Relative Safety

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. A safety property restricts the set of reachable states. In this paper, we introduce a notion of *relative safety* which states that certain program states are reachable provided certain other states are. A key, but not exclusive, application of this method is in representing *symmetry* in a program. Here, we show that relative safety generalizes the programs that are presently accommodated by existing methods for symmetry. Finally, we provide a practical algorithm for proving relative safety.

1 Introduction

A safety property restricts the set of reachable states. Let $[\![P]\!]$ denote the collecting semantics of a program P with variables $\tilde{X}$. Thus each sequence $\tilde{x}$ of variable values in $[\![P]\!]$ represents a reachable state. A safety property may be simply written as a constraint Ψ over the variables $\tilde{X}$. For example, the safety property $X+Y<9$ states that in all reachable states, the values of the program variables X and Y sum to less than 9. If we let the predicate $p(\tilde{x})$ be *true* just in case the sequence of values of program variables $\tilde{x}$ is in $[\![P]\!]$, then a safety property may be written in the form $p(\tilde{X}) \models \Psi$, for example, $p(X,Y) \models X+Y<9$.

In this paper, we introduce the notion of *relative safety*. Briefly and informally, this asserts that a certain state is reachable provided a certain other state is reachable. Note that this does not mean that these two states share a computation path. Specifically, consider the specification of states in the form $p(\tilde{X}) \wedge \Psi$. That is, we use the constraint Ψ to identify the set of solutions of Ψ which correspond to reachable states. Then our notion of relative safety simply relates two of these specifications in the following way: $p(\tilde{X}) \wedge \Psi \models p(\tilde{Y}) \wedge \Psi'$ where Ψ and Ψ' are constraints over $\tilde{X}, \tilde{Y}$. For example, $p(X_1,X_2) \models p(Y_1,Y_2) \wedge X_1 = Y_2 \wedge X_2 = Y_1$ (or more succinctly, $p(X_1,X_2) \models p(X_2,X_1)$) asserts that if the state (α,β) is reachable, then so is (β,α), for all values α and β. In other words, the observable values of the two program variables commute.

Relative safety can specify powerful structural properties of programs. The driving application we consider in this paper is that of verification with symmetry reduction. Symmetry has been widely employed for minimizing the search space in program verification. It is a reduction technique employed in Murφ [13] and SMC [21] model checkers among many others. Symmetry is often defined using automorphisms π on the symmetric objects. These induce an equivalence relation between program states. Efficiency in state exploration is hence achieved by only checking the representatives of the equivalence classes.

Let us take as an example a concurrent program with two almost identical processes, where process 1 updates the variables PC_1 and X_1, and process 2, PC_2 and

X_2. Here PC_1 and PC_2 are process 1 and 2's program counters, respectively. Let us consider $(\alpha,\beta,\gamma,\delta)$ to be values of (PC_1,PC_2,X_1,X_2). Classic symmetry "exchanges" process 1 and 2, that is, $\pi((\alpha,\beta,\gamma,\delta)) = (\beta,\alpha,\delta,\gamma)$. A necessary condition for π to be an automorphism is that whenever $\tilde{x}$ is a reachable state, so is $\pi(\tilde{x})$. Such a relation between $\tilde{x}$ and $\pi(\tilde{x})$ can be logically represented as the relative safety assertion $p(PC_1,PC_2,X_1,X_2) \models p(PC_2,PC_1,X_2,X_1)$ where the predicate p, once again, represents the reachable states of the program. Below we show many more examples of symmetry, including ones that are not covered by existing techniques.

The main technical part of this paper is a proof method. In its most basic form, the method to prove the assertion $G_1 \models G_2$ checks that the set of states represented by the symbolic formula G_2 is reachable, whenever the set G_1 is reachable. This is done by the basic operation of "backward unfolding" the program's transition relation. A key element in our algorithm is the use of the principle of coinduction which is critical for termination of the unfolding process.

The paper is organized as follows. We discuss some related work in Section 2. We then formalize the program semantics and the proof method in the framework of *Constraint Logic Programming* (*CLP*) [14], for two main reasons. First, the logical framework of CLP is eminently suitable for the represntation of our concept of relative safety, and second, the established implementation technology of CLP systems allow us to perform unfolding operations efficiently. We introduce some preliminary CLP concepts in Section 3. Relative safety is then formally defined in Section 4. Here, we show via several examples, novel ways to realize symmetry. In addition to these, we will also show a non-symmetry example. Section 5 formally presents our algorithm. Finally, in Section 6, we demonstrate the use of our prototype implementation on some classes of programs in order to show the practical potential of our algorithm.

2 Related Work

Existing approaches define symmetry on syntactic considerations. In contrast, our notion of relative safety is based on semantics. An advantage is more flexibility in specifying a wide range of symmetry-like properties, including many that would not be considered a symmetry property by the existing methods. One example, shown later, is a mutual exclusion algorithm with priority between processes. We can handle a wider range than [7, 20], for example. Importantly, relative safety goes far beyond symmetry (and below, we demonstrate the property of serializability).

In more detail, symmetry is often defined as a transition-preserving equivalence [8, 3, 13, 9, 20], where an automorphism π, other than being a bijection on the reachable states, also satisfies that $(\tilde{x},\tilde{x}')$ is a transition iff $(\pi(\tilde{x}),\pi(\tilde{x}'))$ is. Another notion of equivalence used is bisimilarity [7], replacing the second condition with bisimilarity on the state graph. These stronger equivalences allows for the handling of larger class of properties beyond safety such as CTL* properties. However, stronger equivalence also means less freedom in handling symmetries on the collecting semantics, which we exploit further in this paper.

In [20], while still defining symmetry as transition-preserving equivalence, they attempt to handle systems which state graphs are not fully symmetric. The approach transforms the state graph into a fully symmetric one, while keeping annotation for

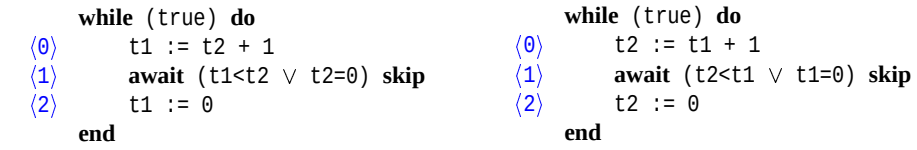

Fig. 1. Bakery-2

each transition that has no correspondence in the original state graph. The graph with full symmetry is then reduced by equating automorphic states. This work is the most general and can reduce the state graph of even totally asymmetric programs, however, its application is limited to programs with syntacticly specified static transition priority.

