A CLP Method for Compositional and Intermittent Predicate Abstraction

JOXAN JAFFAR, ANDREW E. SANTOSA, AND RĂZVAN VOICU

School of Computing
National University of Singapore
S16, 3 Science Drive 2, Singapore 117543
Republic of Singapore
{joxan,andrews,razvan}@comp.nus.edu.sg

Abstract. We present an implementation of symbolic reachability analysis with the features of compositionality, and *intermittent* abstraction, in the sense of pefrorming approximation only at selected program points, if at all. The key advantages of compositionality are well known, while those of intermittent abstraction are that the abstract domain required to ensure convergence of the algorithm can be minimized, and that the cost of performing abstractions, now being intermittent, is reduced.

We start by formulating the problem in CLP, and first obtain compositionality. We then address two key efficiency challenges. The first is that reasoning is required about the strongest-postcondition operator associated with an arbitrarily long program fragment. This essentially means dealing with constraints over an unbounded number of variables describing the states between the start and end of the program fragment at hand. This is addressed by using the variable elimination or projection mechanism that is implicit in CLP systems. The second challenge is termination, that is, to determine which subgoals are redundant. We address this by a novel formulation of memoization called *coinductive tabling*.

We finally evaluate the method experimentally. At one extreme, where abstraction is performed at every step, we compare against a model checker. At the other extreme, where no abstraction is performed, we compare against a program verifier. Of course, our method provides for the middle ground, with a flexible combination of abstraction and Hoare-style reasoning with predicate transformers and loop-invariants.

1 Introduction

Predicate abstraction [15] is a successful method of abstract interpretation. The abstract domain, constructed from a given finite set of predicates over program variables, is intuitive and easily, though not necessarily efficiently, computable within a traversal method of the program's control flow structure.

While it is generally straightforward to optimize the process of abstraction to a certain extent by performing abstraction at selected points only (eg. several consecutive asignments may be compressed and abstraction performed accross one composite assignment, as implemented in the BLAST system [19]), to this point there has not been a systematic way of doing this. Moreover, since the abstract description is limited to a fixed number of variables, such an ad-hoc method would not be compositional. For example, [2] requires an elaborate extension of predicate abstraction which essentially considers a second set of variables (called "symbolic constants"), in order to describe the behaviour of a *function*, in the language of predicate abstraction. This provides only a limited form of compositionality.

```
⟨0⟩ i := 0 ; c := 0
⟨1⟩ while (i < n) do
⟨2⟩   i++
⟨3⟩   c++
⟨4⟩   c++
⟨5⟩ end ⟨6⟩
```

(a)

$$
\begin{array}{l}
even(0,i,n,c) \mapsto even(1,i_1,n_1,c_1), i_1 = 0, n_1 = n, c_1 = 0.\\
even(1,i,n,c) \mapsto even(2,i_1,n_1,c_1), i_1 = i, n_1 = n, i < n, c_1 = c.\\
even(2,i,n,c) \mapsto even(3,i_1,n_1,c_1), i_1 = i+1, n_1 = n, c_1 = c.\\
even(3,i,n,c) \mapsto even(4,i_1,n_1,c_1), i_1 = i+1, n_1 = n, c_1 = c+1.\\
even(4,i,n,c) \mapsto even(5,i_1,n_1,c_1), i_1 = i+1, n_1 = n, c_1 = c+1.\\
even(5,i,n,c) \mapsto even(2,i_1,n_1,c_1), i_1 = i, n_1 = n, c_1 = c, i < n.\\
even(5,i,n,c) \mapsto even(6,i_1,n_1,c_1), i_1 = i, n_1 = n, c_1 = c, i \geq n.
\end{array}
$$

(b)

Fig. 1. Even counts

In this paper, we present a way of engineering a general proof method of program reasoning based on predicate abstraction in which the process of abstraction is intermittent, that is, approximation is performed only at selected program points, if at all. There is no restriction of when abstraction is performed, even though termination issues will usually restrict the choices. The key advantages are that (a) the abstract domain required to ensure convergence of the algorithm can be minimized, and (b) the cost of performing abstractions, now being intermittent, is reduced.

For example, to reason that $x = 2$ after executing `x := 0; x++; x++`, one needs to know that $x = 1$ holds before the final assignment. Thus, in a predicate abstraction setting, the abstract domain must contain the predicate $x = 1$ for the above reasoning to be possible. Also, consider proving $x = 2n$ for the program snippet in Figure 1a. A textbook Hoare-style loop invariant for the loop is $c = 2i$. Having this formula in the abstract domain would, however, not suffice; one in fact needs to know that $c = 2i - 1$ holds in between the two increments to c. Thus in general, a proper loop invariant is useful only if we could propagate its information throughout the program *exactly*.

A main challenge with exact propagation is that reasoning will be required about the strongest-postcondition operator associated with an arbitrarily long program fragment. This essentially means dealing with constraints over an unbounded number of variables describing the states between the start and end of the program fragment at hand. The advantages in terms of efficiency, however, are significant: less predicates needed in the abstract domain, and also, less frequent execution of the abstraction operation. Alternatively, it may be argued that using the weakest precondition operator for exact propagation may result in a set of constraints over a constant number of variables, and thus circumvent the challenge mentioned above. To see that this is not true, let us consider the following program fragment: `while(x%7!=0)x++ ; while(x%11!=0)x++`. Also, let us assume that we have an exact propagation algorithm, based on either the weakest preconditon or the strongest postcondition propagation operator, which computes a constraint that reflects the relationship between the values of `x` before and after the execution of the program fragment. Our algorithm needs to record the fact that between the two while loops the value of `x` is a multiple of 7. This cannot be done without introducing an auxilliary variable in the set of constraints. Assume now that this program fragment appears in the body of another loop. Since that (outer) loop may be traversed multiple times in the analysis process, and every traversal of the loop will introduce a new auxilliary variable, the number of auxilliary variables is potentially unbounded, irrespective of the propagation operator that is used.

An important feature of our proof method is that it is compositional. We represent a proof as a Hoare-style triple which, for a given program fragment, relates the input values of the variables to the output values. This is represented as a formula, and in general, such a formula must contain auxiliary variables in addition to the program variables. This is because it is generally impossible to represent the projection of a formula using a predefined set of variables, or equivalently, it is not possible to perform quantifier elimination. Consequently, in order to have unrestricted composition of such proofs, it is (again) necessary to deal with an unbounded number of variables.

