Basic Research in Computer Science

Dynamic Partitioning in Linear Relation Analysis

Application to the Verification of Synchronous Programs

Bertrand Jeannet

BRICS Report Series **RS-00-38**

ISSN 0909-0878 **December 2000**

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/00/38/`

Dynamic Partitioning In Linear Relation Analysis Application To The Verification Of Reactive Systems*

Bertrand Jeannet†

December, 2000

Abstract

We apply linear relation analysis [CH78, HPR97] to the verification of declarative synchronous programs [Hal98]. In this approach, *state partitioning* plays an important role: on one hand the precision of the results highly depends on the fineness of the partitioning; on the other hand, a too much detailed partitioning may result in an exponential explosion of the analysis. In this paper we propose to consider very general partitions of the state space and to dynamically select a suitable partitioning according to the property to be proved. The presented approach is quite general and can be applied to other abstract interpretations.

Keywords and Phrases: Abstract Interpretation, Partitioning, Linear Relation Analysis, Reactive Systems, Program Verification

1 Introduction

Reactive systems are a privileged field for applying formal verification methods, as they are generally used in critical systems where errors can have dramatic consequences. Since these systems interact with an environment, they

*This work was mostly carried out in VERIMAG and finalized in BRICS; it has been partially supported by the ESPRIT-LTR project "SYRF".

†**BRICS**: **B**asic **R**esearch **i**n **C**omputer **S**cience, Centre of the Danish National Research Foundation, Department of Computer Science, Aalborg University, Fr. Bajersvej 7E, 9220 Aalborg Ø, Denmark. Email: `bjeannet@cs.auc.dk`. Fax: +45 9815 9889. Supported by an **INRIA** grant: **I**nstitut **N**ational de **R**echerche en **I**nformatique et **A**utomatique, Domaine de Voluceau – BP 105, 78153 Rocquencourt, France

are usually modeled as dynamic systems the evolution of which depends on the environment. They can be classified according to the structure of their state space and the expressiveness of their evolution laws, and as a consequence to the decidability of the verification problem. In this paper, we are interested in the verification of safety properties of systems that exhibit both Boolean and numerical behavior, such as linear hybrid systems [ACH+95], or synchronous programs containing numerical state variables [HPR97], for which the verification problem is undecidable.

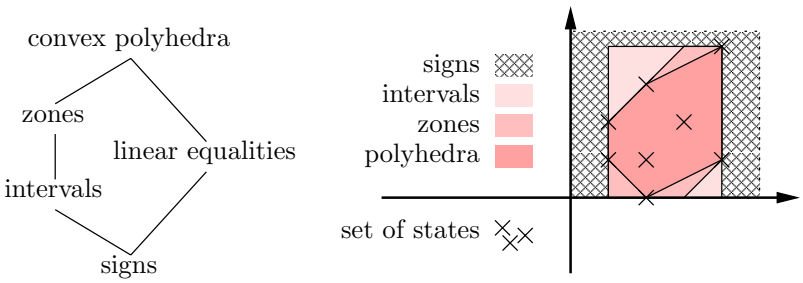
A common solution to overcome the undecidability of the verification is to make use of *approximation* to conservatively check properties and to use *abstract interpretation* as a theoretical framework. The use of approximation can also be justified, in a more positive way, as a technique to reduce the complexity of the algorithms and to afford the practical verification of bigger systems.

Now, a major difficulty is to adjust the level of approximation used. A tradeoff has to be found between precision and efficiency. Rough approximations make analysis cheaper but may fail in showing non trivial properties; more precise analyses may be too expensive to be of practical interest.

In this paper, we propose a solution to find automatically such a tradeoff between precision and efficiency, based on abstract interpretation techniques, and we apply it to the particular case of synchronous programs. This solution could be applied as well to linear hybrid systems with minor modifications. Let us make more precise the problem we attack and the principle of our solution.

Partitioning in abstract interpretation. Partitioning is a classical technique, in abstract interpretation [CC77], which consists of splitting the state space of the system $S = K \times S'$ into a control part (a set K of control point) and a data value part S', and in associating with each control point a set of reachable data values. No approximation is made on the control part — control points are enumerated —, whereas sets of data values are represented by elements of an *abstract lattice* L having suitable relationships with S'. The analysis consists then in solving a system of fixpoint equations $\bigwedge_{k \in K} X^{(k)} = F^{(k)}(X^{(1)}, \ldots, X^{(|K|)})$ where $X^{(k)} \in L$ is associated to the control point k. As a consequence, the "global" abstract lattice used by the analysis is the lattice L^K.

Precision and complexity issues. In this context, a classical approach to adjust the precision — and the complexity — of the analysis, is to



(a) Hierarchy of abstract domains for numerical sets

(b) Example of abstraction

Figure 1: Abstract domains for numerical sets

choose the abstract lattice L depending on the type of information one wants to obtain. For numerical sets, one can choose, for instance, signs, intervals [CC76], zones [HNSY92], linear equalities [Kar76], and convex polyhedra lattices [CH78], the precision of which can be ordered as shown on Fig. 1(a). Fig. 1(b) gives the abstract value representing a set of states, depending on the abstract domain.

However, this approach is somewhat coarsed-grained, and sometimes inefficient: for instance, all the abovementioned numerical lattices abstract the set $[-\infty, -5] \cup [5, +\infty]$ into the top element: these lattices can only represent convex sets, and their *lub* (least upper bound) operator can lose too much information.

In addition, another problem arises, which is not addressed at all by the choice of the numerical lattice: the set K of control point may be large enough to become the main source of complexity. This *state explosion* problem, well-known in the verification of finite state systems, happens as well in our case:

- Timed and hybrid systems are generally described by compositions of many timed or hybrid automata. Performing the product of these components to obtain a single automaton, often results in an explosion of the size of the automaton.

- In data-flow synchronous languages [HCRP91, LGLL91], and in circuits, the automaton is implicitly given by means of Boolean (or finite domain) *state variables*. By performing a partial evaluation [JGS93] of all Boolean variables of the program, an explicit interpreted automaton may be built [Hal98] and used for the analysis.

In both cases, the control structure is often too large to be managed: obvi-

```
initial not b0 and not b1 and (x=0) and (y=0);
transition
    b0' = not b1;
    b1' = b0;
    x' = if b0=b1 then x+1 else x;
    y' = if b0=b1 then y else y+1;
```

Figure 2: An example "program"

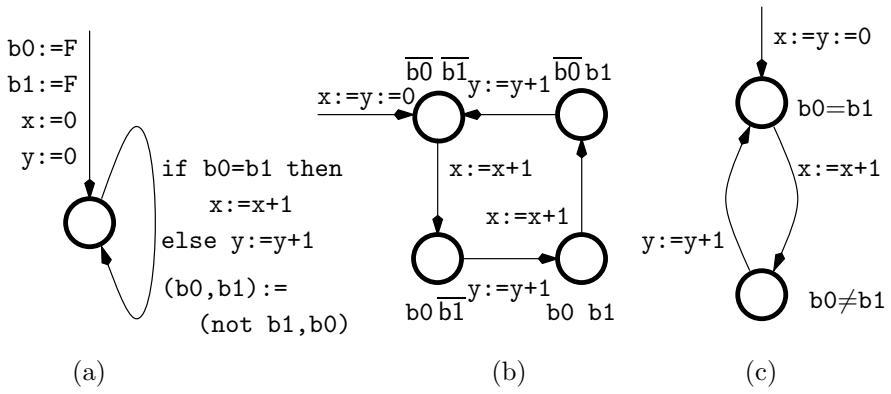


Figure 3: Control structures for the example program

ously, in case where abstract values are convex polyhedra, solving a system made of millions of equations is hopeless.

Let us consider a trivial example: the synchronous program shown on Fig. 2 is, basically, a system of equations defining the dynamic behavior of a set of Boolean ("b1, b2") and integer ("x, y") variables. Their possible initial states are specified by a formula ("initial"), then each equation defines the "next" (primed) value of a variable as an expression of the current values of variables. Fig. 3 (a) and (b) show two imperative versions of this program: Fig. 3(a) is the trivial control structure (one control state), while Fig. 3(b) shows the automaton obtained by a partial evaluation of the Boolean variables. If our goal is to prove, by linear relation analysis, that x is always greater than or equal to y, the control structure of Fig. 3(a) is too coarse (the behavior of Boolean variables is ignored); conversely, the partitioning of Fig. 3(b) is too much detailed: we could complexify the program, with n Boolean variables, and get a ring of 2^n locations, while the property can indeed be shown on an automaton with only 2 locations (Fig. 3(c)).

Principle of our method. We propose a fine-grained method to adjust the precision according to the property under verification, which was first presented in [JHR99]. This method is orthogonal to the choice of the lattice L and addresses the state explosion problem. It relies on the use of a new abstract lattice combining Boolean and numerical properties and a more general partitioning method:

- On one hand it allows an explicit union of elements of L to be associated to a control point, thus providing a better precision when the *lub* operator of the lattice L loses too much information; in the same time, as the cardinal of the explicit union is bounded, the gain in complexity is kept under control.

- On the other hand it makes possible to associate a unique element of L to several control points, as in Fig 3(c), thus reducing the complexity of the analysis, at the expense of a possible loss of precision.

The challenge is now to choose automatically a suitable partition for the verification of a safety property on a program, i.e., being detailed enough to conclude if the property is true, but being also the coarsest possible to limit the complexity of analysis. This partition cannot be found statically, as it depends of the dynamic behavior of the program. Our solution consists in starting from a very coarse partition (basically, distinguishing only between initial states, bad states and others), and in refining it dynamically according to heuristics, until the unreachability of bad states from initial ones has been shown.

The paper is organized as follows. Section 3 introduces the Abstract Interpretation theory as well as the notion of partitioning, and defines the model of programs we want to check. We define in section 4 a new abstract lattice that is suitable for dynamic partitioning. It allows the definition, in section 5, of more general control structures, as well as the kind of analysis we perform on them. Section 6 describes an original *approximate* technique to compute pre- and post-condition of abstract values, which appears experimentally to be essential in the overall success of dynamic partitioning. Section 7 presents the most challenging part of this work, namely the refinement heuristics, which are required to be automatic. Section 8 describes the tool NBac implementing the method and some experiments we conducted with it. Section 9 concludes the paper.