Similar to the work of [20], prior works infer symmetry based on syntactic conditions, such as concurrent program with identical processes or syntactic restrictions on program statements and variable usage. These also include the *scalarset* approach of Murϕ [13], and the limitation to permutation of process identifiers in SMC model checker [21]. In contrast, our approach to prove symmetry semanticly for each program enables us to treat more programs where the semantics is symmetric although the syntax is not.

An application of our symmetry proof method has been demonstrated in the context of timed automata verification [16]. This paper presents a generalization and automation of the method.

There have been many works in the area of verification using CLP (see [11] for a non-exhaustive survey), partly because it is natural to express transition relations as CLP rules. Due to its ability in handling constraints, CLP has been notably used in verification of infinite-state systems [5, 10, 12, 17], although results for finite-state systems are also available [18, 19]. None of these works, however, deal with relative safety.

3 CLP Representation of Programs

We start by stipulating that each process in a concurrent program has the usual syntax of a deterministic imperative language, and communication occurs via shared variables. We also have a blocking primitive **await** (**b**) **s** where **b** is a boolean expression and **s** a program statement, which can be executed only when **b** holds. A *program* is a collection of a fixed number of processes. We provide the 2-process bakery algorithm in Figure 1 as an example. We display program points in angle brackets.

We now introduce *CLP programs*. CLP programs have a *universe of discourse* $\mathcal{D}$ which is a set of terms, integers, and arrays of integers. A *constraint* is written using a language of functions and relations. They are used in two ways: in the base programming language to describe expressions and conditionals, and in user assertions, defined below. In this paper, we will not define the constraint language explicitly, but invent them on demand in accordance with our examples. Thus the terms of our CLP programs include the function symbols of the constraint language.

An *atom*, is as usual, of the form $p(\tilde{t})$ where p is a user-defined predicate symbol and the $\tilde{t}$ a tuple of terms. The set $\{p(\tilde{d})\}$ where p ranges over the predicates and $\tilde{d}$ ranges over the tuples in $\mathcal{D}$ is called the *domain base* $\mathcal{B}$ of our CLP programs.

$p([0,0],T_1,T_2) \leftarrow T_1 = 0, T_2 = 0.$ % init
$p([1,P_2],T_1',T_2) \leftarrow p([0,P_2],T_1,T_2), T_1' = T_2 + 1.$ % r1
$p([2,P_2],T_1,T_2) \leftarrow p([1,P_2],T_1,T_2), (T_1 < T_2 \vee T_2 = 0).$ % e1
$p([0,P_2],T_1',T_2) \leftarrow p([2,P_2],T_1,T_2), T_1' = 0.$ % x1
$p([P_1,1],T_1,T_2') \leftarrow p([P_1,0],T_1,T_2), T_2' = T_1 + 1.$ % r2
$p([P_1,2],T_1,T_2) \leftarrow p([P_1,1],T_1,T_2), (T_2 < T_1 \vee T_1 = 0).$ % e2
$p([P_1,0],T_1,T_2') \leftarrow p([P_1,2],T_1,T_2), T_2' = 0.$ % x2

Fig. 2. CLP Representation of Bakery-2

Now, a CLP program is a set of *rules*. A rule is an implication of the form $A \leftarrow \tilde{B}, \phi$ where the atom A is the *head* of the rule, and the sequence of atoms $\tilde{B}$ and the constraint ϕ constitute the *body* of the rule. We say that a rule is a (constrained) *fact* if $\tilde{B}$ is the empty sequence.

Translating a user program P_0 into an appropriate CLP program P is in fact intuitively straightforward; we thus provide only an informal outline here. Our CLP rules corresponding to a transition of the program will be of the form

$$p(PC', X_1', X_2', \ldots, X_n') \leftarrow p(PC, X_1, X_2, \ldots, X_n), \phi.$$

Here, PC is a list representing the program counters in the k processes of P_0 before the transition. Its primed counterpart PC' represents the list after the transition. $X_1, X_2, \ldots, X_n$ and their primed counterparts represent the variables in P_0 before and after the transition, while ϕ is a constraint on all the variables. Note that as in the above rule, throughout this paper we often use a comma in place of $\wedge$ to denote conjunction. The above rule depicts a transition from rhs to lhs .

Example 1 (Bakery-2). Consider our 2-process bakery algorithm in Figure 1. Note that the point $\langle 2 \rangle$ indicates the critical section, and initially, $\mathtt{t1} = \mathtt{t2} = 0$. The CLP program in Figure 2 (the parts preceded by % are comments) is in fact its CLP representation. x

The semantics of a CLP program is based on the concept of ground instances. A *ground instance* of a constraint ϕ is obtained by instantiating the variables therein from $\mathcal{D}$, and the result is true or false. We write this as $\phi\sigma$ [14] where $\sigma : var(\phi) \mapsto \mathcal{D}$ a *grounding*. Similarly, a *ground instance* of an atom or rule is obtained by instantiating variables therein with values in $\mathcal{B}$ using a grounding σ. Now consider the fixpoint operator $T_P : 2^{\mathcal{B}} \mapsto 2^{\mathcal{B}}$ for a CLP program P defined as follows: a ground atom $A\sigma$ is in $T_P(S)$ if $A\sigma \in S$ or there is a ground instance $(A \leftarrow \tilde{B}, \phi)\sigma$ of a rule $A \leftarrow \tilde{B}, \phi$ in P such that $\tilde{B}\sigma \subseteq S$ and $\phi\sigma$ is true. A basic theorem of CLP is that the least fixpoint of T_P is the least model of P, and this is also equal to the set of ground atoms. We denote this set by $[\![P]\!]$. A ground instance $A\sigma$ is true iff $A\sigma \in [\![P]\!]$. Similarly, a ground instance $(\tilde{B}, \phi)\sigma$ of a goal is true iff $\tilde{B}\sigma \subseteq [\![P]\!]$ and $\phi\sigma$ is true. We denote the set of true ground instances of a goal G by $[\![G]\!]$.

In general, where P is the CLP representation of P_0, we have that the collecting semantics of P_0 is characterized by $[\![P]\!]$.

4 Relative Safety

We now present an assertion language to express relative safety property, and demonstrate its expressive power for program reasoning. We start with a definition of a *constraint state*.