The paper is organized as follows. We start by formulating the problem in CLP, and first obtain compositionality. We then address two key efficiency challenges. The first is that reasoning is required about the strongest-postcondition operator associated with an arbitrarily long program fragment. This means dealing with constraints over an unbounded number of variables describing the states between the start and end of the program fragment at hand. We address this problem by using the variable elimination or projection mechanism that is implicit in CLP systems. The second challenge is termination, which translates into determining the redundancy of subgoals. We address this by a novel formulation of memoization called *coinductive tabling*.

We finally evaluate the method experimentally. At one extreme, where abstraction is performed at every step, we compare against the model checker BLAST [19]. Here we employ a standard realization of intermittence by abstracting at prespecified points, and thus our algorithm becomes automatic. At the other extreme, where no abstraction is performed (but where invariants are used to deal with loops), we compare against the program-verifier ESC/Java [6]. Of course, our method provides for the middle ground, with a flexible combination of abstraction and Hoare-style reasoning with predicate transformers and loop-invariants.

In summary, we present a CLP-based proof method which has the properties of being compositional, and which employs intermittent abstraction. The major technical contributions, toward this goal, are: the CLP formulation of the proof obligation, which provides expressiveness, and compositionality; a coinduction principle, which provides the basic mechanism for termination; and engineering the use of the underlying CLP projection mechanism in the process of exact propagation. Our method thus provides a flexible combination of abstraction and Hoare-style reasoning with predicate transformers and loop-invariants, that is compositional, and its practical implementation is feasible.

1.1 Related Work

An important category of tools that use program verification technology have been developed within the framework of the Java Modelling Language (JML) project. JML allows one to specify a Java method's pre- and post-conditions, and class invariants. Examples of such program verification tools are: Jack [11], ESC/Java2 [6], and Krakatoa [24]. All these tools employ weakest precondition/strongest postcondition calculi to generate proof obligations which reflect whether the given post-conditions and class invariants hold at the end of a method, whenever the corresponding pre-conditions are valid at the procedure's entry point. The resulting proof obligations are subsequently discharged by theorem provers such as Simplify [6], Coq [3], PVS [27], or HOL light [18]. While these systems perform exact propagation, they depend on user-provided loop invariants, as opposed to an abstract domain.

Cousot and Cousot [7] have recognized a long time ago that coarse-grained abstractions are better than fine-grained ones. Moreover, recently there have emerged systems based on abstract interpretation, and in particular, on predicate abstraction. Some examples are BLAST [19], SLAM [1], MAGIC [5], and Murphi– – [8], amongst others. While abstract interpretation is central, these systems employ a further technique of *automatically* determining the abstract domain needed for a given assertion. This technique iteratively refines the abstract domain based on information derived from previous counterexamples. These systems do not perform exact propagation in a systematic way.

The use of CLP for program reasoning is not new (see for example [14] for a non-exhaustive survey). Due to its capability for handling constraints, CLP has been notably used in verification of infinite-state systems [9, 10, 13, 17, 23], although results for finite-state systems are also available [26, 12]. Indeed, it is generally straightforward to represent program transitions as CLP rules, and to use the CLP operational model to prove assertions stated as CLP goals. What is novel in our CLP formulation is firstly, the compositional assertion, and then, coinductive tabling. More importantly, our formulation considers CLP programs, assertions and tabling in full generality.

2 Preliminaries

Apart from a program counter k, whose values are program points, let there be *n system variables* $\tilde{v} = v_1, \cdots, v_n$ with domains $\mathcal{D}_1, \cdots, \mathcal{D}_n$ respectively. In this paper, we shall use just two example domains, that of integers, and that of integer arrays. We assume the number of system variables is larger than the number of variables required by any program fragment or procedure.

Definition 1 (States and Transitions). *A* system state *(or simply state) is of the form* $(k, d_1, \cdots, d_n)$ *where k is a program point and* $d_i \in \mathcal{D}_i, 1 \leq i \leq n$*, are values for the system variables. A* transition *is a pair of states.* $\Box$

In what follows, we define a language of first-order formulas. Let $\mathcal{V}$ denote an infinite set of variables, each of which has a type in $\mathcal{D}_1, \cdots, \mathcal{D}_n$, let Σ denote a set of *functors*, and Π denote a set of *constraint symbols*. A *term* is either a constant (0-ary functor) in Σ or of the form $f(t_1, \cdots, t_m)$, $m \geq 1$, where $f \in \Sigma$ and each t_i is a term, $1 \leq i \leq m$. A *primitive constraint* is of the form $\phi(t_1, \cdots, t_m)$ where ϕ is a $m-ary$ constraint symbol and each t_i is a term, $1 \leq i \leq m$.

A *constraint* is constructed from primitive constraints using logical connectives in the usual manner. Where Ψ is a constraint, we write $\Psi(\tilde{X})$ to denote that Ψ possibly refers to variables in $\tilde{X}$, and we write $\tilde{\exists}\Psi(\tilde{X})$ to denote the existential closure of $\Psi(\tilde{X})$ over variables distinct from those in $\tilde{X}$.

A *substitution* is a mapping which simultaneously replaces each variable in a term or constraint by some expression. Where e is a term or constraint, we write $e\theta$ to denote the result of applying θ to e. A *renaming* maps each variable in a given sequence, say $\tilde{X}$, into the corresponding variable in another given sequence, say $\tilde{Y}$. We write $[\tilde{X} \mapsto \tilde{Y}]$ to denote such a mapping. A *grounding substitution*, or simply *grounding* maps each variable of an expression into a ground term representing a value in its respective domain. We denote by $[\![e]\!]$ the set of *all possible* groundings of e.

3 Constraint Transition Systems

A key concept is that a program fragment P operates on a sequence of *anonymous* variables, each corresponding to a system variable at various points in the computation of P.

In particular, we consider two sequences $\tilde{x} = x_1, \cdots, x_n$ and $\tilde{x}^t = x_1^t, \cdots, x_n^t$ of anonymous variables to denote the system values before executing P and at the "target" point(s) of P, respectively. Typically, but not always, the target point is the terminal point of P. Our proof obligation or *assertion* is then of the form

$$\{\Psi(\tilde{x})\}\ P\ \{\Psi_1(\tilde{x}, \tilde{x}^t)\}$$

where Ψ and Ψ_1 are constraints over the said variables, and possibly including new variables. Like the Hoare-triple, this states that if P is executed in a state satisfying Ψ, then all states at the target points (if any) satisfy Ψ_1. Note that, unlike the Hoare-triple, P may be nonterminating and Ψ_1 may refer to the states of a point that is reached infinitely often. We will formalize all this below.