2 Related Work

Some works proposed solutions, either to solve the precision problem, either to reduce the size of the control structure.

The problem of the approximations induced by the *lub* operator on an abstract lattice L is well-known. A simple solution is to use the *disjonctive completion* of L [CC92a], which consists basically in considering non redundant unions of elements of L. The problem is then to limit the size of the union. [Bou92] proposes a refined version of this solution, in which the number of disjunctions is limited by the application of suitable widening operators. The term "dynamic partitioning" is used in this context. This solution is applied to the computation of good abstractions of minimal function graphs. Thus, this method is not guided by a property to prove, unlike ours.

In order to reduce the size of control structure, in the particular case of timed automata, minimization methods were proposed [ACD+92, YL93, STA98]. The refinement process is always associated with a reachability analysis. These works however differ greatly from ours, because they handle decidable systems, and they perform only exact computations. Moreover, they do not address the complexity of non-numerical control: locations are split w.r.t. numerical constraints only, starting from the explicit timed automaton to be verified, possibly furthermore decomposed w.r.t. numerical guards of transitions [YL93].

The work that is the closest of ours in spirit is certainly [DWT95]. Our refinement method presents also some similarities with refinement methods that have been devised for techniques combining theorem proving and model checking [CGJ+00]. These works will be cited with greater details in the following.

3 Preliminaries

3.1 Abstract interpretation

Abstract interpretation is a general method to find approximate solutions of fixpoint equations. Most program analysis problems come down to solving a fixpoint equation $x = F(x)$. Solving such an equation generally raises two kinds of problems:

(1) *The solution must be computed in a complex ordered domain* (typically, the powerset of the state space of a program). Elements of this domain must be efficiently represented and normalized; functions defined on the

domain, and the ordering relation among the domain, must be computed. A first approximation can take place at this level: instead of computing in the complex domain C of *concrete values*, one can choose a simpler *abstract domain* A, connected to C by means of two functions $\alpha : C \mapsto A$, $\gamma : A \mapsto C$ forming a Galois connection:

$$\forall x \in C, \forall y \in A, \quad \alpha(x) \leq_A y \iff x \leq_C \gamma(y)$$

where $\leq_C, \leq_A$ respectively denote the order relations on C and A. The approximation of a function F, from C to C, will be the function $\alpha(F) = \alpha \circ F \circ \gamma$, from A to A. The basic result is that, if C is a complete lattice, if F is increasing from C to C, then

$$\alpha(\mathit{lfp}(F)) \leq_A \mathit{lfp}(\alpha(F))$$

where $\mathit{lfp}(F)$ denotes the least fixpoint of F. So, computing the least fixpoint in the abstract domain provides an upper approximation of the fixpoint in the concrete one.

(2) *The iterative resolution of a fixpoint equation can involve infinite (or even transfinite) iterations.* In some cases, the abstraction performed in (1) is so strong that the abstract domain is either finite or of finite depth (there is no infinite, strictly increasing chain $y_0 <_A y_1 <_A \ldots$). In such a case, the resolution in the abstract domain converges in a finite number of steps. However, requiring the abstract domain to satisfy such a finiteness condition is quite restrictive. Better results [CC77, CC92b] can often be obtained by performing another kind of approximation: when the depth of the abstract domain is infinite, specific operators may be defined to extrapolate the limit of a sequence of abstract values. For an increasing sequence (computation of a least fixpoint) one uses a *widening operator*, usually noted ∇, from $A \times A$ to A, satisfying the following properties:

- $\forall y_1, y_2 \in A, \quad y_1 \leq_A y_1 \nabla y_2 \quad \text{and} \quad y_2 \leq_A y_1 \nabla y_2$
- For any increasing chain ($y_0 \leq_A y_1 \leq_A \ldots$), the increasing chain defined by $y'_0 = y_0$, $y'_{i+1} = y'_i \nabla y_{i+1}$, is not strictly increasing (i.e., stabilizes after a finite number of terms).

Now, to approximate the least fixpoint $\overline{y}$ of a function G:

$$\overline{y} = \lim_{i \geq 0} y_i, \text{ with } y_0 = \bot \text{ (the least element of } A) \text{ and } y_{i+1} = G(y_i)$$

we can compute an *ascending approximation sequence* $(y'_i)_{i \geq 0}$:

$$y'_0 = \bot \quad , \quad y'_{i+1} = y'_i \nabla G(y'_i)$$

which converges after a finite number of steps towards an upper approximation $\widetilde{y}$ of $\overline{y}$. This approximation can be made more precise by computing a *descending approximation sequence*

$$y''_0 = \widetilde{y} \quad , \quad y''_{i+1} = G(y''_i)$$

i.e., starting from $\widetilde{y}$ a standard sequence, without widening. Each term of the descending sequence is an upper approximation of the least fixpoint $\overline{y}$.

3.2 Partitioning in abstract interpretation

Assume the concrete domain C is the powerset of some set S of states, let K be a finite set of *locations* and $\mathcal{K} = \{S^{(1)}, \ldots, S^{(|K|)}\}$ be a finite partition (or a finite covering) of S (i.e., such that $S = \bigcup_{k\in K} S^{(k)}$). For each $k \in K$, let $C^{(k)} = 2^{S^{(k)}}$; for each $x \in C$, $x^{(k)} = x \cap S^{(k)}$ belongs to $C^{(k)}$. Clearly, for each $x \in C$, the set $\{x^{(k)} \mid k \in K\}$ is a finite covering of x. Now, any fixpoint equation $x = F(x)$ on the domain C can be written as a system of equations

$$\bigwedge_{k\in K} x^{(k)} = F^{(k)}(x^{(1)}, x^{(2)}, \ldots, x^{(|K|)})$$

on the domain $C^{\mathcal{K}} = C^{(1)} \times \ldots \times C^{(|K|)}$, where

$$F^{(k)}(x^{(1)}, x^{(2)}, \ldots, x^{(|K|)}) = F(x^{(1)} \cup x^{(2)} \cup \ldots \cup x^{(|K|)}) \ \cap \ S^{(k)}$$

The partition can obviously be reflected in the abstract domain, by setting $y^{(k)} = \alpha(x^{(k)})$, resulting in an abstract system of equations

$$\bigwedge_{k\in K} y^{(k)} = G^{(k)}(y^{(1)}, y^{(2)}, \ldots, y^{(|K|)})$$

on the domain $A^{\mathcal{K}} = \bigotimes_{k\in K} A^{(k)}$, where $A^{(k)} = \{y \in A \mid y \sqsubseteq \alpha(S^{(k)})\}$ and $G^{(k)}$ is an upper-approximation of $F^{(k)}$ on the functional domain $A^{\mathcal{K}} \to A^{(k)}$. C and $A^{\mathcal{K}}$ are then connected by the Galois connection $(\alpha^{\mathcal{K}}, \gamma^{\mathcal{K}})$ with

$$\begin{array}{rcl} \alpha^{\mathcal{K}}(x) & = & \Big(\alpha(x \cap S^{(1)}), \ldots, \alpha(x \cap S^{(|K|)})\Big) \\ \gamma^{\mathcal{K}}(y^{(1)}, \ldots, y^{(|K|)}) & = & \gamma(y^{(1)}) \cup \ldots \cup \gamma(y^{(|K|)}) \end{array}$$

We will often denote $(y^{(1)}, \ldots, y^{(|K|)}) \in A^{\mathcal{K}}$ by $y^{(1)} \cup \ldots \cup y^{(|K|)}$. This partitioning first allows more efficient iterative resolution by using chaotic iterations [Cou77]. But it can also make the results more precise for two main reasons:

- When the least upper bound operator on A loses information, which is the case for most classical numerical lattice (intervals, zones, linear equalities, convex polyhedra), partitioning allows a concrete element $x \in C$ to be abstracted by an explicit union. For example, if $S = \mathbb{Z}^2$ and $C = 2^S$ is abstracted by the lattice of intervals $A = I(\mathbb{Z})^2$, taking $S^{(1)} = \{(x, y) \mid x \geq 0\}$ and $S^{(2)} = \{(x, y) \mid x < 0\}$ as a partition allows $X = \{(x, y) \mid x \geq 0 \Leftrightarrow y \geq 0\}$ to be abstracted by $[0, +\infty] \times [0, +\infty] \cup [-\infty, -1] \times [-\infty, -1]$ instead of $\mathbb{Z}^2$. Notice that a partitioning of S according to the positivity of y would give the same result: different partitioning can lead to the same results.

- Partitioning allows also a less frequent application of the widening operator, which can lose most information. Let us define the dependence relation $\mathcal{R}_G$ on elements of K as follows: k depends on k' if the value of $G^{(k)}(y^{(1)}, y^{(2)}, \ldots, y^{(|K|)})$ can depend on the value of $y^{(k')}$. Let K_∇ be a subset of K such that the graph of $\mathcal{R}_G$ restricted to $K \setminus K_\nabla$ has no loop. Then the convergence of the ascending approximation sequence is guaranteed even if the widening operator is only applied to components belonging to K_∇:

$$\begin{array}{ll} \forall k \in K, & y_0'^{(k)} = \bot \\ \forall k \in K_\nabla, & y_{i+1}'^{(k)} = y_i'^{(k)} \nabla G^{(k)}(y_i^{(1)}, y_i^{(2)}, \ldots, y_i^{(|K|)}) \\ \forall k \in K \setminus K_\nabla, & y_{i+1}'^{(k)} = G^{(k)}(y_i^{(1)}, y_i^{(2)}, \ldots, y_i^{(|K|)}) \end{array}$$

In the case of imperative programs, a standard partitioning is naturally given by a set K of control points ("program counter" values). The state space of the program has the form $S = K \times S'$ and an abstract value is a pair $\langle k, v \rangle$ where $k \in K$ and v representing a set of data values in S'. However, we will start from declarative programs, where no obvious partitioning is available.