Definition 1 (Constraint State). *A* constraint state *is a goal in the form* $p(PC, X_1, \ldots, X_n), \phi$ *where* $PC, X_1, \ldots, X_n$ *represent the list of program counters and program variables, and* ϕ *is a constraint on the variables.*

Now let G^L be a constraint state and G^R either constraint or constraint state. Let $\tilde{X} = var(G^L) \cup var(G^R)$.

Definition 2 (Relative Safety). *A* relative safety assertion *is of the form* $G^L \models G^R$. *Its meaning is* $\forall \tilde{X} : G^L \rightarrow G^R$ *that is, for each grounding* σ *such that* $G^L\sigma \in [\![G^L]\!]$, $G^R\sigma \in [\![G^R]\!]$.

Intuitively, a relative safety assertion specifies that certain states are reachable only if certain other states are.

Here we start with a traditional safety property, generally of the form:

$$p(PC, \tilde{X}),\ \phi \ \models\ \phi'$$

where ϕ and ϕ' are constraints on the program counter array *PC* and program variables $\tilde{X}$. For example, in the Bakery-2 program, the following assertions specify mutual exclusion.

$p([P_1, P_2], T_1, T_2) \models P_1 \neq 2 \wedge P_2 \neq 2,$ or, $p([2,2], T_1, T_2) \models \mathit{false}$

Now consider a relative safety assertion, stating symmetry for Bakery-2:

$$p([P_1, P_2], T_1, T_2) \models p([P_2, P_1], T_2, T_1).$$

Note that an automorphism must be included in a group with the composition of automorphisms as its operator [23]. Such a group is known as an *automorphism group*. Our idea is to use a set of relative safety assertions to specify possible automorphisms on reachable states. Note that a single relative safety assertion in general only describes a partial mapping, while an automorphism is total. In general we need a set of assertions to describe a total mapping π. Moreover, equivalence between states is obtained by also proving a complete set of assertions which represent the mappings in an automorphism group. This would include inverses, which proof is often straightforward. Suppose that $map(G^L \models G^R)$ is the mapping represented by the assertion $G^L \models G^R$. Now, as an example, the above symmetry assertion for Bakery-2 characterizes an automorphism group *Aut* on the collecting semantics as follows:

- We include the obvious $map(p([P_1, P_2], T_1, T_2) \models p([P_1, P_2], T_1, T_2))$ in *Aut* satisfying the existence of identity.
- By simple renaming $\{P_1 \mapsto P_2, P_2 \mapsto P_1, T_1 \mapsto T_2, T_2 \mapsto T_1\}$ on the above assertion, the reverse $map(p([P_2, P_1], T_2, T_1) \models p([P_1, P_2], T_1, T_2))$ is in *Aut* satisfying the existence of inverse.
- It is straightforward to show that if $map(G_1 \models G_2) \in Aut$ and $map(G_2 \models G_3) \in Aut$ then $map(G_1 \models G_3) \in Aut$.

We will prove the assertion later in Section 5. We now proceed with several examples.

Example 2 (Rotational Symmetry). Next we demonstrate *rotational* symmetry in the solution of N dining philosophers' problem using $N-1$ tickets. For simplicity, we assume there are N=3 philosophers having ids 1, 2 and 3, and there are 3 forks represented as boolean array `f`, where `f[1]`, `f[2]`, `f[3]` are forks between philosopher 3 and 1, 1

```
    while (true) do                          while (true) do
⟨0⟩     await (x2 = 0) x1 := 1       ⟨0⟩     x2 := 1
⟨1⟩     skip                         ⟨1⟩     await (x1 = 0) skip
⟨2⟩     x1 := 0                      ⟨2⟩     x2 := 0
    end                                      end
```

Fig. 3. Priority Mutual Exclusion

$p([0,0],0,0).$

$p([1,P_2],1,X_2) \leftarrow p([0,P_2],X_1,X_2), X_2 = 0.$

$p([2,P_2],X_1,X_2) \leftarrow p([1,P_2],X_1,X_2).$

$p([0,P_2],0,X_2) \leftarrow p([2,P_2],X_1,X_2).$

$p([P_1,1],X_1,1) \leftarrow p([P_1,0],X_1,X_2).$

$p([P_1,2],X_1,X_2) \leftarrow p([P_1,1],X_1,X_2), X_1 = 0.$

$p([P_1,0],X_1,0) \leftarrow p([P_1,2],X_1,X_2).$

Fig. 4. CLP Representation of Priority Mutual Exclusion

and 2, and 2 and 3, respectively. Initially the ticket number `t`=2. To save space, we do not show the actual code. For our purpose it is suffice to demonstrate the rotational symmetry as the assertion:

$$p([P_1,P_2,P_3],F_1,F_2,F_3,T) \models p([P_3,P_1,P_2],F_3,F_1,F_2,T),$$

where P_i denotes the program point of philosopher i, F_1, F_2 and F_3 are the values of `f[`i`]`, $1 \leq i \leq 3$, respectively, and T is the number of tickets left. The above assertion specifies a cyclic shift. For this example, arbitrary transposition does not result in automorphism.

Example 3 (Permutation of Variable-Value Pair). In [16] we discussed a timed automata version of Fischer's algorithm, a timing-based mutual exclusion algorithm. The pseudocode can be found in [1] and is not presented here to save space. The algorithm uses a global variable `k` whose value is the process identifier of the process that is about to enter the critical section. This is translated into a variable K in our CLP representation (also not shown here). Since the example uses timing, our CLP representation for the 2-process version uses the variables T_1 and $T2$, denoting the running time of each process. Our symmetry assertion here is

$$p([P_1,P_2],T_1,T_2,K) \models p([P_2,P_1],T_2,T_1,K'),\phi,$$

where ϕ constrains (K,K') to $(0,0),(1,2)$ or $(2,1)$. This is called *permutation of variable-value pair* [20] since it maps the value of a variable onto a new one without exchanging it with another variable. This is not covered by some previous approaches such as [13, 21].