For example, let there be just one system variable x, let P be `<0>` x `:=` x `+ 1 <1>`, and let the target point be `<1>`. Then $\{true\}P\{x^t = x + 1\}$ holds, meaning P is the successor *function* on x. Similarly, if P were the (perpetual) program `<0> while (`$true$`)` x `:=` x `+ 2 <1> endwhile <2>`, and if `<1>` were the target point, then $\{true\}P\{x^t = x + 2z\}$ holds, that is, any state $(1, x)$ at point `<1>` satisfies $\exists z(x^t = x + 2z)$. This shows, amongst other things, that the parity of x always remains unchanged.

Our proof method accomodates concurrent programs of a fixed number of processes. Where we have n processes, we shall use as a program point, a sequence of n program points so that the i^{th} program point is one which comes from the i^{th} process, $1 \leq i \leq n$.

We next represent the program fragment P as a transition system which can be executed symbolically. The following key definition serves two main purposes. First, it is a high level representation of the operational semantics of P, and in fact, it represents its exact *trace* semantics. Second, it is an *executable specification* against which an assertion can be checked.

Definition 2 (Constraint Transition System). *A* constraint transition *of P is a formula*

$$p(k, \tilde{x}) \mapsto p(k_1, \tilde{x_1}), \Psi(\tilde{x}, \tilde{x_1})$$

where k and k_1 are variables over program points, each of $\tilde{x}$ and $\tilde{x_1}$ is a sequence of variables representing a system state, and Ψ is a constraint over $\tilde{x}$ and $\tilde{x_1}$, and possibly some additional auxiliary variables.

A constraint transition system *(CTS) of P is a finite set of constraint transitions of P. The symbol p is called the CTS predicate of P.* $\square$

In what follows, unless otherwise stated, we shall consistently denote by P the program of interest, and by p its CTS predicate.

Consider for example the program in Figure 1a; call it *Even*. Figure 1b shows a CTS for *Even*, whose CTS predicate is *even*.

Consider another example: the Bakery algorithm with two processes in Figure 2. A CTS for this program, call it *Bak*, is given in Figure 3. Note that we use the first and second arguments of the term *bak* to denote the program points of the first and second process respectively.

Clearly the variables in a constraint transition may be renamed freely because their scope is local to the transition. We thus say that a constraint transition is a *variant* of another if one is identical to the other when a renaming subsitution is performed. Further, we may *simplify* a constraint transition by renaming any one of its variables x by an expression

```
Process 1:                          Process 2:
  while (true) do                     while (true) do
⟨0⟩  x := y + 1                      ⟨0⟩  y := x + 1
⟨1⟩  await (x<y ∨ y=0)               ⟨1⟩  await (y<x ∨ x=0)
⟨2⟩  x := 0                          ⟨2⟩  y := 0
  end                                 end
```

Fig. 2. Two Process Bakery

$bak(0,p2,x,y) \mapsto bak(1,p2,x_1,y), x_1 = y+1.$
$bak(1,p2,x,y) \mapsto bak(2,p2,x,y), x < y \vee y = 0.$
$bak(2,p2,x,y) \mapsto bak(0,p2,x_1,y), x_1 = 0.$
$bak(p1,0,x,y) \mapsto bak(p1,1,x,y_1), y_1 = x+1.$
$bak(p1,1,x,y) \mapsto bak(p1,2,x,y), y < x \vee x = 0.$
$bak(p1,2,x,y) \mapsto bak(p1,0,x,y_1), y_1 = 0.$

Fig. 3. CTS of Two Process Bakery

y provided that $x = y$ in all groundings of the constraint transition. For example, we may simply state the last constraint transition in Figure 3 into

$$bak(p1,2,x,y) \mapsto bak(p1,0,x,0)$$

by replacing the variable y_1 in the original transition with 0.

The above formulation of program transitions is familiar in the literature for the purpose of defining a set of transitions. What is new, however, is how we use a CTS to define *symbolic* transition sequences, and thereon, the notion of a proof.

By similarity with logic programming, we use the term *goal* to denote a formula that can be subjected to an *unfolding process* in order to infer a logical consequence.

Definition 3 (Goal). *A query or* goal *of a CTS is of the form* $\langle p(k,\tilde{x}), \Psi(\tilde{x})\rangle$, *where* k *is a program point,* $\tilde{x}$ *is a sequence of variables over system states, and* Ψ *is a constraint over some or all of the variables* $\tilde{x}$*, and possibly some additional variables. The variables* $\tilde{x}$ *are called the* primary *variables of this goal, while any additional variable in* Ψ *is called an* auxiliary *variable of the goal.* ☐

Thus a goal is just like the conclusion of a constraint transition. We say the goal is a *start goal* if k is the start program point. Similarly, a goal is a *target goal* if k is the target program point. Running a start goal is tantamount to asking the question: which values of $\tilde{x}$ which satisfy $\tilde{\exists}\Psi(\tilde{x})$ will lead to a goal at the target point(s)? The idea is that we successively reduce one goal to another until the resulting goal is at a target point, and then inspect the results.

Next we define the meaning of proving a goal against a CTS.

Definition 4 (Proof Step, Sequence and Tree). *Let there be a CTS for* p*, and let* $\mathcal{G} = \langle p(k,\tilde{x}), \Psi\rangle$ *be a goal for this. A* proof step *from* $\mathcal{G}$ *is obtained via a variant* $p(k,\tilde{y}) \mapsto p(k_1,\tilde{y_1}), \Psi_1$ *of a transition in the CTS in which all the variables are fresh. The result is a goal of the form* $\langle p(k_1,\tilde{y_1}), \Psi, \tilde{x} = \tilde{y}, \Psi_1\rangle$*, providing that the constraints* $\Psi, \tilde{x} = \tilde{y}, \Psi_1$ *are satisfiable.*

A proof sequence *is a finite or infinite sequence of proof steps. A proof tree is defined from proof sequences in the obvious way. A tree is* complete *if every internal node representing a goal* $\mathcal{G}$ *is succeeded by nodes representing* every *goal obtainable in a proof step from* $\mathcal{G}$. ☐