3.3 Symbolic Representation of Programs and Properties

We address the particular case of declarative synchronous programs, whose syntax and semantics are given below.

3.3.1 Syntax

We express a program together with the property to be proved by:

```
state b0,b1,ok : bool; x,y : int;
initial not b0 and not b1 and (x=0) and (y=0) and ok;
transition
    b0' = not b0;
    b1' = b0;
    x' = if b0=b1 then x+1 else x;
    y' = if b0=b1 then y else y+1;
    ok' = ok and (x≥y);
invariant ok;
```

Figure 4: A program with a property

- the declaration of Boolean state variables $(b_i)_{i=1\ldots m}$ and Boolean input variables $(c_j)_{j=1\ldots n}$, and the declaration of numerical state variables $(x_k)_{k=1\ldots p}$ and numerical input variables $(y_\ell)_{\ell=1\ldots q}$
- a set of equations $b'_i = \phi_i(\vec{b}, \vec{c}, \vec{x}, \vec{y})$, $i = 1 \ldots m$ and $x'_k = \psi_k(\vec{b}, \vec{c}, \vec{x}, \vec{y})$, $k = 1 \ldots p$ giving the value of each state variable at the next instant, as a function of the current values of state and input variables. In these equations, expressions ϕ_i and ψ_k are well-formed expressions (of suitable types), possibly mixing Boolean expressions and *linear* numerical expressions. Atoms of Boolean expressions can be either Boolean variables or *linear* constraints.[1]
- two Boolean functions $Init(\vec{b}, \vec{x})$ and $Inv(\vec{b}, \vec{x})$, respectively defining the set of initial states and the invariant (i.e., the safety property to be verified) of the program.

For instance, Fig. 4 shows our example program augmented with an "invariant", which becomes false whenever x < y.

3.3.2 Semantics and Verification Goal

Let us note $\mathbb{B} = \{\mathrm{T}, \mathrm{F}\}$ the set of Boolean values, and $\mathcal{N}$ the set of numerical values (which may be integer, rational, or real numbers). A *state* of the automaton is a pair $(\vec{\beta}, \vec{\xi}) \in \mathbb{B}^{\mathrm{m}} \times \mathcal{N}^{\mathrm{p}}$ of valuations of the Boolean and numerical state variables. A state $(\vec{\beta}, \vec{\xi})$ is initial iff $Init(\vec{\beta}, \vec{\xi}) = T$. The

[1] Linearity is here for simplicity: one can also abstract non linear constraints or assignments.

transition relation is defined by

$$(\vec{\beta},\vec{\xi})\xrightarrow{(\vec{\chi},\vec{\nu})}(\vec{\beta}',\vec{\xi}') \iff \begin{cases} \beta'_i = \phi_i(\vec{\beta},\vec{\chi},\vec{\xi},\vec{\nu}) & i = 1\ldots m \\ \xi'_k = \psi_k(\vec{\beta},\vec{\chi},\vec{\xi},\vec{\nu}) & k = 1\ldots p \end{cases}$$

A run of the automaton is an infinite sequence $(\vec{\beta}^{(i)},\vec{\chi}^{(i)},\vec{\xi}^{(i)},\vec{\nu}^{(i)})_{i\geq 0}$, where $(\vec{\beta}^{(0)},\vec{\xi}^{(0)})$ is an initial state, and $\forall i \geq 0 : (\vec{\beta}^{(i)},\vec{\xi}^{(i)}) \xrightarrow{(\vec{\chi}^{(i)},\vec{\nu}^{(i)})} (\vec{\beta}^{(i+1)},\vec{\xi}^{(i+1)})$.

A state $(\vec{\beta},\vec{\xi})$ is reachable if it belongs to the projection of a run onto state variables. The goal of the verification is to show that, for any reachable state $(\vec{\beta},\vec{\xi})$, $Inv(\vec{\beta},\vec{\xi}) = T$.

In the sequel, $\mathbb{B}^{\mathrm{m}}\times\mathcal{N}^{\mathrm{p}}$ will be denoted by S, the set of inputs $\mathbb{B}^{\mathrm{n}}\times\mathcal{N}^{\mathrm{q}}$ by E, the set of initial states by S_{init} and the set of states violating the invariant by S_{error}. We assume that $S_{init}\cap S_{error} = \emptyset$ [2]. Let τ be the projection of the relation $\longrightarrow$ onto state variables: $\tau(s,s') \Leftrightarrow (\exists e\in E :\ s\xrightarrow{e}s')$.

4 Choosing an abstract domain

Our state space is $S = \mathbb{B}^{\mathrm{m}}\times\mathcal{N}^{\mathrm{p}}$. We have to define an abstract domain A connected to $C = 2^S$ as a basic abstract domain on which dynamic partitioning will be defined. The small example presented in the introduction shows the interest of associating the same polyhedron (or more generally the same abstract value for numerical variables) to several valuations of Boolean variables. Conversely it can be necessary to associate *several* polyhedra to the same Boolean valuation, as shown by the example of Fig. 5.

We assume the existence of a lattice $(L_{\mathcal{N}},\sqsubseteq_{\mathcal{N}},\sqcup_{\mathcal{N}},\sqcap_{\mathcal{N}})$ connected by a Galois connection $(\alpha_{\mathcal{N}},\gamma_{\mathcal{N}})$ to the concrete lattice $2^{\mathcal{N}^p}$. $L_{\mathcal{N}}$ may be any lattice for numerical domains mentioned in the introduction.

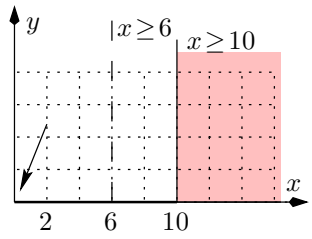
4.1 Definition

We propose to take as an abstract lattice $A = 2^{\mathbb{B}^{\mathrm{n}}}\times L_{\mathcal{N}}$: an abstract value (B,P) is a pair made of a subset of the Boolean space (or a Boolean formula) and an abstract numerical value. We will call these abstract values *convex states*. The meaning function γ is simply the canonical injection, whereas α can be defined as $\forall x\subseteq S,\ \alpha(x) = \sqcap\{y\in A \mid x\subseteq\gamma(y)\}$. This construction is the classical reduced product of abstract domain defined in [CC79], the reduction is here the implicit merging of the different representations of the empty set.

[2] In the opposite case, the property is trivially false.

Consider the following program:

```
state x,y : int;
input ix,iy : bool;
initial (x=0) and (y=0);
transition
    x' = if ix then x+1 else x;
    y' = if x>=10 and iy then y+1 else y;
invariant (y=0) or (x>=6);
```



It has no Boolean state variable and two integer state variables which are incremented when a corresponding input signal is true. Moreover, y *can be incremented only if* x >= 10*. The exact reachable state space is shown on the picture. We want to show that if* $y > 0$ *then* $x \geq 6$*. Obviously, because of the convex hull, if we use only one polyhedron* P *to represent an upper-approximation of the reachable state space, we will obtain* $P = \{(x, y) \mid x \geq 0 \wedge y \geq 0\}$ *and we will not be able to prove the property. Now, if we partition the space according to the constraint* $x \geq 6$ *for instance, the reachable state space will be approximated by the exact union* $P_1 \cup P_2$ *of two polyhedra, where* $P_1 = \{(x, y) \mid 0 \leq x \leq 5 \wedge y = 0\}$ *and* $P_2 = \{(x, y) \mid x \geq 6 \wedge y \geq 0\}$*, which allows to conclude that the property is true.*

Figure 5: A program for which you need an exact union of polyhedra

4.2 Fundamental operations

The definition and implementation of inclusion test, *glb* (greatest lower bound) and *lub* operators are straightforward:

$$\begin{array}{rcl} (B_1, P_1) \sqsubseteq (B_2, P_2) & \Leftrightarrow & (B_1 \Rightarrow B_2) \wedge (P_1 \sqsubseteq_{\mathcal{N}} P_2) \\ (B_1, P_1) \sqcap (B_2, P_2) & = & (B_1 \wedge B_2, P_1 \sqcap_{\mathcal{N}} P_2) \\ (B_1, P_1) \sqcup (B_2, P_2) & = & (B_1 \vee B_2, P_1 \sqcup_{\mathcal{N}} P_2) \end{array}$$

The widening operator is defined in the same way:

$$(B_1, P_1) \nabla (B_2, P_2) = (B_1 \vee B_2, P_1 \nabla_{\mathcal{N}} P_2)$$

where $\nabla_{\mathcal{N}}$ is the standard widening operator on $L_{\mathcal{N}}$. $\mathbb{B}^n$ being of finite depth 2^n, and $\nabla_{\mathcal{N}}$ being a widening operator, $\nabla : A \times A \to A$ is a widening.

Notice that *lub* loses information not only because of the use of the *lub* on the numerical part, but also because Boolean and numerical parts are considered separately; for instance, if $L_{\mathcal{N}}$ is the polyhedra lattice, $(b, x > 0) \sqcup (\neg b, x \leq 0) = (\top, \top) = \top$ instead of the exact result which is $(b \equiv (x > 0))$. Here, a partitioning of S according to b or the constraint $x > 0$ would allow to represent exactly this set.

The transformation of a convex state by our transition functions is a more complex topic and is delayed to the section 6.

4.3 Discussion

The lattice of convex states can only represent simple relations between Boolean and numerical values. Several proposals have been made in the literature to combine Boolean and numerical properties in a more precise way, most of which being based on Binary Decision Diagrams (BDDs). The principle is to use extended BDDs, the atoms of which are either normal Boolean variables, or *pseudo-variables* whose value are interpreted as the satisfaction of a numerical constraint.

For instance, [Mau96] associates to pseudo-variables linear constraints, and can represent arbitrary union of convex polyhedra, or more generally any relation between Boolean variables and numerical constraints; the figure shows such a diagram representing $b \equiv (x > 0)$. By associating *difference constraints* to pseudo-variables (i.e., constraints of the form $x_i - x_j \leq d$) [MLAH99] is able to represent finite union of zones and apply this technique to the verification of timed automata. [BLP+99] proposes a variation on that idea by using n-ary nodes allowing multiple choice depending of the value of a difference $x_i - x_j$.

