}_B(\mathcal{H})$. This situation is depicted in Figure 3. Then, the proof holds if

$$\mathcal{G}_1 \vee \ldots \vee \mathcal{G}_n \models \mathcal{H}_1 \vee \ldots \vee \mathcal{H}_m$$

or alternatively, $\mathcal{G}_i \models \mathcal{H}_1 \vee \ldots \vee \mathcal{H}_m$ for all $1 \leq i \leq n$. This follows from the fact that $\mathcal{G} \models \mathcal{G}_1 \vee \ldots \vee \mathcal{G}_n$, (which is not true in general, but true in the least-model semantics of CLP), and the fact $\mathcal{H}_j \models \mathcal{H}$ for all j such that $1 \leq j \leq m$. If all variables in $\mathcal{H}$ appear in $\mathcal{G}$, we can reduce the proof to $\forall i : 1 \leq i \leq n, \exists j : 1 \leq j \leq m : \mathcal{G}_i \models \mathcal{H}_j$. Finally, we seek to eliminate the predicates in $\mathcal{H}_j$ so the remaining proof is one about basic constraints.

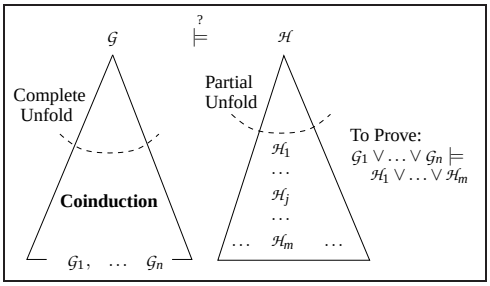

Fig. 3: Informal Structure of Proof Process

In this paper we do not go further with the proof of basic constraints. We instead assume the use of a standard black-box solver, such as the SMT solvers [13–15]. In our own experiments, we use a method from [16] to convert constraints on array segments into constraints on integers, and then dispatch the integer constraints using the real-number solver of CLP($\mathcal{R}$).

In addition to this overall idea of using left and right unfolds, there are a few more rules, as detailed below.

4.1 The Coinduction Rule

Before presenting our collection of proof rules, one of them, the coinduction rule, deserves preliminary explanation. Let us illustrate this rule on a small example. Consider the definition of the following two recursive predicates

```
m4(0).                 even(0).
m4(X+4) :- m4(X).      even(X+2) :- even(X).
```

whose domain is the set of non-negative integers. The predicate `m4` defines the set of multiples of four, whereas the predicate `even` defines the set of even numbers. We shall attempt to prove that `m4(X)`⊨`even(X)`, which in fact states that every multiple of four is even. We start the proof process by performing a *complete* unfolding on the lhs goal (see definition in Section 4). We note that `m4(X)` has two possible unfoldings, one leading to the empty goal with the answer `X=0`, and another one leading to the goal `m4(X')`,`X'=X-4`. The two unfolding operations, applied to the original proof obligation result in the following two new proof obligations, both of which need to be discharged in order to prove the original one.

$$\texttt{X=0} \models \texttt{even(X)} \quad (1) \qquad \texttt{m4(X')},\texttt{X'=X-4} \models \texttt{even(X)} \quad (2)$$

The proof obligation (1) can be easily discharged. Since unfolding on the lhs is no longer possible, we can only unfold on the rhs. We choose[1] to unfold with rule `even(0)`, which results in a new proof obligation which is trivially true, since its lhs and rhs are identical.