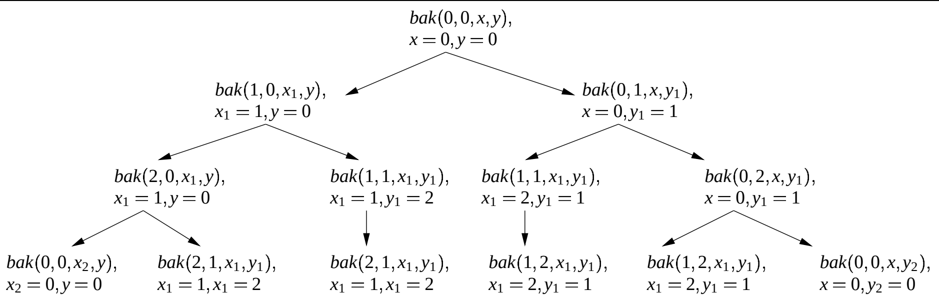
$bak(0,0,x,y)$, $x=0, y=0$
$bak(1,0,x_1,y)$, $x_1=1, y=0$
$bak(0,1,x,y_1)$, $x=0, y_1=1$
$bak(2,0,x_1,y)$, $x_1=1, y=0$
$bak(1,1,x_1,y_1)$, $x_1=1, y_1=2$
$bak(1,1,x_1,y_1)$, $x_1=2, y_1=1$
$bak(0,2,x,y_1)$, $x=0, y_1=1$
$bak(0,0,x_2,y)$, $x_2=0, y=0$
$bak(2,1,x_1,y_1)$, $x_1=1, x_1=2$
$bak(2,1,x_1,y_1)$, $x_1=1, x_1=2$
$bak(1,2,x_1,y_1)$, $x_1=2, y_1=1$
$bak(1,2,x_1,y_1)$, $x_1=2, y_1=1$
$bak(0,0,x,y_2)$, $x=0, y_2=0$

Fig. 4. Proof Tree of 2-Process Bakery Algorithm (Partially Shown)

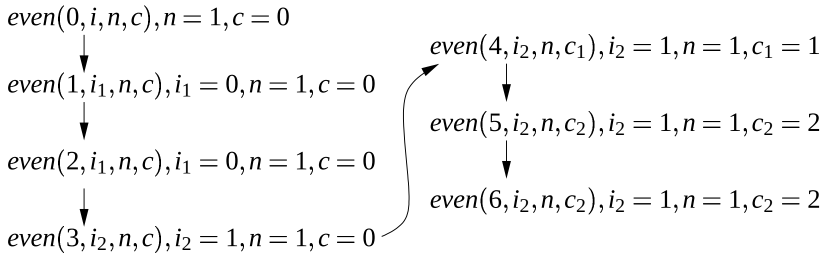
$even(0,i,n,c), n=1, c=0$
$even(1,i_1,n,c), i_1=0, n=1, c=0$
$even(2,i_1,n,c), i_1=0, n=1, c=0$
$even(3,i_2,n,c), i_2=1, n=1, c=0$
$even(4,i_2,n,c_1), i_2=1, n=1, c_1=1$
$even(5,i_2,n,c_2), i_2=1, n=1, c_2=2$
$even(6,i_2,n,c_2), i_2=1, n=1, c_2=2$

Fig. 5. Proof Tree of Even Counts Program

Consider again the CTS in Figure 1b, and we wish to prove $\{n=1\}p\{c=2\}$. There is in fact only one proof sequence from the start goal

$even(0,i,n,c), n=1, c=0.$

or equivalently, $even(0,i,1,0)$. This proof sequence is shown in Figure 5, and note that the counter, represented in the last goal by the variable c_2, has the value 2.

Definition 5 (Assertion). *Let p be a program with start variables $\tilde{x}$, and let Ψ be a constraint. Let $\tilde{x}^t$ denote a sequence of variables representing system states not appearing in p or Ψ. (These represent the target values of the system variables.) An assertion for p wrt to $\tilde{x}^t$ is of the form $p(k,\tilde{x}), \Psi \models \Psi_1(\tilde{x},\tilde{x}^t)$. In particular, when k is the start program point, we may abberviate the assertion using the notation $\{\Psi\}p\{\Psi_1\}$* □

It is intuitively clear what it means for an assertion to hold. That is, execution from every instance θ of $p(k,\tilde{x}), \Psi$ cannot lead to a target state where the property $\Psi_1(\tilde{x}\theta,\tilde{x}^t)$ is violated.

In the example above, we could prove the assertion $even(0,i,n,c) \models c^t = 2n$ where it is understood that the final variable c^t corresponds to the start variable c. Note that the last occurrence of n in the assertion means that we are comparing c^t with the initial and not final value of n (though in this example, the two are in fact the same).

We now state the essential property of proof sequences:

Theorem 1. *Let a CTS for p have the start point k and target point k^t, and let $\tilde{x}$ and $\tilde{x_1}$ each be sequences of variables over system states. The assertion $\{\Psi(\tilde{x})\}$ p $\{\Psi_1(\tilde{x}^t,\tilde{x})\}$ holds if for any goal of the form $p(k^t,\tilde{x_1})$, $\Psi_2(\tilde{x_1},\tilde{x})$ appearing in a proof sequence from the goal $p(k,\tilde{x})$, $\Psi(\tilde{x})$, the following holds: $\tilde{\exists}\Psi_2(\tilde{x_1},\tilde{x}) \models \tilde{\exists}\Psi_1(\tilde{x_1},\tilde{x})$* □

The above theorem provides the basis of a search method, and what remains is to provide a means to ensure termination of the search. Toward this end, we next define the concepts of *subsumption* and *coinduction* and which allow the (successful) termination of proof sequences. However, these are generally insufficient. In the next section, we present our version of *abstraction* whose purpose is to transform a proof sequence so that it is applicable to the termination criteria of subsumption and coinduction.

3.1 Subsumption. Consider a finite and complete proof tree from some start goal. A goal $\mathcal{G}$ in the tree is *subsumed* if there is a different path in the tree containing a goal $\mathcal{G}'$ such that $[\![\mathcal{G}]\!] \subseteq [\![\mathcal{G}']\!]$.

The principle here is simply memoization: one may terminate the expansion of a proof sequence while constructing a proof tree when encountering a subsumed goal.

3.2 Coinduction. The principle here is that, within one proof sequence, the proof obligation associated with the final goal may *assume* that the proof obligation of an ancestor goal has already been met. This can be formally explained as a principle of coinduction (see eg: Appendix B of [25]). Importantly, this simple form of coinduction does not require a base case nor a well-founded ordering.