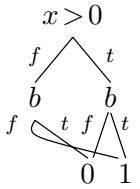


Figure 6: BDD for $b \equiv (x > 0)$

We could have used such diagrams as an abstract domain, but we did not because such abstract domains do not induce *any* approximation (and makes partitioning useless), whereas we are convinced that approximation is the key issue to overcome the complexity of the verification problem. Other objections are first that these diagrams do not exhibit canonicity properties and need to be simplified regularly, a costly operation, and secondly that it is not obvious at all to define a suitable widening operator.

5 Control structures

A control structure is a more operational view of a partitioned system, equipped with some attributes. Let $\mathcal{K} = \{S^{(1)}, \ldots, S^{(|K|)}\}$ be a partition (or a covering) of the state space S of the program. A control structure based on the partition (or covering) $\mathcal{K}$ is an automaton $(K, K_I, K_F, \rightsquigarrow, \mathit{def})$ where

- K is the finite set of locations;

- *def* is the function $\left\{ \begin{array}{ccc} K & \longrightarrow & A \\ k & \longmapsto & \alpha(S^{(k)}) \end{array} \right.$

- K_I and K_F are subsets of K (respectively called sets of initial and final locations), such that

$$S_{init} \subseteq \bigcup_{k \in K_I} \gamma(def(k)) \text{ and } S_{error} \subseteq \bigcup_{k \in K_F} \gamma(def(k))$$

- $\rightsquigarrow$ is a binary relation on K (transition relation), such that

$$\Big(\exists s \in def(k), \exists s' \in def(k') \text{ such that } \tau(s, s')\Big) \;\Rightarrow\; k \rightsquigarrow k'$$

Such an automaton is an abstraction of the program. Each location k represents a set of concrete states $def(k)$ and the transition relation is an upper approximation of the program transition relation. The important point is that, if there is no path in the automaton from an initial to a final location, then no bad state can be reached from an initial state in the program.

5.1 Initial Control Structure

We start from a very coarse control structure, which distinguishes initial states, final (bad) states, and others; ideally, K_I, K_F and $K \setminus K_F$ should cover exactly respectively initial, bad and invariant states. However, we only require exact separation of bad and invariant state:

$$\bigcup_{k \in K_I} def(k) = S_{init} \;, \quad \bigcup_{k \in K \setminus K_F} def(k) = S \setminus S_{error} \;, \quad \bigcup_{k \in K_F} def(k) = S_{error}$$

Before any analysis, the transition relation $\rightsquigarrow$ is assumed to be complete, with two exceptions: locations in K_I are sources, which is sound because initial states are also covered by locations in $K \setminus K_I$, and locations in K_F are sinks, as we are not interested in the future of bad states. Fig. 7 shows the initial control structure for our example program. In this figure, the definition $def(k)$ of each location k is shown in a grey box.

Notice that such a detailed covering of initial, bad and invariant states results from a deliberate choice: on one hand, it starts the analysis with a precise separation of relevant states; on the other hand, the influence of this separation on the analysis cost will not be dramatic: since locations in K_I and K_F never belong to loops, they will not be involved in iterations. Anyway, one can always transform the program such that there is only one

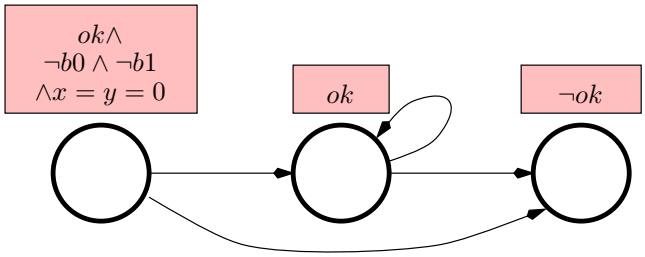


Figure 7: Initial control structure

location of each type. For instance, in the program of Fig 4, the invariance property "x≥y" has been "delayed" into a Boolean state variable ok, and "good" and "bad" states are those in which respectively ok=true and ok=false. A similar transformation can ensure that there is only one initial location.

5.2 Analysis on control structures

We use both forward and backward analyses on control structures, using the underlying abstract domain $A^{\mathcal{K}}$. Forward analysis computes, for each location k, an upper approximation $reach(k)$ of the set of states in $def(k)$ that are reachable from an initial state of the program. Backward analysis computes, for each location k, an upper approximation $coreach(k)$ of the set of states in $def(k)$ from which one can reach a final state (these states are said to be "coreachable" from final states). The equations for $reach(k)$ and $coreach(k)$ are respectively

$$\begin{cases} k \in K_I & reach(k) = def(k) \\ k \notin K_I & reach(k) = \bigsqcup_{k' \rightsquigarrow k} post(reach(k')) \sqcap def(k) \end{cases}$$

$$\begin{cases} k \in K_F & coreach(k) = def(k) \\ k \notin K_F & coreach(k) = \bigsqcup_{k \rightsquigarrow k'} pre(coreach(k')) \sqcap def(k) \end{cases}$$

assuming, for any $X \in L$, that

$post(X)$ is an upper-approximation in L of $\{s' \in S \mid \exists s \in X \ : \ s \rightarrow s'\}$

$pre(X)$ is an upper-approximation in L of $\{s \in S \mid \exists s' \in X \ : \ s \rightarrow s'\}$

The solutions are computed by standard fixpoint computation, using chaotic iterations, and application of the widening operator ∇ to ensure convergence, as described in section 3.2.

Restricting the considered state space. Remember that our goal is to show that there is no path from initial to final states: we are interested only in states that are both reachable from initial states, and coreachable from final states, which we call *dangerous* states. So we discard other states in the control structure:

- After a forward analysis, we update the relation "$\rightsquigarrow$" as:

$$k \rightsquigarrow k' \;\Leftrightarrow\; post(reach(k)) \sqcap reach(k') \neq \bot$$

  and we take $def(k) = reach(k)$;

- After a backward analysis, we update the relation "$\rightsquigarrow$" as:

$$k \rightsquigarrow k' \;\Leftrightarrow\; pre(coreach(k')) \sqcap coreach(k) \neq \bot$$

  and we take $def(k) = coreach(k)$.

As a consequence, $def(k)$ represents an upper-approximation of the set of dangerous states in location k, and $S = \bigcup_{k \in K} \gamma(def(k))$ becomes the global set of dangerous states after analysis.

Forward and backward analyses are combined by performing them in alternation, until convergence. Fig. 8.(a) shows in an oval box for each location the result obtained by the forward analysis on the initial control structure shown on Fig. 7. Then, the definitions of locations are updated, as shown on Fig. 8.(b). Backward analysis doesn't give any additional information on this new control structure.

6 Approximate pre- and post-condition operator

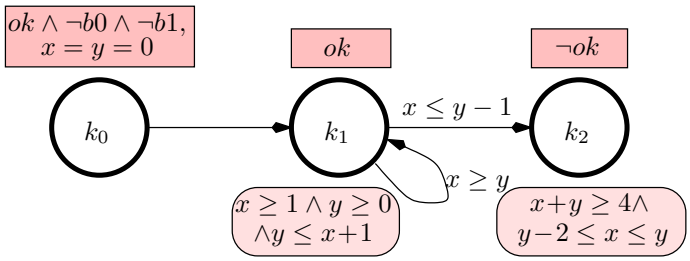
We describe here our technique to compute pre- and post-condition of abstract values by transition functions. The difficulty comes from the composite nature of convex states and transition functions.

6.1 An introductive example

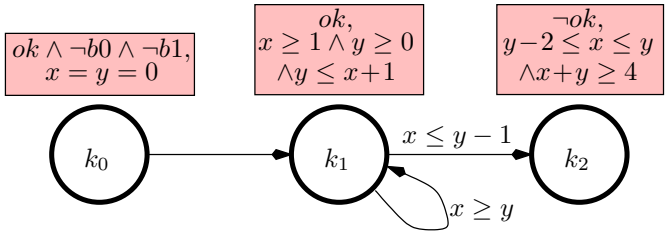
Let us start with an example: consider a program with a Boolean state variable b and an integer one x, the transition function of which is

$$\begin{cases} \phi_b &= b \wedge (x \leq 2) \\ \psi_x &= \textbf{if } b \wedge (x \leq 2) \textbf{ then } x + 10 \textbf{ else } x - 1 \end{cases}$$

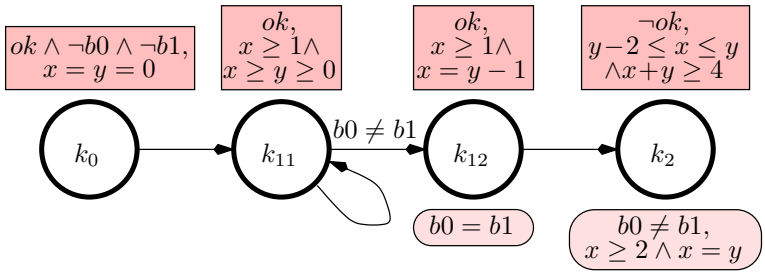
Let τ be the transition relation induced by the above transition functions. Assume we want to compute $post(\tau)(X)$ with $X = (\top, x \in [0, 5])$, and that states are partitioned according to $x \leq 8$.