Example 4 (Priority Mutual Exclusion). We can also express the kind of "approximate" symmetry, as exemplified by the simple 2-process priority mutual exclusion in Figure 3. Each process has ⟨2⟩ as the critical section. Initially, the values of both `x1` and `x2` are 0. We show the CLP representation in Figure 4. This example is semantically similar to the asymmetric readers-writers in [6] and the priority mutual exclusion in [20]. Although the state graph of the program is not symmetric, the state space, i.e. the set of nodes in the state graph, is, and knowing this is already useful to prove safety properties such as mutual exclusion. We can represent the symmetry on the state space simply as:

$$p([P_1,P_2],X_1,X_2) \models p([P_2,P_1],X_2,X_1).$$

It is not immediately obvious that the program is symmetric based on syntactic observation alone.

```
    while (true) do
⟨0⟩     x1:=1
⟨1⟩     await(x2<3) skip
⟨2⟩     x1:=3
⟨3⟩     if (x2=1) do
⟨4⟩         x1:=2
⟨5⟩         await(x2=4) skip
        end
⟨6⟩     x1:=4
⟨7⟩     skip
⟨8⟩     await(x2<2∨x2>3) skip
⟨9⟩     x1:=0
    end
```

```
    while (true) do
⟨0⟩     x2:=1
⟨1⟩     await(x1<3) skip
⟨2⟩     x2:=3
⟨3⟩     if (x1=1) do
⟨4⟩         x2:=2
⟨5⟩         await(x1=4) skip
        end
⟨6⟩     x2:=4
⟨7⟩     await(x1<2) skip
⟨8⟩     skip
⟨9⟩     x2:=0
    end
```

Fig. 5. 2-Process Szymanski's Algorithm

$p([0,0],0,0)$. % Initial State
% Rules for Process 1
$p([1,P_2],1,X_2) \leftarrow p([0,P_2],X_1,X_2)$.
$p([2,P_2],X_1,X_2) \leftarrow p([1,P_2],X_1,X_2), X_2 < 3$.
$p([3,P_2],3,X_2) \leftarrow p([2,P_2],X_1,X_2)$.
$p([4,P_2],X_1,X_2) \leftarrow p([3,P_2],X_1,X_2), X_2 = 1$.
$p([5,P_2],2,X_2) \leftarrow p([4,P_2],X_1,X_2)$.
$p([6,P_2],X_1,X_2) \leftarrow p([3,P_2],X_1,X_2), X_2 \neq 1$.
$p([6,P_2],X_1,X_2) \leftarrow p([5,P_2],X_1,X_2)$.
$p([7,P_2],4,X_2) \leftarrow p([6,P_2],X_1,X_2)$.
$p([8,P_2],X_1,X_2) \leftarrow p([7,P_2],X_1,X_2)$.
$p([9,P_2],X_1,X_2) \leftarrow p([8,P_2],X_1,X_2)$,
$(X_2 < 2 \vee X_2 > 3)$.
$p([0,P_2],0,X_2) \leftarrow p([9,P_2],X_1,X_2)$.

% Rules for Process 2
$p([P_1,1],X_1,1) \leftarrow p([P_1,0],X_1,X_2)$.
$p([P_1,2],X_1,X_2) \leftarrow p([P_1,1],X_1,X_2), X_1 < 3$.
$p([P_1,3],X_1,3) \leftarrow p([P_1,2],X_1,X_2)$.
$p([P_1,4],X_1,X_2) \leftarrow p([P_1,3],X_1,X_2), X_1 = 1$.
$p([P_1,5],X_1,2) \leftarrow p([P_1,4],X_1,X_2)$.
$p([P_1,6],X_1,X_2) \leftarrow p([P_1,3],X_1,X_2), X_1 \neq 1$.
$p([P_1,6],X_1,X_2) \leftarrow p([P_1,5],X_1,X_2)$.
$p([P_1,7],X_1,4) \leftarrow p([P_1,6],X_1,X_2)$.
$p([P_1,8],X_1,X_2) \leftarrow p([P_1,7],X_1,X_2), X_1 < 2$.
$p([P_1,9],X_1,X_2) \leftarrow p([P_1,8],X_1,X_2)$.
$p([P_1,0],X_1,0) \leftarrow p([P_1,9],X_1,X_2)$.

Fig. 6. CLP Representation of Szymanski's Algorithm

Example 5 (Szymanski's Algorithm). Szymanski's algorithm is a more complex priority-based mutual exclusion algorithm which is commonly encountered in the literature. We show the pseudocode in Figure 5. Its CLP representation is in Figure 6.

Roughly speaking, since the algorithm is based on prioritizing Process 1 to enter the critical section ⟨8⟩, it is not possible for Process 2 to be in the critical section while Process 1 is at its trying section. For example, the following does not hold:

$$p([8,7],X_1,X_2) \models p([7,8],X_2,X_1).$$

It is because the program points $[8,7]$ are reachable while $[7,8]$ are not. In other words, there is a grounding for the lhs goal, but no grounding for the rhs goal. Therefore, a simple symmetry assertion such the one given in the bakery algorithm does not hold. However, the following "not-quite" symmetry assertions still hold:

$$\begin{aligned} p([8,P_2],X_1,X_2), P_2 < 3 &\models p([P_2,8],X_2,X_1). \\ p([8,P_2],X_1,X_2), P_2 > 7 &\models p([P_2,8],X_2,X_1). \\ p([9,P_2],X_1,X_2), P_2 \neq 7 &\models p([P_2,9],X_2,X_1). \\ p([P_1,P_2],X_1,X_2), P_1 \neq 8, P_1 \neq 9 &\models p([P_2,P_1],X_2,X_1). \end{aligned}$$

At first it seems that the above assertions no longer defines an automorphism group since $p([P_1,8],X_1,X_2), 3 \leq P_1 \leq 7 \models p([8,P_1],X_2,X_1)$ can be derived from the last as-

```
Consumer:                                    Producer:
    while (true) do                              while (true) do
⟨0⟩     await (full=1) full:=0               ⟨0⟩     pro_1() ⟨1⟩ ... ⟨n−1⟩ pro_n()
⟨1⟩     con_1() ⟨2⟩ ... ⟨n⟩ con_n() ⟨n+1⟩    ⟨n⟩     await (full=0) full:=1 ⟨n+1⟩
    end                                          end
```

Fig. 7. Producer/Consumer

```
p([0,0],0,X). % Initial State
% Consumer                                   % Producer
p([1,P2],0,X) ← p([0,P2],1,X).               p([P1,1],Full,X) ← p([P1,0],Full,X).
p([2,P2],Full,X) ← p([1,P2],Full,X).         p([P1,n],Full,X) ← p([P1,n−1],Full,X).
p([n,P2],Full,X) ← p([n−1,P2],Full,X).       p([P1,n+1],Full,X) ← p([P1,n],Full,X).
p([0,P2],Full,X) ← p([n,P2],Full,X).         p([P1,0],1,X) ← p([P1,n+1],0,X).
```

Fig. 8. Partial CLP Representation of Producer/Consumer

sertion, yet the inverse does not hold. However, by observation the assertion $p([P_1,8], X_1,X_2) \models P_1 < 3 \vee P_1 > 7$ holds since it is not possible for process 2 to be in the critical section while process 1 is waiting. Similarly, $p([P_1,9],X_1,X_2) \models P_1 \neq 7$ also holds. These impose restrictions on the last assertion above.