We shall simply demonstrate this principle by example. Suppose we had the transition $p(0,x) \mapsto p(0,x'), x' = x+2$ and we wished to prove the assertion $p(0,x) \models even(x^t - x)$, that is, the difference between x and its final value is even. Consider the derivation step:

$$p(0,x) \models even(x^t - x)$$
$$p(0,x'), x' = x+2 \models even(x^t - x)$$

We may use, in the latter goal, the fact that the earlier goal satisfies the assertion. That is, we may reduce the obligaton of the latter goal to $even(x^t - x'), x' = x+2 \models even(x^t - x)$. It is now a simple matter of inferring whether this formula holds. In general practice, the application of coinduction testing is largely equivalent to testing if one goal is simply an instance of another.

4 Abstraction

In the literature on predicate abstraction, the abstract description is a specialized data structure, and the abstraction operation serves to propagate such a structure though a small program fragment (a contiguous group of assignments, or a test), and then obtaining another structure. The strength of this method is in the simplicity of using a finite set of predicates over the fixed number of program variables as a basis for the abstract description.

We choose to follow this method. However, our abstract description shall not be a distinguished data structure. In fact, our abstract description of a goal is itself a goal.

Definition 6 (Abstraction). *An abstraction $\mathcal{A}$ is applied to a goal. It is specified by a program point $pc(\mathcal{A})$, a sequence of variables $var(\mathcal{A})$ corresponding to a subset of the system variables, and finally, a finite set of constraints $pred(\mathcal{A})$ over $var(\mathcal{A})$, called the "predicates" of $\mathcal{A}$.*

Let $\mathcal{A}$ be an abstraction and G be a goal $p(k,\tilde{x}),\Psi$ where $k = pc(\mathcal{A})$. Let $\tilde{x_1}$ denote the subsequence of $\tilde{x}$ corresponding to the system variables $var(\mathcal{A})$. Let $\bar{x}$ denote the remaining subsequence of $\tilde{x}$. Without losing generality, we assume that $\tilde{x_1}$ is an initial subsequence of $\tilde{x}$, that is, $\tilde{x} = \tilde{x_1}, \bar{x}$. Then the abstraction $\mathcal{A}(G)$ of G by $\mathcal{A}$ is $p(k\tilde{Z}, \bar{x}), \Psi, \Psi_2[var(\mathcal{A}) \mapsto \tilde{Z}]$, *where $\tilde{Z}$ is a sequence of fresh variables renaming $\tilde{x_1}$, and Ψ_2 is the finite set of constraints* $\{\psi_2 \in pred(\mathcal{A}) : \Psi \models \psi_2[var(\mathcal{A}) \mapsto \tilde{x_1}]\}$ $\square$

For example, let $\mathcal{A}$ be such that $pc(\mathcal{A}) = 0$, $var(\mathcal{A}) = \{v_1\}$ and $pred(\mathcal{A}) = \{v_1 < 0, v_1 \geq 0\}$. That is, the first variable is to be abstracted into a negative or a nonnegative value. Let G be $p(0,[x_1,x_2,x_3]), x_1 = x_2, x_2 = 1$. Then the abstraction $\mathcal{A}(G)$ is a goal of the form $p(0,[Z,x_2,x_3]), x_1 = x_2, x_2 = 1, Z \geq 0$, which can be simplified into $p(0,[Z,x_2,x_3]), x_2 = 1, Z \geq 0$. Note that the orginal goal had ground instances $p(0,[1,1,n])$ for all n, while the abstracted goal has the instances $p(0,[m,1,n])$ for all n and all nonnegative m. Note that the second variable x_2 has not been abstracted even though it is tightly constrained to the first variable x_1. Note further that the value of x_3 is unchanged, that is, the abstraction would allow any constraint on x_3, had the example goal contained such a constraint, to be *propagated*.

Lemma 1. *Let $\mathcal{A}$ be an abstraction and G a goal. Then* $[\![G]\!] \subseteq [\![\mathcal{A}(G)]\!]$. $\square$

The critical point is that the abstraction of a goal has the *same format* as the goal itself. Thus an abstract goal has the expressive power of a regular goal, while yet containing a notion of abstraction that is sufficient to produce a finite-state effect. Once again, this is facilitated by the ability to reason about an unbounded number of variables.

Consider the “Bubble” program and its CTS in Figures 7(a) and 7(b), which is a simplified skeleton of the bubble sort algorithm (without arrays). Consider the subprogram corresponding to start point 2 and whose target point is 6, that is, we are considering the inner loop. Further suppose that the following assertion had already been proven:

$$bub(2,i,j,t,n) \models i^t = i, t^t = t + n - i - 1, n^t = n$$

that is, the subprogram increments t by $n - i - 1$ while preserving both i and n, but not j. Consider now a proof sequence for the goal $bub(0,i,j,t,n), n \geq 0$, where we want to prove that at program point $\langle 8 \rangle$, $t = (n^2 - n)/2$. The proof tree is depicted in Figure 6. The proof shows a combination of the use of intermittent abstraction and compositional proof:

- At point (A), we abstract the goal $bub(2,i_1,j,t_1,n), i_1 = 0, t_1 = 0, n > 1$ using the predicates $i < n - 1$ and $t = n \times i - (i^2 - i)/2$. Call this abstraction $\mathcal{A}$. Here the set of variables is $var(\mathcal{A}) = \{i,t\}$, hence both the variables i_1 and t_1 that correspond respectively to system variables i and t are renamed to fresh variables i_2, and t_2. Meanwhile, the variables j and n retain their original values.
- After performing the above abstraction, we reuse the proof of the inner loop above. Here we immediately move to program point $\langle 6 \rangle$, incrementing t with $n - i - 1$, and updating j to an unknown value. However, i and n retain their original values at $\langle 2 \rangle$.
- The result of the intermittent abstraction above is a coinductive proof.