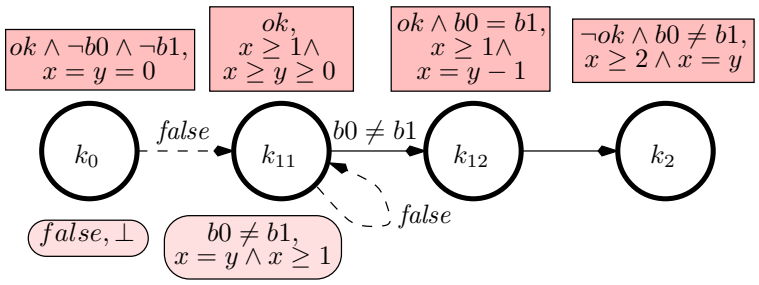
(a) Forward analysis on the initial structure



(b) Updating of the definitions



(c) Refinement + forward analysis



(d) Updating + Backward analysis

Figure 8: Analysis of the simple example

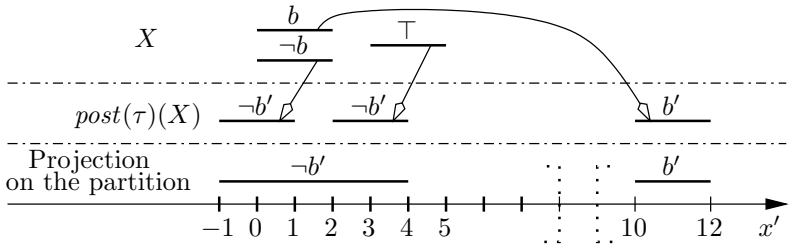


Figure 9: Computation of a post-condition

6.1.1 The exact technique

If we want to compute the best correct approximation of $post(\tau)(X)$, we can decompose X according to the value of b and $x \leq 2$, in order to remove:

- any numerical constraints from Boolean transition functions;
- any literal from numerical transition functions.

This decomposition is necessary because functions are composite and we only know how to compute directly the image of a purely Boolean function and the image of a non-guarded linear assignment. We obtain here, Fig. 9,

$$\begin{array}{rccccccc} X = & (\neg b, x \in [0,2]) & \cup & (\top, x \in [3,5]) & \cup & (b, x \in [0,2]) \\ post(\tau)(X) = & (\neg b', x' \in [-1,1]) & \cup & (\neg b', x' \in [2,4]) & \cup & (b', x' \in [10,12]) \end{array}$$

We project then this result on the partition and apply possibly convex union: $(\neg b', x' \in [-1,4]) \cup (b', x' \in [10,12])$.

Such a technique has two main drawbacks:

1. The size of decomposition grows exponentially with the number of conditions appearing in the transition functions;
2. When the size of the partition is small, it is a bad idea to compute with accuracy the postcondition, knowing that several convex unions will be applied afterwards that may lose a lot of information.

6.1.2 A compilation technique

We will compute postconditions using the Lustre compiler technique [Ray91], which generates a sequential version of the transition function: (state) variables are sorted according to their dependencies, a reduced number of auxiliary variables are introduced if necessary (for instance for the

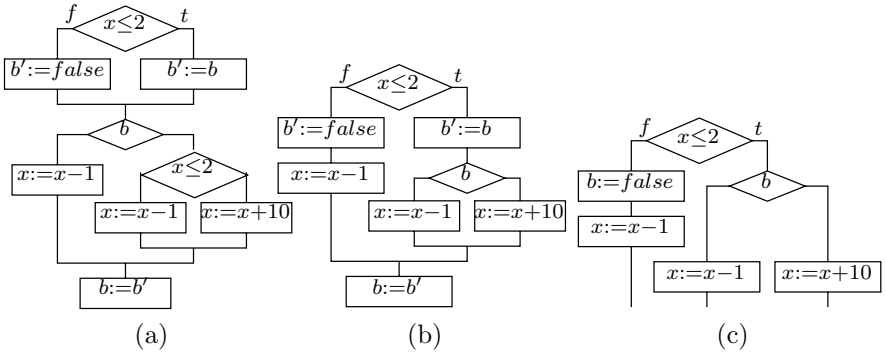


Figure 10: Three possible sequences for our transition function

exchange of two variables), and the postcondition is progressively computed by a sequence of tests and assignments. For the example above, three correct *sequences* would be given by the flowcharts of Fig. 10.

When using such a sequence for computing a postcondition, test opening corresponds to splitting the input abstract value according to the condition of the test, whereas test closing corresponds to merging (by convex union) the results of the two branches. The motivation to minimize the number of auxiliary variables comes from the fact that introducing systematically primed variables is known to be expensive in BDDs computations, and that the complexity of polyhedra operations is exponential w.r.t. the number of variables. The drawback of this choice is that dependency constraints between operations are tighter.

This technique allows us to apply least upper bounds not only on locations of the control structure but also while computing postconditions, and to make the precision vary. Here the second sequence (Fig. 10(b)) can be considered better, because the test on $x \leq 2$ is shared between the two operations and the least upper bound is thus applied less frequently. We say that the test on $x \leq 2$ is *factorized* between the two operations. Notice that if we never close tests (Fig. 10(c)), we obtain the exact technique described in the previous paragraph.

6.2 Definition of sequences and associated operators

Sequences have the following syntax:

$$
\begin{array}{rcl}
\langle \text{sequence} \rangle & ::= & \langle \text{operation} \rangle ; \ldots ; \langle \text{operation} \rangle \\
\langle \text{operation} \rangle & ::= & \langle \text{action} \rangle \\
& | & \textbf{if } \langle \text{cond} \rangle \textbf{ then } \langle \text{sequence} \rangle \textbf{ else } \langle \text{sequence} \rangle
\end{array}
$$

Actions are operations that can be computed directly: purely Boolean assignments or non-guarded linear assignments; *conditions* are formulas partitioning any convex state into two ones: purely Boolean formula or single linear constraints. A sequence s defines approximate post- and pre- condition operators:

$$\begin{array}{rcl}
post(op_1;\ldots;op_n)(X) & = & post(op_2;\ldots;op_n)\circ post(op_1)(X) \\
pre(op_1;\ldots;op_n)(X) & = & pre(op_1;\ldots;op_{n-1})\circ pre(op_n)(X) \\
post(\textbf{if } c \textbf{ then } s^+ \textbf{ else } s^-) & = & post(s^+)(X\sqcap c)\ \sqcup\ post(s^-)(X\sqcap\neg c) \\
pre(\textbf{if } c \textbf{ then } s^+ \textbf{ else } s^-) & = & (pre(s^+)(X)\sqcap c)\sqcup(pre(s^-)(X)\sqcap\neg c) \\
post(act)(X), pre(act)(X) & : & \text{terminal cases}
\end{array}$$

6.3 Data structures for transition functions

In order to represent transition functions, we use the diagrams mentioned in section 4.3:

Boolean functions are represented by BDDs, the atoms of which are either Boolean variables of the program, or pseudo-variables associated with linear inequalities of the program.

Numeric functions are represented in the same spirit by MTBDDs (Multi-Terminal BDDs), the set of atoms of which is the same than for Boolean functions and the leaves of which are affine expressions on numerical variables of the program.

Such diagrams are well-suited for the operations we will use to build sequences and compute pre- and post-conditions. Our tool uses the CUDD package [Som].

6.4 Basic sequentialisation algorithm

We call *sequentialisation* the operation consisting in generating a sequence for a transition function. We use a more complex algorithm than in [Ray91], but the cost remains still reasonable w.r.t that of fixpoint computation. We suppose that auxiliary variables have been introduced, and that we have an acyclic graph of operations as input to the algorithm. An edge $s_i \rightarrow s_j$ between two variables and their associated operations indicates that the assignment of s_i should take place before the assignment of s_j. This sequentialisation operation can then be sketched as follows:

Scheduling of actions: we put first in the sequence the operations that can be generated directly, i.e., the actions that have no predecessors in the graph; we pick them until there is not such actions in the graph any more; at this point, a test has to be opened;

Selection of a test condition: we have first to select a condition on which opening the test;

Selection of operations under test: then we have to select the operations we will schedule in the two branches of the test. The graph of selected operations $\mathcal{O}$ are partially evaluated by the condition and its negation, giving respectively the graphs $\mathcal{O}^+$ and $\mathcal{O}^-$. Partial evaluation is done with the *cofactor* operations on BDDs and MTBDDs, and it may relax some dependencies in these graphs.

Recursive calls: the algorithm is finally applied recursively to the graphs $\mathcal{O}^+$ and $\mathcal{O}^-$ under the branches of the test, and also to the graph of non selected operations, which are scheduled after the test closing.

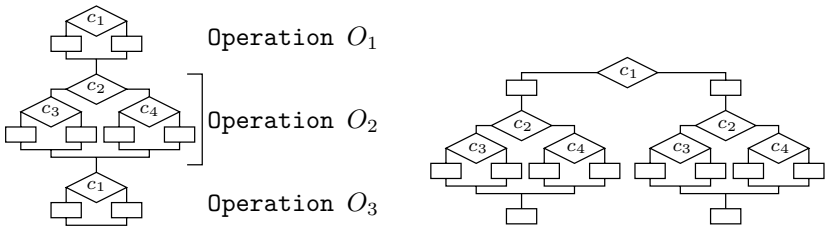
The most important part of the algorithm is the selection of the condition and the selection of operations under test heuristics:

- In order to select a condition, we consider the set of operations without predecessors in the graph, and we select a condition by choosing the one that appears in the biggest number of these operations. This is implemented by considering the *support* of corresponding diagrams, i.e., the set of atoms that appear in them.
- We select operations under the test by picking iteratively operations without predecessors, that are actions or that depend on the test condition, until this process is blocked. We don't allow the selection of operations that don't depend on the condition in order to avoid "duplication" of sequences, as in the Fig. 11.

This algorithm represents a good tradeoff between the factorization of test between different operations, which favorized accurate results, and the size of the sequence.

6.5 Refining sequences and improving the precision

Approximations are valuable only if we know how to control them. We describe how we link the precision induced by sequences to the fineness of control structure.



(a) Our choice: no duplication

(b) Factorization of test on c_1 but duplication of operation O_2

Figure 11: Factorization v.s. duplication in sequences. O_2 is supposed to be scheduled between O_1 and 0_3.

6.5.1 Principle

Using a single sequence, computing the post- and pre- condition of any state $X \in A$ would be too unprecise. An obvious idea is to take advantage of the control structure to specialize sequences, taking into account the definitions $def(k)$ of locations k. This can be done by partially evaluating the transition functions on $def(k)$ before applying the sequentialisation algorithm. Partial evaluation may remove conditions from the transition functions and may also remove some dependency constraints: it makes sequentialisation easier and favorizes the factorization of tests, which increase the precision of the associate operators.