We are not aware of any verification technique that would allow us to express and use this kind of symmetry.

Example 6 (Serializability). We next discuss an application of relative safety assertion beyond symmetry. We show a producer/consumer program in Figure 7, which CLP representation is in Figure 8. The macros $\texttt{con}_k\texttt{()}$ and $\texttt{pro}_l\texttt{()}$, abstract program fragments that serve to produce and consume respectively. We will imagine that apart from the variable `full` there are other variables `x` which may be used in $\texttt{con}_k\texttt{()}$ and $\texttt{pro}_l\texttt{()}$.

Consider the assertions:

$p([n+1,P_2],Full,f(X)),P_2 \leq n \models p([1,P_2],Full,X)$.
$p([P_1,n],Full,g(X)),P_1 \geq 1 \models p([P_1,0],Full,X)$.

where the expression $f(X)$ and $g(X)$ are the results of performing $\texttt{con}_1\texttt{()} \ \ldots \ \texttt{con}_n\texttt{()}$ and $\texttt{pro}_1\texttt{()} \ \ldots \ \texttt{pro}_n\texttt{()}$ respectively on X. Then the assertions say that the *result* of performing the interleaving of $\texttt{con}_k\texttt{()}$ and $\texttt{pro}_l\texttt{()}$ macros, $1 \leq k \leq P_1 - 1$, $1 \leq l \leq P_2$ is as though the two sequences of transitions are serializable. Note that here we still have an automorphism group which contains the above assertions and their inverses.

Both symmetry and serializability are examples of *non-behavioral* properties, i.e., properties determined by the structure of the program. They are not necessarily related to the intended result of the computation. Relative safety is potentially useful to specify many other useful non-behavioral properties, possibly ad-hoc and application specific. The class of such properties is potentially large. It is intuitively clear that such information can help in speeding up the proof process of other properties, which we will demonstrate later.

5 The Proof Method

Now let $G = (B_1,\ldots,B_n,\phi)$ and P denote a goal and program respectively. Let $R = A \leftarrow C_1,\ldots,C_m,\phi_1$ denote a rule in P, written so that none of its variables appear in

G. Let the equation $A = B$ be shorthand for the pairwise equation of the corresponding arguments of A and B. A *reduct* of G using R, denoted by reduct(G,R), is of the form

$$(B_1,\ldots,B_{i-1},C_1,\ldots,C_m,B_{i+1},\ldots,B_n,B_i = A \wedge \phi \wedge \phi_1)$$

provided the constraint $B_i = A \wedge \phi \wedge \phi_1$ has a true ground instance. Since the CLP rules are implications, it follows that $G \leftarrow \text{reduct}(G,R)$ holds.

Definition 3 (Unfold). *Given a program P and a goal G which contain one atom, a complete unfold of a goal G, denoted by* unfold(G) *is the set* $\{G' | \exists R \in P : G' = \text{reduct}(G,R)\}$. *A (not necessarily complete) unfold of G is a set* unfold$'(G) \subseteq$ unfold(G).

Note that since $[\![G]\!] \neq \emptyset$ only if $G \cap T_P([\![\bigvee \text{unfold}(G)]\!]) \neq \emptyset$, and this holds only if $[\![\bigvee \text{unfold}(G)]\!] \neq \emptyset$, we have the *logical semantics of unfold*: $G \rightarrow \bigvee \text{unfold}(G)$.

Definition 4 (Unfold Tree Goals). *Given a program P and a set H of goals each contain one atom, we define the function $\delta(H) = H \cup \text{unfold}'(G_1)$, when $G_1 \in H$. We obtain a set of* unfold tree goals *of G by a finite successive applications of δ on $\{G\}$.*

Since for any goal G, $G \leftarrow \text{reduct}(G,R)$, for any goal G_1 in the unfold tree goals of G, $G_1 \rightarrow G$.

Definition 5 (Frontier). *Given a program P and a set H of goals which contains one atom, when there exists $G_1 \in H$, we define the nondeterministic function $\varepsilon(H) = (H - \{G_1\}) \cup \text{unfold}(G_1)$. $\varepsilon()$ can be successively applied to a singleton set containing an initial goal G obtaining a* frontier $F = \varepsilon(\ldots(\varepsilon(\{G\}))\ldots)$.

From the logical semantics of unfold, for any frontier F of G, $G \rightarrow \bigvee F$.

Intuitively, in order to prove $G^L \models G^R$, we proceed as follows: unfold G^L completely to obtain a frontier containing the goals $G^L_1,\ldots,G^L_n$, and unfold G^R (not necessarily completely) obtaining unfold tree goals $G^R_1,\ldots,G^R_m$. This is depicted in Figure 9. Then the proof holds if

$$G^L_1 \vee \ldots \vee G^L_n \models G^R_1 \vee \ldots \vee G^R_m$$

or alternatively, if $G^L_i \models G^R_1 \vee \ldots \vee G^R_m$ for all $1 \leq i \leq n$. The justification for this result comes from the logical semantics of unfold: we have that $G^L \rightarrow G^L_1 \vee \ldots \vee G^L_n$, and $G^R_j \rightarrow G^R$ for all j such that $1 \leq j \leq m$. By a chain of implications we may conclude $G^L \models G^R$.

More specifically, but with some loss of generality, the proof holds if

$$\forall i : 1 \leq i \leq n, \exists j : 1 \leq j \leq m : G^L_i \models G^R_j.$$

and for this reason, our *proof obligation* shall be defined below to be simply a pair of goals, written $G^L_i \models G^R_j$.

Note that since we replace the global satisfaction criterion by local criteria, our proof method is therefore incomplete in cases where we need to perform some unfolds of G^R, that is, when proving relative safety assertions. Unfold of G^R is not needed for proving traditional safety assertions.

Our proof method can also be viewed as checking that the set of states represented by the symbolic formula G^R is reachable, whenever the set G^L is reachable. This is done by showing that a frontier of states that reach G^L also reaches G^R. If G^L is to be reachable from the initial state, it must be through at least one of the states in this frontier. And since from all states in the frontier G^R is reachable, G^R must also be reachable from the initial state.