5 The Whole Algorithm

We now summarize our proof method for an assertion

$$\{\Psi\} p \{\Psi_1\}$$

Suppose the start program point of p is k and the start variables of p are $\tilde{x}$. Then consider the start goal $p(k,\tilde{x}), \Psi$ and incrementally build a complete proof tree. For each path in the tree constructed so far leading to a goal $\mathcal{G}$ if:

- $bub(0,i,j,t,n), n \geq 0$
  - $bub(1,i_1,j,t_1,n), i_1 = 0, t_1 = 0, n \geq 0$
    - $bub(8,i_1,j,t_1,n), i_1 = 0, t_1 = 0, 0 \leq n \leq 1$ **(Satisfies** $t_1 = (n^2 - n)/2$**)**
    - $bub(2,i_1,j,t_1,n), i_1 = 0, t_1 = 0, n > 1$ (A)
      - $bub(2,i_2,j,t_2,n), i_2 < n-1, t_2 = n \times i_2 - (i_2^2 - i_2)/2$ **(Intermittent abstraction)**
        - $bub(6,i_2,j_1,t_3,n), i_2 < n-1, t_3 = n \times (i_2+1) - ((i_2+1)^2 - i_2 - 1)/2$ **(Proof Composition)**
          - $bub(7,i_3,j_1,t_3,n), i_3 < n, t_3 = n \times i_3 - (i_3^2 - i_3)/2$
            - $bub(8,i_3,j_1,t_3,n), i_3 = n-1, t_3 = n \times i_3 - (i_3^2 - i_3)/2$ **(Satisfies** $t_3 = (n^2 - n)/2$**)**
            - $bub(2,i_3,j_1,t_3,n), i_3 < n-1, t_3 = n \times i_3 - (i_3^2 - i_3)/2$ **(Coinduction using (A))**

Fig. 6. Compositional Proof

```
⟨0⟩ t := 0; i := 0
⟨1⟩ while (i < n-1) do
⟨2⟩   j := 0
⟨3⟩   while (j < n-i-1) do
⟨4⟩      j := j+1; t := t+1
⟨5⟩   end
⟨6⟩   i := i+1
⟨7⟩ end ⟨8⟩
```

(a)

$bub(0,i,j,t,n) \mapsto bub(1,i_1,j,t_1,n), i_1 = 0, t_1 = 0.$
$bub(1,i,j,t,n) \mapsto bub(8,i,j,t,n), i \geq n-1.$
$bub(1,i,j,t,n) \mapsto bub(2,i,j,t,n), i < n-1.$
$bub(2,i,j,t,n) \mapsto bub(3,i,j_1,t,n), j_1 = 0.$
$bub(3,i,j,t,n) \mapsto bub(6,i,j,t,n), j \geq n-i-1.$
$bub(3,i,j,t,n) \mapsto bub(4,i,j,t,n), j < n-i-1.$
$bub(4,i,j,t,n) \mapsto bub(5,i,j_1,t_1,n), j_1 = j+1, t_1 = t+1.$
$bub(5,i,j,t,n) \mapsto bub(6,i,j,t,n), j \geq n-i-1.$
$bub(5,i,j,t,n) \mapsto bub(4,i,j,t,n), j < n-i-1.$
$bub(6,i,j,t,n) \mapsto bub(7,i_1,j,t_1,n), i_1 = i+1.$
$bub(7,i,j,t,n) \mapsto bub(8,i,j,t,n), i \geq n-1.$
$bub(7,i,j,t,n) \mapsto bub(2,i,j,t,n), i < n-1.$

(b)

Fig. 7. Program "Bubble"

- $\mathcal{G}$ is either subsumed or is coinductive, then consider this path *closed*, ie: not to be expanded further;
- $\mathcal{G}$ is a goal on which an abstraction $\mathcal{A}$ is defined, replace $\mathcal{G}$ by $\mathcal{A}(\mathcal{G})$;
- $\mathcal{G}$ is a target goal, and if the constraints on the primary variables $\tilde{x_1}$ in $\mathcal{G}$ do *not* satisfy $\Psi\theta$, where θ renames the target variables in Ψ into $\tilde{x_1}$, terminate and return *false*.
- the expansion of the proof tree is no longer possible, terminate and return *true*.

Theorem 2. *If the above algorithm, applied to the assertion* $\{\Psi\}p\{\Psi_1\}$, *terminates and does not return false, then the assertion holds.* $\square$

6 CLP Technology

It is almost immediate that CTS is implementable in CLP. Given a CTS for p, we build a CLP program in the following way: (a) for every transition of the form $(k,\tilde{x}) \mapsto (k',\tilde{x'}), \Psi$ we use the CLP rule the clause $\mathtt{p}(k,\tilde{x}) :- \mathtt{p}(k',\tilde{x'}), \Psi$ (assuming that Ψ is in the constraint domain of the CLP implementation at hand); (b) for every terminal program point k, we use the CLP fact $\mathtt{p}(k,\_,\ldots,\_)$, where the number of anonymous variables is the same as the number of variables in $\tilde{x}$.

We see later that the key implementation challenge for a CLP system is the *incremental satisfiability* problem. Roughly stated, this is the problem of successively determining that a monotonically increasing sequence of constraints (interpreted as a conjunction) is satisfiable.

6.1 Exact Propagation is "CLP-Hard"

Here we informally demonstrate that the incremental satisfiability problem is reducible to the problem of analyzing a straight line path in a program. We will consider here constraints in the form of linear diophantine equations, i.e., multivariate polynomials over the integers. Without loss of generality, we assume each constraint is written in the form $X = Y + Z$ or $X = nY$ where n is an integer. Throughout this section, we denote by X, Y, Z logic variables, and by x, y, z their corresponding program variables, respectively.

Suppose we already have a sequence of constraints $\Psi_0, \cdots, \Psi_i$ and a corresponding path in the program's control flow.

Suppose we add a new constraint $\Psi_{i+1} = (X = Y + Z)$. Then, if one of these variables, say Y, is new, we add the assignment $y := x - z$ where y is a new variable created to correspond to Y. The remaining variables x and z are each either new, or are the corresponding variables to X and Z, respectively. If however all of X, Y and Z are not new, then add the statement `if` $(x = y + z)$ `...` . Hereafter we pursue the `then` branch of this `if` statement.

Similarly, suppose the new constraint were of the form $X = nY$. Again, if x is new, we simply add the assignment $x := n * y$ where x is newly created to correspond to X. Otherwise, add the statement `if (x = n * y) ...` to the path, and again, we now pursue the `then` branch of this `if` statement.

Clearly an exact analysis of the path we have constructed leading to a successful traversal required, incrementally, the solving of the constraint sequence $\Psi_0, \cdots, \Psi_n$.

6.2 Key Elements of CLP Systems

A CLP system attempts to find answers to an initial goal $\mathcal{G}$ by searching for valid substitutions of its variables, in a depth-first manner. Each path in the search tree in fact involves the solving of an incremental satisfiability problem. Along the way, unsatisfiability of the constraints at hand would entail backtracking.

The key issue in CLP is the incremental satisfiability problem, as mentioned above. A standard approach is as follows. Given that the sequence of constraints $\Psi_0, \ldots, \Psi_i$ has been determined to be satisfiable, represent this fact in a *solved form*. Essentially, this means that when a new constraint Ψ_{i+1} is encountered, the solved form can be combined efficiently with Ψ_{i+1} in order to determine the satisfiability of the new conjunction of constraints.