Remember that our motivation, in approximating post- and pre-condition operators, was to avoid too precise computations when the roughness of the control structure makes them useless. Specializing sequences on definitions allows the precision of the sequence to be linked with the precision of the control structure: globally, as the analysis progresses, the global dangerous state space becomes smaller; locally, higher is the number of locations, smaller are their definitions. In both cases, the precision induced by the sequence is improved.

We will actually specialize sequences according to their destination location as well, and combine forward and backward propagation, as for the analysis of control structures. As a consequence, they will be attached to the edges of the control structure.

6.5.2 Refinement by forward and backward propagation of definitions

Consider again our introductory example. Fig. 12(a) represents the sequence of Fig 10(a) joining the two locations of the partition (the big circles), the definition of which is given by the grey boxes. The small circles are intermediate states, called *micro-states*, which correspond to intermediate steps during the computation of a reaction. We will use for micro-states similar notions of *definition*, *reachable part* and *coreachable part* than for locations. Initially, just after the generation of the sequence, the definition of micro-states is the top element, $\top$.

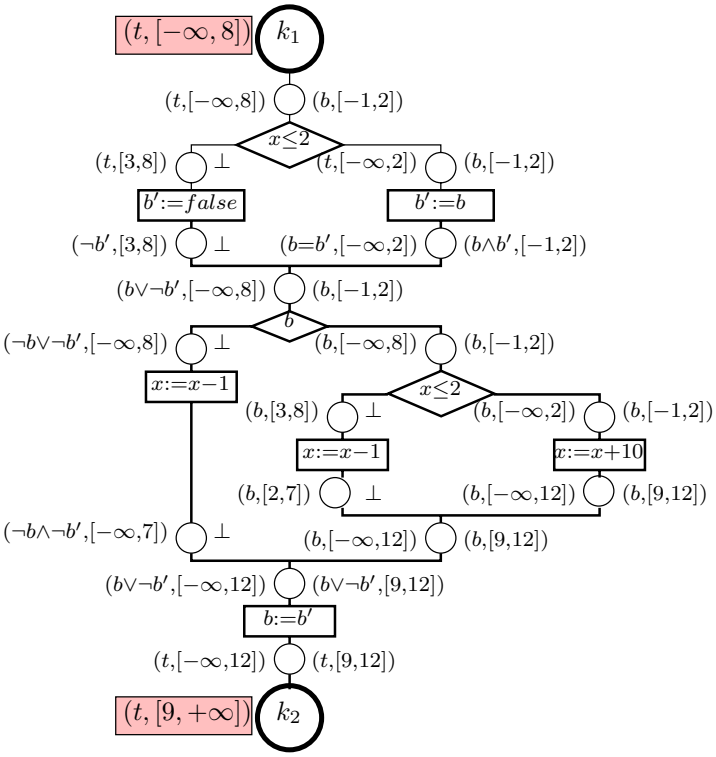
A necessary condition for a state of location k_1 to lead to location k_2 by a postcondition is $b \wedge (x \leq 2)$; obviously, one could take advantage of this fact to replace the tests by a simple intersection. Unfortunately, a normal pre-condition computation cannot discover this fact because of least upper bounds. To achieve this goal, we need to use forward and backward propagation of definitions through micro-states:

- first, we will compute $post(seq(k_1 \rightsquigarrow k_2))(def(k_1))$ and propagate corresponding values on micro-states, thus computing their reachable part from location k_1, which becomes their new definition. In Fig. 12(a), those are given by convex states on the left of micro-states;
- then, we will compute $pre(seq(k_1 \rightsquigarrow k_2))(def(k_2))$, and intersect the intermediate values with the new definitions of corresponding micro-states, thus computing their coreachable part; definitions are then updated; on the figure, these new definitions are given by convex states on the right of micro-states.

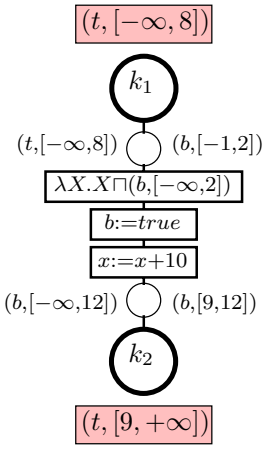
Micro-states the definition of which becomes empty, correspond to "dead" branches that can be discarded. By considering conditions guarding such branches, we can associate with each transition $k \rightsquigarrow k'$ of the control structure a formula $ast(k \rightsquigarrow k')$, called *assertion*, which is a necessary condition for the transition to be taken. In our example, we have $ast(k_1 \rightsquigarrow k_2) = b \wedge (x \leq 2)$. Now a partial evaluation of the transition functions according to this formula, followed by a sequentialisation give the new sequence of Fig. 12(b), which is much simpler and moreover gives exact results.

6.5.3 Partial evaluation method

Partial evaluation has been mentioned several times, in this section. Let us make precise how it is applied: partial evaluation is not performed according



(a) Sequence before refinement



(b) Sequence after refinement

Figure 12: Sequences and micro-states

to convex states, but to formulas, and more precisely to assertions, represented with BDDs. Indeed, BDDs make partial evaluation easy and very powerful: it is done by applying a generalized cofactor operator, such as the "restrict" operator [CBM89], or the one presented in [Ray91], which is very efficient to reduce the support of BDDs, although much more expensive. We use one of those two operators to simplify transition functions by assertions before building the corresponding sequence.

6.5.4 Conclusion about sequences

Now a control structure will be equiped with assertions (represented by BDDs) and sequences, both associated to transitions. On the initial control structure, we have $ast(q \rightsquigarrow q') = def(q)$. Each time we build the corresponding sequence $seq(k \rightsquigarrow k')$, this assertion is taken into account. Then we propagate definitions of locations k and k' as described in the previous section. If we discover dead branches, we strengthen the assertion and we repeat the operation.

During fixpoint analysis, we intersect intermediate values computed by post- or pre-condition operators with the definitions of micro-states, and we memorize their reachable (forward analysis) or coreachable (backward analysis) part.

Finally, after an analysis, definitions of locations and micro-states are updated, and, when micro-states of a sequence becomes empty, assertions are updated and the sequence is recomputed. Notice that the result of an analysis is now not only given by the new definitions, but also by the new assertions.

The technique presented here for computing post- and pre-condition is powerful: it speeds up computations without introducing too much loss in precision. Moreover, as the control structure becomes finer and induces a better precision in analysis, sequences are refined as well.

7 Control structure refinement

7.1 Principles

Control structure refinement is needed when the analysis on the initial control structure doesn't succeed to prove the property, which usually happens because that control structure is too rough. Remember that our goal is to show that there are no dangerous states, or equivalently that there is no

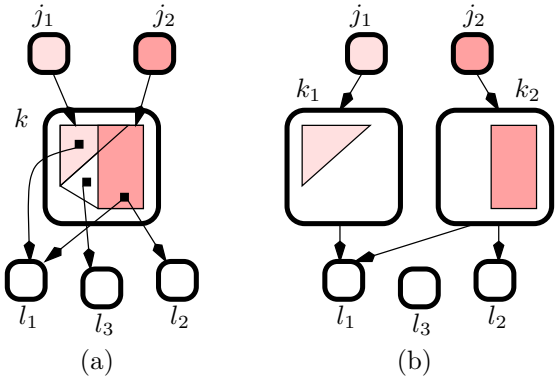


Figure 13: Refining a location: principle

path from initial to final location in the control structure:

$$\forall k, \mathit{def}(k) = \bot, \text{and} \rightsquigarrow = \emptyset$$

We suppose that the analysis fails because of approximations induced by the least upper bound.

The goal of our refinement method is to separate states that exhibit different behavior in term of accessibility, and to refine the abstract transition relation $\rightsquigarrow$. For instance, consider the fragment of structure shown on Fig. 13(a). Let us note, for $n = 1, 2$, $Y_n = \mathit{post}(\mathit{def}(j_n)) \sqcap \mathit{def}(k)$ (which are resp. the triangle and the rectangle of the figure), and let X be the set of states added by the least upper bound: $X = \gamma(Y_1 \sqcup Y_2) \setminus (\gamma(Y_1) \cup \gamma(Y_2))$. Assume, as shown on the figure, that:

$$\begin{array}{rcl} \mathit{post}(Y_1) \sqcap \mathit{def}(l_2) & = & \mathit{post}(Y_1) \sqcap \mathit{def}(l_3) \quad = \quad \bot \\ \mathit{post}(Y_2) \sqcap \mathit{def}(l_3) & = & \bot \\ \mathit{post}(\alpha(X)) \sqcap \mathit{def}(l_3) & \neq & \bot \end{array}$$

In such a case, splitting k into k_1 and k_2 with $\mathit{def}(k_i) = Y_i$, as shown in Fig. 13(b), has three desirable effects:

- it suppresses a least upper bound, thus allowing the exact representation of $Y_1 \cup Y_2$; moreover, the set of states X added by the least upper bound is no longer propagated in the control structure;
- it suppresses the transition $k \rightsquigarrow l_3$, which was only enabled because of the set X;

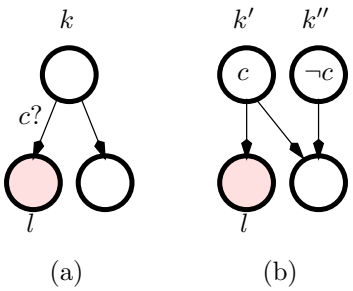


Figure 14: Refining a location by removing a transition

- it suppresses also *paths* in the control structure; for example after refinement the path $j_1 \rightsquigarrow^* l_2$ is removed; remember that we have to cut any paths between initial and final locations.

Now such refinements must be performed with care: arbitrary refinements may uselessly complexify the control structure, and even result in infinite computations.