For proof obligation (2), before attempting any further unfolding, we note that the lhs `m4(X')` of the current proof obligation, and the lhs `m4(X)` of the original proof obligation, are unifiable (as long as we consider `X'` a fresh variable), which enables the application of the coinduction principle. First, we "discover" the *induction hypothesis* $\texttt{m4(X')} \models \texttt{even(X')}$, as a variant of the original proof obligation. Then, we use this induction hypothesis to replace `m4(X')` in (2) by `even(X')`. This yields the new proof obligation

$$\texttt{even(X')},\texttt{X'=X-4} \models \texttt{even(X)} \quad (3)$$

To discharge (3), we unfold twice on the rhs, using the `even(X+2) :- even(X)` rule. The resulting proof obligation is

$$\texttt{even(X')},\texttt{X'=X-4} \models \texttt{even(X''')},\texttt{X'''=X''-2},\texttt{X''=X-2} \quad (3)$$

where variables `X''` and `X'''` are existentially quantified[2]. Using constraint simplification, we reduce this proof obligation to $\texttt{even(X-4)} \models \texttt{even(X-4)}$, which is obviously true.

In the above example, `m4(X)` is unfolded to a goal with answer `X=0`, however, in general the proof method does not require a base case. We could remove the fact `m4(0)` from the definition of `m4`, and still obtain a successful proof. We call our technique "coinduction" from the fact that it does not require any base case.

4.2 The Proof Rules

We now present a formal calculus for the proof of assertions $\mathcal{G} \models \mathcal{H}$. To handle the possibly infinite unfoldings of $\mathcal{G}$ and $\mathcal{H}$, we shall depend on coinduction, which allows the assumption of a *previous* obligation. The proof proceeds by manipulating a set of *proof obligations* until it finally becomes empty or a counterexample is found. Formally, a *proof obligation* is of the form $\tilde{A} \vdash \mathcal{G} \models \mathcal{H}$ where the $\mathcal{G}$ and $\mathcal{H}$ are goals and $\tilde{A}$ is a set of *assumption* goals whose assumption (coinductively) can be used to discharge the proof obligation at hand. This set is implemented in our algorithm as a memo table.

Our proof rules are presented in Figure 4. The $\uplus$ symbol represents the disjoint union of two sets, and emphasizes the fact that in an expression of the form $A \uplus B$, we have that $A \cap B = \emptyset$. Each rule operates on the (possibly empty) set of proof obligations Π, by selecting one of its proof obligations and attempting to discharge it. In this process, new proof obligations may be produced. We note that our proof rules are presented in the "reverse" manner than usual, where the conclusions to be proven is written above the horizontal line and the premise to achieve the conclusion is written below the line. Our proof rules can be considered as a system of production of premises whose proofs establish the desired conclusion.

The *left unfold with new induction hypothesis* (LU+I) (or simply "left unfold") rule performs a complete unfold on the lhs of a proof obligation, producing a new set of

[2] For clarity, we sometimes prefix such variables with '?'.

$$(\text{LU+I})\ \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \bigcup_{i=1}^{n} \{\tilde{A} \cup \{\mathcal{G} \models \mathcal{H}\} \vdash \mathcal{G}_i \models \mathcal{H}\}} \quad \text{UNFOLD}(\mathcal{G}) = \{\mathcal{G}_1, \ldots, \mathcal{G}_n\}$$

$$(\text{RU})\ \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}'\}} \quad \mathcal{H}' \in \text{UNFOLD}(\mathcal{H})$$

$$(\text{CO})\ \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \{\tilde{A} \vdash \mathcal{H}'\theta \models \mathcal{H}\}} \quad \mathcal{G}' \models \mathcal{H}' \in \tilde{A} \text{ and there exists a substitution } \theta \text{ s.t. } \mathcal{G} \models \mathcal{G}'\theta$$

$$(\text{CP})\ \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \wedge p(\tilde{x}) \models \mathcal{H} \wedge p(\tilde{y})\}}{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H} \wedge \tilde{x} = \tilde{y}\}}$$

$$(\text{SPL})\ \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \bigcup_{i=1}^{k} \{\tilde{A} \vdash \mathcal{G} \wedge \psi_i \models \mathcal{H}\}} \quad \psi_1 \vee \ldots \vee \psi_k \text{ is valid}$$

$$(\text{EXR})\ \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}(z)\}}{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \wedge z = e \models \mathcal{H}(z)\}} \quad z \text{ is existential}$$

Fig. 4: Proof Rules for Recursive Constraints

proof obligations. The original formula, while removed from Π, is added as an assumption to every newly produced proof obligation, opening the door to using coinduction later in the proof.