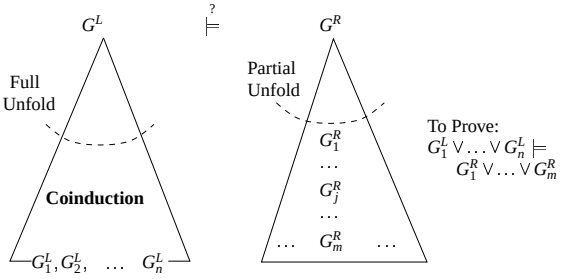

Fig. 9. Informal Structure of Proof Process

5.1 Proof Rules

We now present a calculus for proving relative safety assertions. To handle the possibly infinite unfoldings of G^L and G^R (see Figure 9), we shall depend on the use *coinduction* for the unfolding of G^L.

Proof by coinduction proceeds by assuming everything we like as long as we do not violate any facts. While assuming a set of assertions of the form $G^L \models G^R$ collected along an unfold path, we prove another assertion on the path, making it unnecessary to unfold the path further. For the use of coinduction, we now give the following definition.

Definition 6 (Proof Obligation). *A* proof obligation *is of the form* $\tilde{A} \vdash G^L \models G^R$*, where* G^L *and* G^R *are goals and* $\tilde{A}$ *is a set of assertions that are* assumed*.*

The role of proof obligations is to capture the state of a proof. The set $\tilde{A}$ contains assertions that can be used coinductively to discard the proof obligation at hand.

Our proof rules are presented in Figure 10. Each rule operates on the (possibly empty) set of proof obligations Π, by selecting a proof obligation from Π and attempting to discard it. In this process, new proof obligations may be produced. The proof process is typically centered around unfolding the goals in proof obligations.

The *left unfold and coinduction* (LU+C) rule performs a complete unfold on the lhs of a proof obligation, producing a new set of proof obligations. The original assertion, while removed from Π, is added as an assumption to every newly produced proof obligation, opening the door to using coinduction in the proof.

Example 6 (Proving Symmetry). We exemplify our proof rules using a proof of a symmetry property of the 2-process bakery algorithm (Figure 2):

$$p([P_1,P_2],T_1,T_2) \models p([P_2,P_1],T_2,T_1). \tag{1}$$

Initially, $\Pi = \{\emptyset \vdash p([P_1,P_2],T_1,T_2) \models p([P_2,P_1],T_2,T_1)\}$.

Using the rule LU+C, and all the CLP rules of Figure 2, we perform a left unfold of $G^L = p([P_1,P_2],T_1,T_2)$, obtaining a new set of proof obligations Π'. In particular, by the unfold of CLP rule `r1`, Π' includes the obligation (O1):

$\tilde{A}' \vdash p([P_1',P_2],T_1',T_2), P_1 = 1, P_1' = 0, T_1 = T_2 + 1 \models p([P_2,P_1],T_2,T_1)$,
where $\tilde{A}' = \{p([P_1,P_2],T_1,T_2) \models p([P_2,P_1],T_2,T_1)\}$.

By the unfold of CLP rule `init`, Π' also includes the obligation (O2):

$\tilde{A}' \vdash P_1 = P_2 = 0, T_1 = T_2 = 0 \models p([P_2, P_1], T_2, T_1)$.

Other than these two obligations, Π' also includes the result of unfolding using the rules `e1`, `x1`, `r2`, `e2`, and `x2`.

The rule *right unfold* (RU) performs an unfold operation on the rhs of a proof obligation. Note that only one unfolded goal is used. Now, in practice, it is generally not known which reduct G_i^R of G^R is the one we need later, or indeed if G^R itself is needed later.

Returning to our example, by unfolding $G^R = p([P_2, P_1], T_2, T_1)$ of (O1) using proof rule RU and CLP rule `r2` of Figure 2, we obtain Π'' which includes (O3):

$$\tilde{A}' \vdash p([P_1', P_2], T_1', T_2), P_1 = 1, P_1' = 0, T_1 = T_2 + 1 \models \\ p([P_2, P_1''], T_2, T_1''), P_1 = 1, P_1'' = 0, T_1 = T_2 + 1$$

Similarly, by unfolding the rhs of (O2) using RU and the CLP rule `init`, we obtain Π''' which includes the obligation (O4):

$\tilde{A}' \vdash P_1 = P_2 = 0, T_1 = T_2 = 0 \models P_1 = P_2 = 0, T_1 = T_2 = 0$.

The rule *assumption proof* (AP) transforms an obligation by using an assumption, and realizes the coinduction principle (since assumptions can only be created by the rule (LU+C)).

Continuing our example, we can now immediately prove (O3) by rule AP, and applying the original symmetry assertion (1) which is included in the set of assumed assertions $\tilde{A}'$ of (O3). More concretely, we apply (1) to the lhs of (O3) obtaining the goal $p([P_2, P_1'], T_2, T_1'), P_1 = 1, P_1' = 0, T_1 = T_2 + 1$, which clearly implies the rhs of (O3) by renaming of each double primed variables to its single primed version.

The rule *direct proof* (DP) discards a proof obligation when it can be directly proven that it holds, possibly by some renaming of variables. This rule is used to discharge (O4), since it is immediately clear that it holds. The renaming θ that we apply here is the identity.

Finally, the rule *split* (SPL) converts a proof obligation into several, more specialized ones.

Given an assertion $G^L \models G^R$, a proof shall start with $\Pi = \{\tilde{A} \vdash G^L \models G^R\}$, and proceed by repeatedly applying the rules in Figure 10 to it. The conditions in which a proof can be completed are stated in the following theorem.

Theorem 1 (Proof of Assertions). *A safety assertion $G^L \models G^R$ holds if, starting with the proof obligation $\Pi = \{\emptyset \vdash G^L \models G^R\}$, there exists a sequence of applications of proof rules that results in $\Pi = \emptyset$. The safety assertion holds conditionally on $\tilde{A}$ if we start with $\Pi = \{\tilde{A} \vdash G^L \models G^R\}$, where $\tilde{A} \neq \emptyset$.*

Our proof method can be used to prove traditional safety assertion $G^L \models \Psi$, to prove relative safety assertion $G^L \models G^R$, where G^R contains an atom, and to prove traditional safety assertion using other assertions, e.g., relative safety assertions representing symmetry, possibly obtaining smaller proof. For the last use we start a prove of traditional safety assertion with a non-empty set of assumed assertions.