7.2 Location refinement by means of transition refinement

In this section, we explain the kind of refinements implemented in our tool. To avoid the risk of infinite refinements, we adopt the following restriction: locations will only be split according to *atomic conditions* (Boolean variables, or linear inequalities) appearing in the program. This still allows quite detailed control structures.

Our heuristic in fact tries to refine transitions. Let k and l be two locations such that $k \rightsquigarrow l$, Fig. 14(a). Assume there exists a condition c appearing in the program, such that $def(k) \sqcap pre(def(l)) \sqsubseteq c$; in other words, c is a necessary condition for the transition $k \rightsquigarrow l$ to be taken. Then k can be safely split into two locations k' and k'', with $def(k') = def(k) \sqcap c$ and $def(k'') = def(k) \sqcap \neg c$, and $k'' \not\rightsquigarrow l$, Fig. 14(b): the refinement suppresses at least one transition in the control structure and allows more accurate analysis, because states in $def(k)$ satisfying or violating the condition c will no longer be merged with the least upper bound.

Notice that this case is rather common, since transitions are generally guarded by conditions, which naturally separate preconditions. Moreover, with our technique, such conditions, that we call *splitting conditions*, are easily obtained by considering the assertions of transitions leaving the considered location.

Let us illustrate this heuristic by detailing the analysis of our simple example (Fig. 2). The initial control structure was shown by Fig. 7, and Fig. 8(b) showed the results of the analysis on this structure. Transitions are labelled with non trivial assertions. Now consider the location k_1. Its states can either lead directly to the final location k_2 if $x < y$, or otherwise loop on k_1. Intuitively, the first ones may be viewed as more dangerous than the second ones, and they should be separated. The location k_1 is thus split according to the inequality $x \geq y$. Fig. 8(c) shows the result of forward analysis on the refined structure. Finally, backward analysis disconnects the initial location (Fig 8(d)), thus completing the proof of the property. Notice that initially we wanted to separate states where $b0 = b1$ from the others (Fig. 3(c)); we have effectively reached this goal, but in an indirect way by splitting the invariant location according to $x \geq y$.

This example is trivial, but it shows that the general approach is successful: the verification steps remain exactly the same if we generalize the program into a ring of n cycles of alternate incrementations, or/and by adding more counters incremented alternately. This meets our goal that the verification be independent of details irrelevant to the satisfaction of the property.

7.3 Implementation of the refinement heuristic

Now, the effective application of such refinements has to be detailed. First, we must design a strategy to choose the locations and the transitions to be refined. Next, since the splitting conditions obtained by the first step are not atomic in the general case, we need a strategy to extract *splitting atoms* from such conditions. Finally, we describe the splitting process itself, and we discuss the combination of the analysis with the refinement process.

7.3.1 Choice of locations and transitions to be refined.

How to apply the refinement when the control structure has several locations and, for each location, several outgoing transitions, like in Fig. 15 ? A first approach is to try to cut transitions between invariant and final locations, as in the example above. But this is not always efficient:

- since we want to cut paths from initial to final locations, *all* edges belonging to these paths are dangerous and are candidates for refinement;
- most penalizing approximations often take place in internal loops of the control structure, and are then propagated in the whole graph.

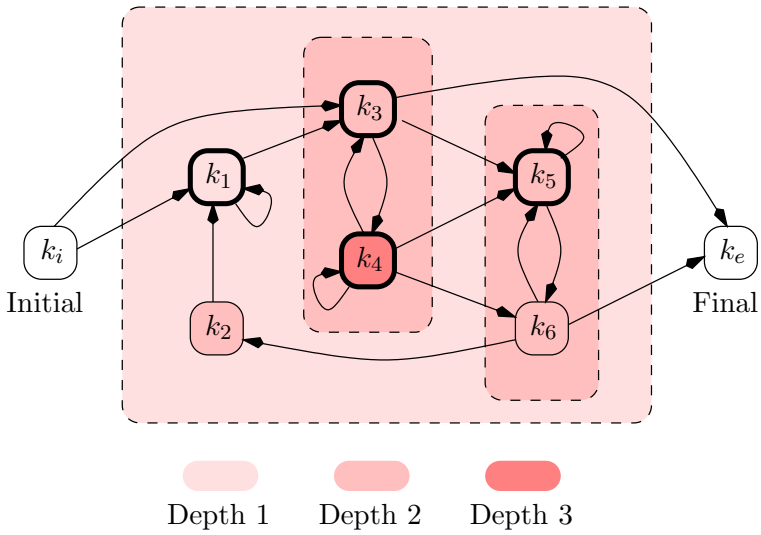


Figure 15: A control structure and its decomposition in strongly connected subcomponent

As a consequence, we decompose the graph into its *strongly connected subcomponents* (SCSCs) using the algorithm of [Bou93], the principle of which is, first, to decompose the graph into its strongly connect components (SCCs), then to apply recursively the decomposition to the components without their heads. The obtained decomposition is not unique, since the head of an SCC depends on the order of the depth-first exploration of the graph. In Fig. 15, components are enclosed by dashed boxes, and their "depth" is indicated by the level of grey. Thick locations are heads of subcomponents. Such a decomposition provides a notion of more or less internal transitions between locations.

Now, our refinement algorithm first tries to disconnect strongly connected components from each other, since this process is the most likely to cut paths between initial and final states. If it does not find any splitting condition, it tries to disconnect from each other subcomponents of increasing depth.

The algorithm always considers a single location, trying to remove its outgoing transitions. For instance, in Fig. 15, suppose we want to cut transition between subcomponents of depth 1 and we consider location k_4: we will consider the two transitions $k_4 \rightsquigarrow k_5$ and $k_4 \rightsquigarrow k_6$, and take the union of their assertions $ast(k_4 \rightsquigarrow k_5) \vee ast(k_4 \rightsquigarrow k_6)$, which is a necessary condition to leave the subcomponent $\{k_3, k_4\}$ from the location k_4. Such a condition

is called a *splitting formula*.

7.3.2 Extraction of a splitting atom from a splitting formula.

We only roughly explain this process, which is rather technical. In order to obtain binary splits, splitting is done only w.r.t. *atoms*, i.e., inequalities or Boolean variables, whereas splitting formulas are general formulas. Assume $(b \vee (x \geq 0)) \wedge (y \geq 0)$ is a necessary condition for a set of transitions to be taken. We can choose any of the atoms b, $x \geq 0$ or $y \geq 0$, but $y \geq 0$ is the best choice because it is still a necessary condition. If there are several such atoms in factor, we choose one arbitrarily. A good heuristic is to favorize Boolean variables and to choose the first one according to the BDDs order. In case no atomic necessary condition exists — as in the formula $b \vee (x \geq 0)$ —, we choose arbitrarily an atom, but we are no longer sure to remove a path in the graph.

7.3.3 Splitting the location w.r.t. a splitting atom.

Given a location k that we must split according to an atom c, we create two new locations k' and k'' with $def(k') = def(k) \sqcap \alpha(c)$ and $def(k'') = def(k) \sqcap \alpha(\neg c)$. Incoming sequences of k are duplicated and refined by backward propagation of micro-states, whereas for every outgoing transitions $k \rightsquigarrow k_s$, we strengthen assertions: $ast(k' \rightsquigarrow k_s) = ast(k \rightsquigarrow k_s) \wedge c$ and $ast(k'' \rightsquigarrow k_s) = ast(k \rightsquigarrow k_s) \wedge \neg c$, and we compute new sequences $seq(k' \rightsquigarrow k_s)$ and $seq(k'' \rightsquigarrow k_s)$.

7.3.4 Combination of the analysis with the refinement process.

When the analysis fails in proving the property, we have clearly to refine the control structure to increase the precision. We should however choose between splitting only one location between two analyses, or many. The former possibility may avoid unnecessary blow-up of the automaton, as a single refinement followed by an analysis can eliminate the apparent need of other refinements, but has the drawback of increasing the number of analyses. Our algorithm tries to split all locations for which it finds a splitting formula before applying a new analysis.

7.4 Related works

The refinement method proposed here presents similarities symbolic bisimulation [BFH+92, BdS92] that has been for instance applied to timed au-

tomata [ACD+92, YL93, STA98]; we use indeed preconditions to splits elements of the partition. The main difference is that in-between we perform fixpoint analysis on the *abstract domain* with propagation over the whole graph, whereas in these methods one deduce reachability informations only by doing one step precondition computations, followed by *discrete* reachability analysis of the obtained automaton. Another difference is that we combine forward and backward analysis: the obtained partition has nothing to do with bisimulation and is usually much coarser.

Another kind of refinement criteria would be to measure directly the loss of information induced by the least upper bound operator. For instance, given two polyhedra P and Q, we could compare the volume of $P \sqcup_{\mathcal{N}} Q$ with the sum of the volumes of P and Q. But this idea is not practical for several reasons: volume computations in high dimension are very expensive, most polyhedra we compute are not bounded, and their dimension vary: how to compare a small cube with an infinite line ? [DWT95] uses a criterion of that kind in order to verify timed automata; the main point is the ability to perform *under*-approximations. Let's give the intuition of the proposed refinement: an under-approximate backward analysis, for instance, computes a set of states that *necessarily* lead to the violation of the property. Now, during an over-approximate forward analysis, if we compute $P \sqcup Q$, with P and Q not containing any such *surely* dangerous states, and that the result contains one of these states, the $\sqcup$ operator should be avoided. Notice that, with convex polyhedra, it is possible to afford the under approximate of *dense* sets, like regions of timed automata, but not of the *discrete* sets considered in our case.

Some works combining theorem proving and model-checking [SUM96, GS97, BLO98, CU98] also use this idea. The basic idea of this combination is the following: the state space is first partitioned, then theorem proving is used to test the fireability of abstract transitions linking members of the partition, and possibly to generate auxiliary invariants [BBM97, BLS96]; finally model checking is used to explore the obtained finite graph. As usual, the proof may fail because of approximations. In that case, [CGJ+00] proposes to start the backward generation of a counter-example trace; such a generation is a particular case of under approximate backward analysis and may be used in the same way for the refinement of the partition.