The rule *right unfold* (RU) performs an unfold operation on the rhs of a proof obligation. In general, the two unfold rules will be systematically interleaved. The resulting proof obligations are then discharged either coinductively or directly, using the (CO) and (CP) rules, respectively.

The rule *coinduction application* (CO) transforms an obligation by using an assumption, and thus opens the door to discharging that obligation via the direct proof (CP) rule. Since assumptions can only be created using the (LU+I) rule, the (CO) rule realizes the coinduction principle. The underlying principle behind the (CO) rule is that a "similar" assertion $\mathcal{G}' \models \mathcal{H}'$ has been previously encountered in the proof process, and assumed as true.

Note that this test for coinduction applicability is itself of the form $\mathcal{G} \models \mathcal{H}$. However, the important point here is that this test can only be carried out using basic constraints, in the manner prescribed for the CP rule described below. In other words, this test does not use the definitions of (recursive) predicates.

The rule *constraint proof* (CP), when used repeatedly, discharges a proof obligation by reducing it to a form which contains no recursive predicates. The intended use of this rule is in case the recursive predicates of the rhs is the subset of the recursive predicates of the lhs such that repeated applications of the rule results in rhs containing no recursive predicates. We then simply ignore the lhs predicates and attempt to establish the remaining obligation using our basic constraint solver.

The rule *split* (SPL) rule is straightforward: to break up the proof into pieces. The rule *existential removal* (EXR) rule is similarly straightforward: to remove one instance of an *existential* variable, one that appears only in the rhs. What is not straightforward

REDUCE($\mathcal{G} \models \mathcal{H}$) **returns boolean**

choose one of the following:

- Constraint Proof: (CP) + Constraint Solving
  Apply a constraint proof to $\mathcal{G} \models \mathcal{H}$.
  If successful, **return** *true*, otherwise **return** *false*
- Memoize ($\mathcal{G} \models \mathcal{H}$) as an assumption
- Coinduction: (CO)
  if there is an assumption $\mathcal{G}' \models \mathcal{H}'$ such that
  REDUCE($\mathcal{G} \models \mathcal{G}'\theta$) = *true* $\wedge$ REDUCE($\mathcal{H}'\theta \models \mathcal{H}$) = *true*
  then return *true*.
- Unfold:
  choose left or right
  case: Left: (LU+I)
  choose an atom A in $\mathcal{G}$ to reduce
  for all reducts $\mathcal{G}_L$ of $\mathcal{G}$ using A: if REDUCE($\mathcal{G}_L \models \mathcal{H}$) = *false* **return** *false*
  return *true*
  case: Right: (RU)
  choose an atom A in $\mathcal{H}$ to reduce, obtaining $\mathcal{G}_R$
  return REDUCE($\mathcal{G} \models \mathcal{G}_R$)
- Split:
  Find an index variable Id and a parameter variable N and apply the split rule using $Id \neq N \vee Id = N$ to split $\mathcal{G}$ into $\mathcal{G}_1$ and $\mathcal{G}_2$.
  return REDUCE($\mathcal{G}_1 \models \mathcal{H}$) $\wedge$ REDUCE($\mathcal{G}_2 \models \mathcal{H}$)
- Existential Variable Removal:
  If an existential array variable z appears in the form $z = \langle x, i, e\rangle$, then simply substitute z by $\langle x, i, e\rangle$ everywhere (in $\mathcal{H}$). If however z appears in the form $x = \langle z, i, e\rangle$ where x is not existential, then find an expression in $\mathcal{G}$ of the form $x = \langle x', i, e\rangle$ and replace z by x'. Let the result be $\mathcal{H}'$.
  return REDUCE($\mathcal{G} \models \mathcal{H}'$)

Fig. 5: Search Algorithm for Recursive Constraints

however is precisely how we use the SPL and EXR rules: in the former case, how do we choose the constraints ψ_i? And in the latter, how do we choose the expression e? We present answers to this in the search algorithm below.