This method essentially requires a representation of the *projection* of a set of constraints onto certain variables. Consider, for example, the set $x_0 = 0, x_1 = x_1 + 1, x_2 = x_1 + 1, \cdots, x_i = x_{i-1} + 1$. Assuming that the new constraint would only involve the variable x_i (and this happens vastly often), we desire a representation of $x_i = i$. This projection problem is well studied in CLP systems [21]. In the system CLP($\mathcal{R}$) [22] for example, various adaptations of the Fourier-Motzkin algorithm were implemented for projection in Herbrand and linear arithmetic constraints.

We finally mention another important optimization in CLP: *tail recursion*. This technique uses the same space in the procedure call stack for recursive calls. Amongst other

```
int main()                              int main()
{   int i=0, j, x=0;                    {   int i=0, j, x=0;
    while (i<7) {                           while (i<50) {
        j=0;                                    i++; j=0;
        while (j<7) { x++; j++; }               while (j<10) { x++; j++; }
        i++; }                                  while (x>i) { x--; }}
    if (x>49) { ERROR: }}                   if (x<50) { ERROR: }}
```

(a) **(b)**

Fig. 8. Programs with Loop

benefits, this technique allows for a potentially unbounded number of recursive calls. Tail recursion is particurly relevant in our context because the recursive calls arising from the CTS of programs are often tail-recursive.

The CLP($\mathcal{R}$) system that we use to implement our prototype has been engineered to handle constraints and auxiliary variables efficiently using the above techniques.

7 Experiments

7.1 Exact Runs

We start with an experiment which shows that concrete execution can potentially be less costly than abstract execution. To that end, we compare the timing of concrete execution using our CLP-based implementation and a predicate abstraction-based model checker. We run a simple looping program, whose C code is shown in Figure 8 (a). First, we have BLAST generate all the 100 predicates it requires. We then re-run BLAST by providing these predicates. BLAST took 22.06 seconds to explore the state space. On the same machine, and without any abstraction, our verification engine took only 0.02 seconds. For comparison, SPIN model checker [20] executes the same program written in PROMELA in less than 0.01 seconds. Note that for all our experiments, we use a Pentium 4 2.8 GHz system with 512 MB RAM running GNU/Linux 2.4.22.

	Time (in Seconds)			
	CLP with Tabling		ESC/Java 2	
	`x==0`	—	`x==0`	—
Non-Looping	2.45	2.47	9.89	9.68
Looping	22.05	21.95	1.00	1.00

Table 1. Timing Comparison with ESC/Java

Next, consider the synthetic program consisting of an initial assignment x `:= 0` followed by 1000 increments to x, with the objective of proving that $x = 1000$ at the end. Consider also an alternative version where the program contains only a single loop which increments its counter x 1000 times. We input these two programs to our program verifier, without using abstraction, and to ESC/Java 2 as well. The results are shown in Table 1. For both our verifier and ESC/Java 2 we run both with x initialized to 0 and not initialized, hopefully forcing symbolic execution.

Table 1 shows that our verifier runs faster for the non-looping version. However, there is a noticeable slowdown in the looping version for our implementation. This is caused by the fact that in our implementation of coinductive tabling, subsumption check is done based on similarity of program point. Therefore, when a program point inside a loop is visited for the i-th time, there are $i-1$ subsumption checks to be performed. This results in a total of about 500,000 subsumption checks for the looping program. In comparison, the non-looping version requires only 1,000 subsumption checks. However, our implementation is currently at a prototype stage and our tabling mechanism is not implemented in the most efficient way. For the looping version, ESC/Java 2 employs a weakest precondition propagation calculus; since the program is very small, with a straightforward invariant (just the loop condition), the computation is very fast. Table 1 also shows that there is almost no difference between having `x` initialized to 0 or not.

7.2 Experiments Using Abstraction

Next we show an example that demonstrates that the intermittent approach requires fewer predicates. Let us consider a second looping program written in C, shown in Figure 8 (b). The program's postcondition can be proven by providing an invariant $\texttt{x=i} \wedge \texttt{i<50}$ before the first statement of the loop body of the outer while loop. For predicate abstraction, we use the following predicates `x=i`, `i<50`, and respectively their negations $\texttt{x}\neq\texttt{i}$, $\texttt{i}\geq\texttt{50}$ for that program point to our verifier. The proof process finishes in less than 0.01 seconds. If we do not provide an abstract domain, the verification process finishes in 20.34 seconds. Here intermittent predicate abstraction requires fewer predicates: We also run the same program with BLAST and provide the predicates `x=i` and `i<50` (BLAST would automatically also consider their negations). BLAST finishes in 1.33 seconds, and in addition, it also produces 23 other predicates through refinements. Running it again with all these predicates given, BLAST finishes in 0.28 seconds.

```
        while (true) do
⟨0⟩         $x_i$ := $max(x_{j\neq i})$ + 1
⟨1⟩         await $(\forall j : j \neq i \rightarrow x_i < x_j \vee x_j = 0)$
⟨2⟩         $x_i$ := 0
        end
```

Fig. 9. Bakery Algorithm Peudocode for

Further, we also tried our proof method on a version of the bakery mutual exclusion algorithm. We need abstraction since the bakery algorithm is an infinite-state program. The pseudocode for process i is shown in Figure 9. Here we would like to verify mutual exclusion, that is, no two processes are in the critical section (program point ⟨2⟩) at the same time. Our version of the bakery algorithm is a concurrent program with asynchronous composition of processes. Nondeterminism due to concurrency can be encoded using nondeterministic choice. We encode the algorithm for 2, 3 and 4 processes in BLAST, where nondeterministic choice is implemented in using the special variable `__BLAST_NONDET` which has a nondeterministic value. When N is the number of processes, each of the program has the N variables $\texttt{pc}_i$, where $1 \leq i \leq N$, each denoting the program point of process i. $\texttt{pc}_i$ can only take a value from $\{0,1,2\}$. and also N variables $\texttt{x}_i$, each denoting the "ticket number" of a process. We also translate the BLAST code into CTS.