The proof rules above are sufficient in principle for our purposes. However, there is a very important principle which gives rise to an optimization: *redundancy* between obligations which essential idea is based on the observation that in proving $G^L \models G^R$, we may obtain a goal G_i^L by a sequence of unfolds from G^L, and prove the obligation

$$(\text{LU+C})\ \frac{\Pi \cup \{\tilde{A} \vdash G^L \models G^R\}}{\Pi \cup \bigcup_{i=1}^{n} \{\tilde{A} \cup \{G^L \models G^R\} \vdash G_i^L \models G^R\}} \quad \text{unfold}(G^L) = \{G_1^L, \ldots, G_n^L\}$$

$$(\text{RU})\ \frac{\Pi \cup \{\tilde{A} \vdash G^L \models G^R\}}{\Pi \cup \{\tilde{A} \vdash G^L \models G_i^R\}} \quad G_i^R \in \text{unfold}(G^R)$$

$$(\text{AP})\ \frac{\Pi \cup \{\tilde{A} \vdash G^L, \phi \models G^R\}}{\Pi \cup \{\tilde{A} \vdash G_1^R\theta, \phi \models G^R\}} \quad \begin{array}{l} G_1^L \models G_1^R \in \tilde{A} \text{ and there} \\ \text{exists a renaming } \theta \text{ s.t.} \\ G^L \models G_1^L\theta \end{array}$$

$$(\text{DP})\ \frac{\Pi \cup \{\tilde{A} \vdash G^L, \phi \models G^R\}}{\Pi} \quad \begin{array}{l} \text{There exists a renaming} \\ \theta \text{ s.t. } G^L \models G^R\theta \end{array}$$

$$(\text{SPL})\ \frac{\Pi \cup \{\tilde{A} \vdash G^L \models G^R\}}{\Pi \cup \bigcup_{i=1}^{k} \{\tilde{A} \vdash G^L, \phi_i \models G^R\}} \quad \phi_1 \vee \ldots \vee \phi_k \text{ is true.}$$

Fig. 10. Proof Rules

$G_i^L \models G^R$. Using this we can try to establish $G_j^L, \phi \models G^R$ in another part of the tree, where $i \neq j$, where there exists a renaming θ such that $G_j^L \models G_i^L\theta$. Here, we *reuse* the proof of $G_i^L \models G^R$ in the proof of $G_j^L, \phi \models G^R$.

A fundamental question in proving relative safety assertion $G^L \models G^R$ in general, is how to interleave the unfolding of the lhs versus the rhs. For this we can repeatedly apply left-unfolding on G^L either until "looping", that is, until each path in the tree contains a repeated occurrence of a program counter, or the final goal of the path is a constraint. This is because coinduction is likely to be applicable at a looping point.

6 Implementation and Experiments

We implemented our proof algorithm as regular CLP($\mathcal{R}$) [15] programs. Our prototype implementations use coinduction, and a tabling mechanism for storing assumed assertions. We run our prototypes using a 2 GHz Pentium 4 Xeon machine with 2 GB of RAM.

Our first prototype is for proving relative assertions. Here we hope that the symmetry proof using coinduction concludes in just 1 level of unfold of both lhs and rhs of the assertion, because this is the case for *perfectly symmetric* programs. These include bakery algorithm and dining philosophers' problem. In these examples, every transition from state s to t has its symmetric counterpart that maps $\pi(s)$ to the $\pi(t)$, where π an automorphism of states. Our implementation therefore first tries to check goals obtained from 1 level of both lhs and rhs unfold. For each goal in the lhs frontier, it tries to search for a goal in the rhs of depth 1, such that the original symmetry assertion is applicable coinductively. Where the proof does not conclude in this manner, we have a program with imperfect symmetry, such is the case with the simple priority mutual exclusion and Szymanski's algorithm. In this case, general depth-first traversal of lhs subtree is initiated. For producer-consumer problem, we do not perform any lhs unfolding.

Problem	A#	LSt	RSt	T
Bakery-2	1	9	27	0.00
Bakery-3	2	44	254	0.10
Bakery-4	3	147	1557	11.28
Bakery-5	4	424	7804	2320.3
Bakery-6	5	∞	∞	∞
Philosopher-3	1	19	124	0.01

Problem	A#	LSt	RSt	T
Philosopher-4	1	24	232	0.02
Priority	1	43	220	0.04
Szymanski-2	8	362	28419	59.11
Szymanski-3	16	∞	∞	∞
Prod/Cons-10	2	0	170	0.19
Prod/Cons-20	2	0	530	1.88

Table 1. Relative Safety Proof Experimental Results

Experimental results in proving relative safety assertions are shown in Table 1, where A#=number of verified assertions, LSt=number of visited lhs goals, RSt=number of visited rhs goals, and T=time in seconds. In *ProblemName-N*, *N* denotes the number of processes, except for *Prod/Cons-N* where *N* denotes that there are *N* produce and consume operations. Note that we could not complete the experiment for 6-process bakery algorithm and 3-process Szymanski's algorithm after a few hours.

We also implemented a second prototype to prove safety assertions of the form $G \models \mathit{false}$ with or without assumed relative safety assertions (e.g., symmetry). $G \models \mathit{false}$ declares non-reachability of error states G.

A coinductive verification requires matching between the goal in an assertion and an assumed assertion such that the said assertion can be proven coinductively. As is common in the literature, for verification using symmetry we need to define a set of *canonical representatives* of the equivalence class of goals induced by given symmetry, such that the matching can be done efficiently among representatives. Unfortunately, finding all the canonical representatives of a goal is a hard problem known as the *orbit problem* [2]. Our solution here is to try to generate canonical representatives of a goal only up to a constant number, and we employ a sorting algorithm as our canonicalization function. We note, however, that canonicalization is not hard for dining philosophers' problem since for this problem it is a cyclic shift which is linear to the permutable domain size (cf. [2]). Also that neither sorting nor cyclic shift is necessary when using serializability assertions.

The results are shown in Table 2 (a). The proof of traditional safety does not require right unfolding, hence there is no column for RSt value. We ran the bakery, Peterson's, Lamport's fast mutual exclusion and Szymanski's algorithms proving mutual exclusion. Note that we do not prove the symmetry assertions of some of the problems (e.g., Szymanski-3). For the dining philosophers' problem, we prove that there cannot be more than $N/2$ philosophers simultaneously eating. For the producer-consumer problem, each $\mathtt{pro}_i()$ increments a variable x, and $\mathtt{con}_j()$ decrements it. Here we verify that the value of x can never be more than $2n$.