4.3 The Search Algorithm

Given a proof obligation $\mathcal{G} \models \mathcal{H}$, a proof shall start with $\Pi = \{\emptyset \vdash \mathcal{G} \models \mathcal{H}\}$, and proceed by repeatedly applying the rules in Figure 4 to it. We now describe a strategy so as to make the application of the rules automated. Here we propose systematic interleaving of the left-unfold (LU+I) and right-unfold (RU) rules, attempting a constraint proof along the way. As CLP can be executed by resolution, we can also execute our proof rules, based on an algorithm which has some resemblance to tabled resolution.

We present our algorithm in pseudocode in Figure 5. Note that the presentation is in the form of a nondeterministic algorithm, and the order executing each choice of the nondeterministic operator **choose** needs to be implemented by some form of systematic strategy, for example, by a breadth-first strategy. Clearly there is a combinatorial explosion here, but in practice the number of steps required for a proof is not large. Even so,

the matter of efficiently choosing which order to apply the rules is beyond the scope of this paper.

In Figure 5, by a *constraint proof* of a obligation, we mean to repeatedly apply the CP rule in order to remove all occurrences of predicates in the obligation, in an obvious way. Then the basic constraint solver is applied to the resulting obligation.

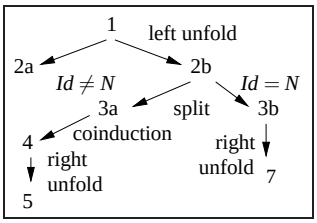

Fig. 6: Proof Tree

Next consider the split rule in Figure 5. Note that we have specified the rather specific instance of the SPL rule in which we replace a constraint of the form $Id \leq N$, where Id is an index variable and N represents the parameter, by (a disjunction of) two constraints $Id \leq N, Id = N$ ($Id = N$) and $Id \leq N, Id \neq N$ ($Id < N$). The reason for this is purely technical; it is essentially because our recursive assertions depend on $Id \leq N$ and since they are recursive on N, a recursive state may end up with the situation where $Id > N-1$, a situation which is not similar to the parent state.

Finally consider the existential variable elimination rule in Figure 5. The essential idea here is simply that an existential variable is most likely to correspond to some array expression on the lhs. Once again, this choice of existential variable elimination is purely technical and was created because it works in practice.

Lemma 1 (Soundness of Rules). *$\mathcal{G} \models \mathcal{H}$ if, starting with the proof obligation $\emptyset \vdash \mathcal{G} \models \mathcal{H}$, there exists a sequence of applications of proof rules that results in proof obligations $\tilde{A} \vdash \mathcal{G}' \models \mathcal{H}'$ such that (a) $\mathcal{H}'$ contains only constraints, and (b) $\mathcal{G}' \models \mathcal{H}'$ can be discharged by the basic constraint solver.* □