In our experiments, we attempt to verify mutual exclusion property, that is, no two processes can be in the critical section at the same time. Here we perform 3 sets of runs, each consisting of runs with 2, 3 and 4 processes. In all 3 sets, we use a basic set of predicates: $\texttt{x}_i\texttt{=0}$, $\texttt{x}_i\geq\texttt{0}$, $\texttt{pc}_i\texttt{=0}$, $\texttt{pc}_i\texttt{=1}$, $\texttt{pc}_i\texttt{=2}$, where $i = 1,\ldots,N$ and N the number of processes, and also their negations.

- **Set 1: Use of predicate abstraction at every state with full predicate set**. We perform abstraction at every state encountered during search. In addition to the basic predicates, we also require the predicates shown in Table 2 (a) (and their negations) to avoid spurious counterexamples.

Bakery-2	`x1<x2`
Bakery-3	`x1<x2, x1<x3, x2<x3`
Bakery-4	`x1<x2, x1<x3, x1<x4` `x2<x3, x2<x4, x3<x4`

(a) Additional Predicates

	Time (in Seconds)			
	CLP with Tabling			BLAST
	Set 1	Set 2	Set 3	
Bakery-2	0.02	0.01	<0.01	0.17
Bakery-3	0.83	0.14	0.09	2.38
Bakery-4	131.11	8.85	5.02	78.47

(b) Timing Constraints

Table 2. Results of Experiments Using Abstraction

- **Set 2: Intermittent predicate abstraction with full predicate set**. We use intermittent abstraction on our prototype implementation. We abstract only when for some process *i*, $\texttt{pc}_i$=1 holds. The set of predicates is as in the first set.
- **Set 3: Intermittent predicate abstraction with reduced predicate set**. We use intermittent abstraction on our tabled CLP system. Wee only abstract whenever there are $N-1$ processes at program point 0 (in the 2-process sequential version this means either `pc1=0` or `pc2=0`). For a *N*-process bakery algorithm, we only need the basic predicates and their negations without the additional predicates shown in Table 2 (a).

We have also compared our results with BLAST. We supplied the same set of predicates that we used in the first and second sets to BLAST. Again, in BLAST we do not have to specify their negations explicitly. Interestingly, for 4-process bakery algortihm BLAST requires even more predicates to avoid refinement, which are `x1=x3+1`, `x2=x3+1`, `x1=x2+1`, $1 \leq x4$, $x1 \leq x3$, $x2 \leq x3$ and $x1 \leq x2$. We suspect this is due to the fact that precision in predicate abstraction-based state-space traversal depends on the power of the underlying theorem prover. We have BLAST generate these additional predicates it needs in a pre-run, and then run BLAST using them. Here since we do not run BLAST with refinement, as the *lazy abstraction* technique [19] has no effect, and BLAST uses all the supplied predicates to represent any abstract state.

For these problems, using our intermittent abstraction with CLP tabling is also markedly faster than both full predicate abstraction with CLP and BLAST. We show our timing results in Table 2 (b) (smallest recorded time of 3 runs each).

The first set and BLAST both run with abstraction at every visited state. The timing difference between them and second and third sets shows that performing abstraction at every visited state is expensive. The third set shows further gain over the second when we understand some intricacies of the system.

Acknowledgement: We thank Ranjit Jhala for help with BLAST.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *15th PLDI*, pages 203–213. ACM Press, May 2001. SIGPLAN Notices 36(5).
2. T. Ball, T. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. *ACM Transactions on Programming Languages and Systems*, 27(2):314–343, 2005.
3. B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. M. Noz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq proof assistant reference manual—version v6.1. Technical Report 0203, INRIA, 1997.
4. A. Bossi, editor. *LOPSTR '99*, volume 1817 of *LNCS*. Springer, 2000.

5. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004.
6. D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS 2004*, volume 3362 of *LNCS*, pages 108–128. Springer, 2005.
7. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *PLILP '92,*, LNCS 631.
8. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In N. Halbwachs and D. Peled, editors, *11th CAV*, number 1633 in LNCS, pages 160–171. Springer, 1999.
9. G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, *5th TACAS*, volume 1579 of *LNCS*, pages 223–239. Springer, 1999.
10. X. Du, C. R. Ramakrishnan, and S. A. Smolka. Tabled resolution + constraints: A recipe for model checking real-time systems. In *21st RTSS*. IEEE Computer Society Press, 2000.
11. L. Burdy et. al. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003*, volume 2805 of *LNCS*.
12. Y. S. Ramakrishna et. al. Efficient model checking using tabled resolution. In Grumberg [16].
13. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite-state systems by specializing constraint logic programs. In M. Leuschel, A. Podelski, C. R. Ramakrishnan, and U. Ultes-Nitsche, editors, *2nd VCL*, pages 85–96, 2001.
14. L. Fribourg. Constraint logic programming applied to model checking. In Bossi [4], pages 30–41.
15. S. Graf and H. Saïdi. Construction of abstract state graphs of infinite systems with PVS. In Grumberg [16], pages 72–83.
16. O. Grumberg, editor. *CAV '97, Proceedings*, volume 1254 of *LNCS*. Springer, 1997.
17. G. Gupta and E. Pontelli. A constraint-based approach for specification and verification of real-time systems. In *18th RTSS*, pages 230–239. IEEE Computer Society Press, 1997.
18. J. Harrison. HOL light: A tutorial introduction. In M. K. Srivas and A. J. Camilleri, editors, *1st FMCAD*, volume 1166 of *LNCS*, pages 265–269. Springer, 1996.
19. T. A. Henzinger, R. Jhala, and R. Majumdar. Lazy abstraction. In *29th POPL*, pages 58–70. ACM Press, 2002. SIGPLAN Notices 37(1).
20. G. J. Holzmann. *The* SPIN *Model Checker: Primer and Reference Manual*. Add.-Wesley, 2003.
21. J. Jaffar, M. Maher, P. Stuckey, and R. Yap. Projecting CLP($\mathcal{R}$) constraints. In *New Generation Computing*, volume 11, pages 449–469. Ohmsha and Springer-Verlag, 1993.
22. J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS*, 14(3):339–395, 1992.
23. M. Leuschel and T. Massart. Infinite-state model checking by abstract interpretation and program specialization. In Bossi [4].
24. C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *J. Log. and Alg. Prog.*, 58(1–2):89–106, 2004.
25. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
26. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model checking. In ed. J. W. Lloyd et. al., editor, *1st CL*, volume 1861 of *LNCS*, pages 384–398. Springer, 2000.
27. S. Owre, N. Shankar, and J. Rushby. PVS: A prototype verification system. In D. Kapur, editor, *11th CADE*, volume 607 of *LNCS*, pages 748–752. Springer, 1992.