Bakery algorithm has infinite reachable states, and therefore cannot be handled by finite-state model checkers. We compare our search space the results of the CLP-based system of Delzanno and Podelski [4]. As also noted by Delzanno and Podelski, the problem does not scale well to larger number of processes, but using symmetry, we have pushed its verification limit to 7 processes without abstraction.

In Table 2 (b) we summarize the effectiveness of the use of a variety of relative safety assertions. The use of symmetry assertion effectively reduces the search space of perfectly symmetric problems (bakery, Peterson's, Lamport's fast mutex, dining philosophers). However, the reduction for Szymanski's algorithm is competitive

Problem	CLP/Coinductive Tabling				Delzanno-Podelski # Facts
	No Assertion		W/ Assertion		
	LSt	T	LSt	T	
Bakery-2	15	0.00	8	0.00	13
Bakery-3	296	0.07	45	0.01	109
Bakery-4	4624	6.60	191	0.20	963
Bakery-5	∞	∞	677	2.88	
Bakery-6	∞	∞	2569	49.08	
Bakery-7	∞	∞	11865	1052.32	
Peterson-2	105	0.05	10	0.00	
Peterson-3	20285	119.03	175	0.15	
Peterson-4	∞	∞	3510	11.98	
Lamport-2	143	0.02	72	0.02	
Lamport-3	4255	1.13	707	0.40	
Lamport-4	∞	∞	5626	7.63	
Szymanski-2	240	0.08	84	0.02	
Szymanski-3	10883	35.43	3176	2.91	
Philosopher-3	882	0.51	553	0.30	
Philosopher-4	4293	27.77	2783	9.67	
Prod/Cons-10	664	0.10	171	0.02	
Prod/Cons-20	2314	1.90	331	0.04	

(a) Stored Assertions and Time

Problem Type	% Reduction	
	LSt	T
Bakery	76%	78%
Peterson	95%	99.9%
Lamport	67%	65%
Szymanski	68%	83%
Philosopher	36%	53%
Prod/Cons	87%	94%

(b) % Reduction

Table 2. Safety Proof Experimental Results

with perfectly symmetric problems, showing that "not-quite" symmetry reduction is worth pursuing. The use of rotational symmetry in the dining philosophers' problem is, expectedly, less effective. We also note that we managed to obtain a substantial reduction of state space for the producer/consumer problem. Reduction in time roughly corresponds to those of state space.

Finally, comparing Table 1 and 2, the proof of relative safety assertions are no easier than the proof of traditional safety assertions, even with coinduction. This is because of the need to perform rhs unfold when proving relative safety.

7 Conclusion

In this paper, we introduced a novel assertion called relative safety. This can be uniquely used to assert structural properties of programs. We chose a driving application area of symmetry, and demonstrated that, by using relative safety, we could accommodate a larger class of programs than have been previously considered by other means. We provided a proof system, based upon well understood computational steps of unfolding, and introduced a new coinductive tabling mechanism. We then ran some experiments in order to show the practical potential of our algorithm. Further work is to discover more important classes of structural properties for which relative safety can be used.

References

1. M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM TOPLAS*, 16(5):1543–1571, September 1994.

2. E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In A. J. Hu and M. Y. Vardi, editors, *10th CAV*, volume 1427 of *LNCS*, pages 147–158. Springer, 1998.
3. E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *5th CAV*, volume 697 of *LNCS*, pages 450–462. Springer, 1993.
4. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *Int. J. STTT*, 3(3):250–270, 2001.
5. X. Du, C. R. Ramakrishnan, and S. A. Smolka. Tabled resolution + constraints: A recipe for model checking real-time systems. In *21st RTSS*, pages 175–184. IEEE Computer Society Press, 2000.
6. E. A. Emerson. From asymmetry to full symmetry: New techniques for symmetry reductions in model checking. In L. Pierre and T. Kropf, editors, *10th CHARME*, volume 1703 of *LNCS*, pages 142–156. Springer, 1999.
7. E. A. Emerson, J. Havlicek, and R. J. Trefler. Virtual symmetry reduction. In *15th LICS*, pages 121–131. IEEE Computer Society Press, 2000.
8. E. A. Emerson and A. P. Sistla. Model checking and symmetry. In *5th CAV*, volume 697 of *LNCS*, pages 463–478. Springer, 1993.
9. E. A. Emerson and A. P. Sistla. Utilizing symmetry when model-checking under fairness assumptions. *ACM TOPLAS*, 19(4):617–638, July 1997.
10. 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.
11. L. Fribourg. Constraint logic programming applied to model checking. In *9th LOPSTR*, volume 1817 of *LNCS*, pages 30–41. Springer, 1999.
12. 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.
13. C. N. Ip and D. L. Dill. Better verification through symmetry. *FMSD*, 9(1/2):41–75, 1996.
14. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. LP*, 19/20:503–581, May/July 1994.
15. 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.
16. J. Jaffar, A. Santosa, and R. Voicu. A CLP proof method for timed automata. In *25th RTSS*, pages 175–186. IEEE Computer Society Press, 2004.
17. M. Leuschel and T. Massart. Infinite-state model checking by abstract interpretation and program specialization. In *9th LOPSTR*, volume 1817 of *LNCS*, pages 62–81. Springer, 1999.
18. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model checking. In J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. La u, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *1st CL*, volume 1861 of *LNCS*, pages 384–398. Springer, 2000.
19. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In O. Grumberg, editor, *9th CAV*, volume 1254 of *LNCS*, pages 143–154. Springer, 1997.
20. A. P. Sistla and P. Godefroid. Symmetry and reduced symmetry in model checking. *ACM TOPLAS*, 26(4):702–734, July 2004.
21. A. P. Sistla, V. Gyuris, and E. A. Emerson. SMC: A symmetry-based model checker for verification of safety and liveness properties. *ACM TOSEM*, 9(2):133–166, April 2000.
22. F. Wang. Efficient data structure for fully symbolic verification of real-time systems. In S. Graf and M. I. Schwartzbach, editors, *6th TACAS*, volume 1785 of *LNCS*, pages 157–171. Springer, 2000.
23. H. Weyl. *Symmetry*. Princeton University Press, 1952.