5 Examples

5.1 Counting Ones

In Figure 1, the tree is closed is due to state subsumption formalized as $\overline{\mathcal{G}_2} \models \overline{\mathcal{G}_1}$:
1: $1 \leq Id_2 \leq N, K'[Id_2] = 0 \models abs(N, \langle K', id_2, 1\rangle, X'+1)$
A complete proof tree is outlined in Figure 6. The algorithm left unfolds Obligation 1 into 2a and 2b (not shown). Obligation 2a can be proved directly. Obligation 2b is now split into 3a and 3b. For 3a, we add the constraint $Id \neq N$, and for 3b we add the complementary constraint $Id = N$. We omit detailing 3b, and we proceed with explaining the proof of 3a. Obligation 3a is as follows:
3a: $abs(N-1, K'', X'-B), bit(B), K''[Id_2] = 0, 1 \leq Id_2 < N$
$\models abs(N, \langle\langle K'', N, B\rangle, Id_2, 1\rangle, X'+1)$
We now perform the crucial step of applying coinduction to Obligation 3a. This is permitted because the lhs of 1 is entailed by the lhs goal 3a. To see this, perform the substitutions $[N-1/N, X'-B/X']$ on Obligation 1. The result of applying coinduction is:
4: $abs(N-1, \langle K'', Id_2, 1\rangle, X'-B+1), bit(B), K''[Id_2] = 0, 1 \leq Id_2 < N$
$\models abs(N, \langle\langle K'', N, B\rangle, Id_2, 1\rangle, X'+1)$
We now right unfold this into Obligation 5, and prove Obligation 5 by constraint reasoning, which is omitted.

```
process(id) {
⟨0⟩  t[id] = max(t[1],...,t[N]) + 1;
⟨1⟩  await(forall j!=id : t[id]==0 ∨ t[id]<t[j]);
⟨2⟩  t[id] = 0; goto ⟨0⟩ }

sys(K,T,N) :- K[Id]=0, 1≤Id≤N, max(T,N,X), sys(⟨K,Id,1⟩, ⟨T,Id,X+1⟩, N).
sys(K,T,N) :- K[Id]=1, 1≤Id≤N, crit(T,N,Id), sys(⟨K,Id,2⟩,T,N).
sys(K,T,N) :- K[Id]=2, 1≤Id≤N, sys(⟨K,Id,0⟩, ⟨T,Id,0⟩,N).

abs(K,T,1) :- (K[1] = 0,T[1] = 0) ∨ (K[1] = 1,T[1] > 0).
abs(K,T,N) :- N > 1,((K[N] = 0,T[N] =) ∨ (K[N] = 1,T[N] > 0)),abs(K,T,N−1).

max(T,1,X) :- X ≥ T[1].
max(T,N,X) :- N > 1,X ≥ T[N],max(T,N−1,X).

crit(T,1,Id) :- Id = 1 ∨ T[1] = 0 ∨ T[1] > T[Id].
crit(T,N,Id) :- N > 1,(Id = N ∨ T[N] = 0 ∨ T[N] > T[Id]),crit(T,N−1,Id).
```

Fig. 7: Transitions and Predicates for Bakery

5.2 Bakery Algorithm (Atomic Version)

To show a more substantial example, consider the bakery mutual exclusion algorithm [17]. Here we consider, somewhat unrealistically, a simplified presentation where the test for entry into the critical section, which considers the collection of all processes, is assumed to be performed atomically.

We represent the transitions and the recursive abstractions used in Figure 7.

A closed computation tree is depicted in Figure 8. The initial state $\mathcal{G}_0$ is where the counter is all zeroes, and the local variables $T[]$ (the "tickets") are also all zero. The state $\mathcal{G}_1$ denotes one transition of one process, symbolically denoted by Id, from point $\langle 0\rangle$ to $\langle 1\rangle$. At this point we perform an abstraction to obtain a state $\overline{\mathcal{G}_1}$ which contains not one but a number of program points at 1. This abstraction also constrains the tickets so that if a counter is zero, then the corresponding ticket is also zero.

No further abstraction is needed. That is, the computation tree under $\overline{\mathcal{G}_1}$ is in fact closed, as indicated. Note that mutual exclusion then follows from the fact that from state $\mathcal{G}_{2b}$ or $\mathcal{G}_{3a}$, the only states in which a process is in the critical section, there is no possible transition by a different process to enter the section. This is emphasized by the the notation "infeasible" in Figure 8.

One of the conditions to show closure is that the (leaf) state $\mathcal{G}_{3a}$ is subsumed by $\mathcal{G}_{2b}$. (There are several others, eg. that $\mathcal{G}_{3c}$ is subsumed by $\overline{\mathcal{G}_1}$. We shall omit considering these.) This is formalized as:

$$\begin{aligned}\text{D.1}: \; & abs(K',T',N), crit(T',N,Id_1), max(T',N,X), 1 \le Id_1 \le N,\\ & K'[Id_1] = 1, 1 \le Id_2 \le N, \langle K',Id_1,2\rangle[Id_2] = 0\\ & \models abs(?S,\langle T',Id_2,X+1\rangle,N), crit(\langle T',Id_2,X+1\rangle,N,?Id_3),\\ & 1 \le ?Id_3 \le N, ?S[?Id_3] = 1, \langle\langle K',Id_1,2\rangle,Id_2,1\rangle = \langle ?S,?Id_3,2\rangle\end{aligned}$$

In the above, the prefix '?' denotes existentially-quantified variables. For space reasons, we omit the detailed proof. Instead, we depict in the proof tree of Figure 8 the major steps that can be used in the proof.

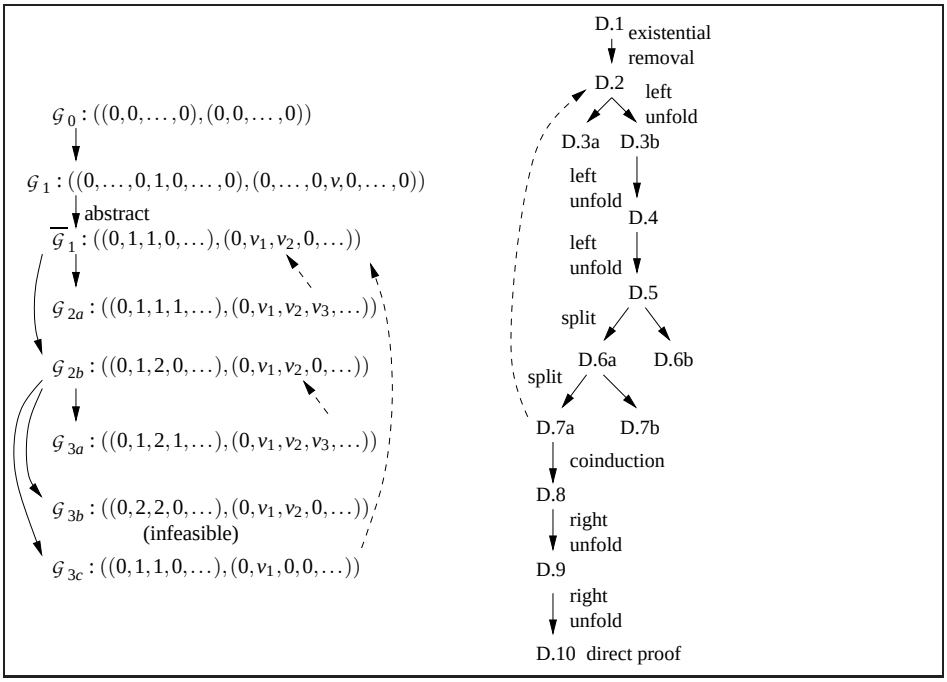

Fig. 8: Computation and Proof Trees of Bakery Algorithm

5.3 Original Bakery Algorithm

We finally discuss the original version of the bakery algorithm [17]. Our purpose here is to demonstrate abstraction beyond an array of variables. Here, abstraction is needed because there is an additional loop implementing the incremental request for entry into the critical section. To our knowledge, we provide the first systematic proof of the original bakery algorithm. Our proof technique is semiautomatic, where the user only provide the declarative specification of loop invariants.

We show the program in Figure 9. We focus on the replacement of the *await* blocking primitive in Figure 7 by a loop from $\langle 3 \rangle$ to $\langle 8 \rangle$, which itself contains two internal busy-waiting loops. Figure 9 also shows the transition system of the loop, and the predicate that is used. In the program and elsewhere, the operator $\prec$ is defined as follows: when $(a,b) \prec (c,d)$ holds, then either $a < b$ or when $a = b$, then $b < d$.

Figure 10 depicts an abstract computation tree. The state $\mathcal{G}_2$ represents entry into the outerloop, and $\overline{\mathcal{G}_2}$ its abstraction. $\mathcal{G}_{3a}$ is its exit. The states $\mathcal{G}_{3b}$ and $\mathcal{G}_{ba}$ represent the two inner loops. The interesting aspect is the abstraction indicated. It alone is sufficient to produce a closed tree. More specifically, we abstract $\mathcal{G}_2 : \mathbf{sys}(K',C,T,J',N), K[Id_1] = 3, K' = \langle K, Id_1, 4\rangle, J' = \langle J, Id_1, 1\rangle$ into $\overline{\mathcal{G}_2} : \mathbf{sys}(K',C,T,J',N), K'[Id_1] = 4, crit(C,T,J',Id_1), 1 \leq J'[Id_1] \leq N+1$.

The state subsumption is formalized as the entailment $\mathcal{G}_6 \models \overline{\mathcal{G}_2}$ as follows:
$K'[Id_1] = 4, crit(C,T,J',Id_1), 1 \leq J'[Id_1] \leq N+1, K'[Id_2] = 4,$
$K'' = \langle K', Id_2, 5\rangle, K''[Id_3] = 5, K''' = \langle K', Id_3, 6\rangle, C[J'[Id_3]] = 3,$
$T[J'] = 0 \vee (T[Id_4], Id_4) \prec (T[J'[Id_4]], J'[Id_4]), K'''[Id_4] = 6, K^{iv} = \langle K''', Id_4, 7\rangle,$

```
    process(id) {
⟨0⟩     c[id] = 1;
⟨1⟩     t[id] = 1 + maximum(t[1],...,t[n]);
⟨2⟩     c[id] = 0;
⟨3⟩     j[id] = 1;
⟨4⟩     while (j ≤ N) {
⟨5⟩         if (c[j]!=0) goto ⟨5⟩;
⟨6⟩         if (t[j]!=0 && (t[j],j) ≺ (t[i],i)) goto ⟨6⟩;
⟨7⟩         j = j+1; }
⟨8⟩     goto ⟨0⟩; }

sys(K,C,T,J,N) :- K[Id]=3, sys(⟨K,Id,4⟩,C,T,⟨J,Id,1⟩,N)
sys(K,C,T,J,N) :- K[Id]=4, J[Id]≤ N, sys(⟨K,Id,5⟩,C,T,J,N)
sys(K,C,T,J,N) :- K[Id]=4, J[Id]>N, sys(⟨K,Id,8⟩,C,T,J,N)
sys(K,C,T,J,N) :- K[Id]=5, C[J]≠ 0, sys(K,C,T,J,N)
sys(K,C,T,J,N) :- K[Id]=5, C[J]=0, sys(⟨K,Id,6⟩,C,T,J,N)
sys(K,C,T,J,N) :- K[Id]=6, T[J]≠0, ((T[J],J)≺(T[Id],Id)), sys(K,C,T,J,N)
sys(K,C,T,J,N) :- K[Id]=6, (T[J]=0∨((T[Id],Id)≺(T[J],J)),
        sys(⟨K,Id,7⟩,C,T,J,N)
sys(K,C,T,J,N) :- K[Id]=7, sys(⟨K,Id,4⟩,C,T,⟨J,Id,J[Id]+1⟩,N)

crit(C,T,1,Id) :- Id = 1 ∨ (C[1] = 0,(T[Id],Id) ≺ (T[1],1))
crit(C,T,N,Id) :- Id = N ∨ (C[N] = 0,(T[Id],Id) ≺ (T[N],N)),crit(C,T,N−1,Id).
```

Fig. 9: Original Bakery with Transitions of the Entry Loop and Predicate

$K^{iv}[Id_5] = 7, K^v = \langle K^{iv}, Id_5, 4\rangle, J'' = \langle J', Id, J'[Id_5]+1\rangle$
$\models crit(C,T,J'',?Id_6), K^v[Id_1] = 4, 1 \le J''[?Id_6] \le N+1$

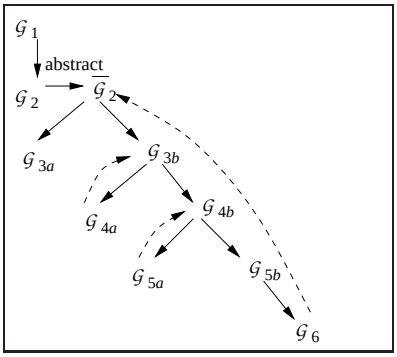

Fig. 10: Abstract Computation Tree for Entry Loop

which can be proven along the lines indicated above. We omit the details.

6 Concluding Remarks

We presented a language of recursively defined formulas about arrays of variables for the purpose of specifying abstract states of parameterized systems. We then present a symbolic transition framework for these formulas. This can produce a finite representation of the behaviour of the system from which safety properties can be ascertained. The main result is a two step algorithm for proving entailment of these formulas. In the first step, we employ a key concept of coinduction in order to reduce the recursive definitions to formulas about arrays and integers. In the second, we reduced these formulas to integer formulas.

Though we considered only safety properties in this paper, it is easy to see that our notion of

closed abstract tree does in fact contain the key information needed to argue about termination and liveness. Essentially, this is because our framework is equiped with symbolic transitions. What is needed is to show that in every path ending in a subsumed state, that the execution from the parent state decreases a well founded measure.

References

1. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized verification of infinite-state processes with gl obal constraints. In Damm, W., Hermanns, H., eds.: 19th CAV. Volume 4590 of LNCS., Springer (2007) 145–157
2. Roychoudhury, A., Ramakrishnan, I.V.: Automated inductive verification of parameterized protocols. In Berry, G., Comon, H., Finkel, A., eds.: 13th CAV. Volume 2102 of LNCS., Springer (2001) 25–37
3. E.M. Clarke, M.T., Veith, H.: Environment abstraction for parameterized verification. In Emerson, E.A., Namjoshi, K.S., eds.: 7th VMCAI. Volume 3855 of LNCS., Springer (2006)
4. Lahiri, S., Bryant, R.: Indexed predicate discovery for unbounded system verification. In: 16th CAV, Volume 3114 of LNCS, Springer (2004).
5. McMillan, K.L.: Induction in compositional model checking. In Emerson, E.A., Sistla, A.P., eds.: CAV 2000. Volume 1855 of LNCS., Springer (2000) 312–327
6. A. Pnueli, S.R., Zuck, L.: Automatic deductive verification with invisible invariants. In Margaria, T., Yi, W., eds.: 7th TACAS. Volume 2031 of LNCS., Springer (2001)
7. A. Pneuli, J.X., Zuck, L.: Liveness with $(0,1,\infty)$ counter abstraction. In Brinksma, E., Larsen, K.G., eds.: 14th CAV. Volume 2404 of LNCS., Springer (2002)
8. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis. In: 4th POPL, ACM Press (1977) 238–252
9. Jaffar, J., Santosa, A.E., Voicu, R.: A coinduction rule for entailment of recursively defined properties. In Stuckey, P.J., ed.: 14th CP. Volume 5202 of LNCS., Springer (2008) 493–508
10. McCarthy, J.: Towards a mathematical science of computation. In Popplewell, C.M., ed.: IFIP Congress 1962, North-Holland (1983)
11. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. J. LP **19/20** (May/July 1994) 503–581
12. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer (1989)
13. Barrett, C., Dill, D.L., Levitt, J.R.: Validity checking for combinations of theories with equality. 1st FMCAD, LNCS 1166 (1996) 187–201
14. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In: 16th CAV, Volume 3114 of LNCS, Springer (2004).
15. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems **1**(2) (October 1979) 245–257
16. Jaffar, J., Lassez, J.L.: Reasoning about array segments. In: ECAI 1982. (1982) 62–66
17. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Comm. ACM **17**(8) (August 1974) 453–455