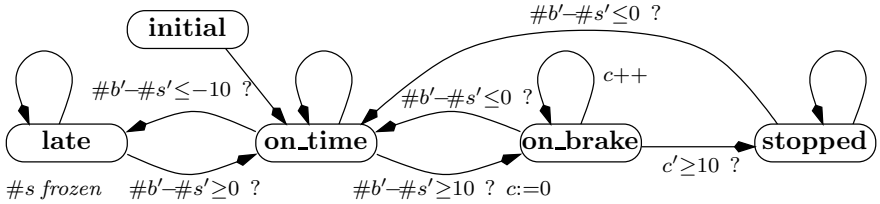


Figure 16: Automaton for one subway

8 Implementation and experimental results

8.1 The verification tool NBac

We implemented the method and algorithms described above in the verification tool NBac, whose size is about 10000 lines of OCaml code [CAM] and 3000 lines of C code. This tool takes as input a Lustre synchronous program together with the observer of the safety property to check [HLR93], performs a so-called cone of influence analysis, translates the result into the format of the section 3.3, and builds an initial control structure for the system. Then it starts the dynamic partitioning process, up to either success or to a point where no further refinement can be performed. In this last case, we have not implemented yet a technique that attempt to extract a diagnosis trace from the obtained automaton. The tool uses the BDD library CUDD [Som] to handle formulas and the recently improved polyhedra library [Jea] to handle convex polyhedra.

8.2 A subway speed regulation system

We first illustrate the behavior of our tool on the subway example taken from [HPR97]. This example is extracted from an actual proposal for an automatic subway. It concerns a (simplified version of a) speed regulation system avoiding collision. Each train detects beacons that are placed along the track, and receives the "second" from a central clock. Ideally, a train should encounter one beacon each second. So the space left between beacons rules the speed of the train. Now, a train adjusts its speed as follows: let $\#b$ and $\#s$ be respectively the number of encountered beacons and the number of received seconds.

- When $\#b \geq \#s + 10$, the train notices it is early, and puts on the brake as long as $\#b > \#s$. Continuously braking makes the train stop before encountering 10 beacons (using a counter `c`).

Prop	Complexity				Verification					
	bool	num	cont	auto	bdd	ana	div	m. size	time	mem
sub1-1	4/2	3/0	11	8	2.6e3	3	0/3	6/13	0.9s	-
sub1-2	3/2	4/0	13	5	2.5e3	3	0/2	5/9	1.3s	-
sub2-1	6/3	5/0	22	29	1.2e4	3	1/2	6/14	12.6s	46
sub2-2	5/3	6/0	24	17	1.3e4	4	1/5	9/27	15.8s	54
sub2-3	5/3	5/0	23	17	1.1e4	4	1/6	10/37	9.1s	37
sub2-4	5/3	5/0	23	17	1.1e4	5	3/10	16/67	14.3s	39
sub3-1	8/4	7/0	33	113	1.6e5	3	1/2	6/14	5m27s	188
sub3-2	7/4	8/0	35	65	9.6e4	4	2/4	9/29	4m53s	192
sub3-3	7/4	7/0	34	65	9.3e4	4	2/5	10/39	10m10s	200
sub3-4	7/4	7/0	34	65	9.3e4	5	6/8	17/93	14m53s	200

Table 1: Verification of the subway system

- When $\#b \leq \#s - 10$, the train is late, and will be considered late as long as $\#b < \#s$. A late train signals it to the central clock, which does not emit the "second" as long as at least one train is late.

Fig. 16 shows the automaton for one subway, after removing the unfeasible transitions. The transition functions of $\#b$ and $\#s$ are not displayed, and the guards are given using primed variables (substituting to them their transition functions leads to more tricky conditions).

This system does not exhibit many Boolean variables, and actually the classical technique of [HPR97] performs well. We aim, here, at showing that our refinement heuristic is powerful enough to separate relevant states to prove properties without introducing too many locations. We wish to prove the following properties, on a system with n trains sharing the central clock

1. A train cannot move in one step from the state **late** to the state **on_brake**, and conversely. If there are several trains, we prove it for the first one.

2. The number $\#b - \#s$ remain in the interval $[-10, 20]$.

3. When there is at least two trains, their respective numbers of beacons, $\#b_1$ and $\#b_2$ satisfy $\#b_2 - \#b_1 \leq 40$. This means that if they are initially separated by 40 beacons, they will never collide.

4. Same property, but with the real bound: $\#b_2 - \#b_1 \leq 30$.

Experiments were conducted on a 600 MHz pentium III processor with 2GB RAM. The results are shown in Table 1, where 'sub$n$-p', n is the number

bool	number of Boolean variables (state/input)
num	idem for numerical variables
cont	number of linear inequalities in the source of the program
auto	size of the Boolean automaton reduced by Boolean bisimulation
bdd	maximum number of alive BDD nodes, as indicated by the library CUDD
ana	number of analysis and refinement step performed
div	number of division on Boolean variables/numerical inequalities
m. size	maximum size of the current control structure (locations/transitions)
time	overall verification time, measured by `time` command
mem	estimation of the memory consumption in MB, using `top` command

Table 2: Meaning of the columns in benchmarks

of trains and p the property to prove. The meaning of the columns is given by Table 2. We can make the following remarks:

- We succeeded in proving *all* properties, and always with a control structure that is smaller than the Boolean automaton; notice also that many splits are done w.r.t. numerical inequalities.
- The verification time increases very quickly with the number of trains, even for properties that explicitly involve only one train. This comes from the fact that all trains are linked by the central clock, and that transition functions become quickly very complex. Another reason is that the cost of polyhedra operations is exponential w.r.t. the number of numerical variables.
- Comparison of properties 3 and 4 shows that the first one, which is intuitively easier to prove, requires less time and refinement to be proved than the second one. Such a behavior is indeed a very nice property of the tool.

We made also some experiments on the sequentialisation method. The use of an *exact* method to compute post- and pre- conditions multiplies by 4 the verification time with $n = 2$, and that we did not succeed in proving properties in this way with $n = 3$.

8.3 A lift controller

As a more challenging example we choosed the lift controller found in [Bou99] and depicted on Fig. 17. This example exhibits complex Boolean behavior,

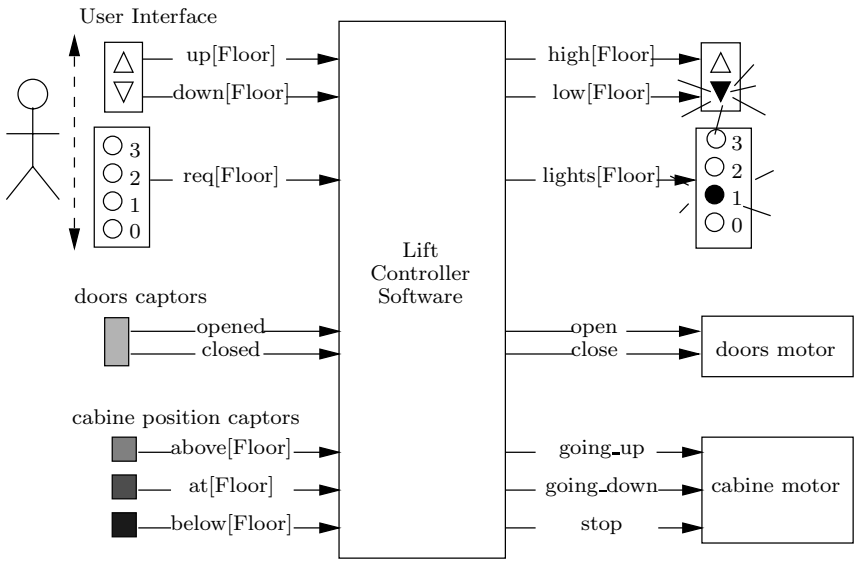


Figure 17: The lift controller and its environment

while containing some counters, and represents a good target for our tool. The lift has N floors. At each floor, there are "up" and "down" request buttons, and in the cabin N request buttons allowing a user to choose the destination floor. The requests are buffered, which results in a quite complex control. The controller has also to set information lights adequately. Concerning the cabin, opening and closing the door are modelled, as well as its moves, decomposed by half floors for the departure and the arrival.

We have modified the original program by adding counters to model the time taken by the lift to move up or down (from one floor, or half a floor for departure and arrival), and the time needed by the door to be opened or closed.

Most logical properties can be proved on this system with a model checker such as Lesar [RHR91] by considering its Boolean abstraction. We concentrated thus on the following time properties:

1. Lower bound for the delay between closing of the doors at floor 0 and opening at floor 1.

2. *Idem* as the last one, but with the *real* bound.

3. Lower bound between two departures of the cabin from the first floor. This implies that in between, the cabin went up then down.

4. *Idem* as the last one, but the first floor is replaced by the second one, from which the cabin can go first up *or* down.

The size of the LUSTRE program with these properties is about 700 lines, and the results are given Tab. 3, with N varying from 3 to 5.[3] The number of Boolean state variables is several dozens, and there are several input variables.

Prop	Complexity				Verification					
	bool	num	cont	auto	bdd	ana	div	m. size	time	mem
lift3-1	26/7	3/0	6	497	3.8e5	6	18/9	30/75	16.9s	52
lift3-2	26/7	3/0	6	497	3.8e5	8	44/26	73/188	21.5s	53
lift3-3	26/7	3/0	6	489	3.9e5	2	1/0	4/6	15.7s	72
lift3-4	26/7	3/0	6	489	2.5e5	4	3/4	10/26	21.2s	69
lift4-1	32/10	3/0	6	3235	8.3e5	4	5/2	10/26	1m03s	90
lift4-2	32/10	3/0	6	$\approx$	8.3e5	8	29/15	47/103	1m12s	90
lift4-3	32/10	3/0	6	$\approx$	9.7e5	4	4/2	9/19	1m02s	110
lift4-4	32/10	3/0	6	$\approx$	1.6e6	6	12/10	25/67	1m34s	150
lift5-1	38/13	3/0	6	?	3.4e6	4	5/2	10/26	8m35s	130
lift5-2	38/13	3/0	6	?	3.4e6	11	87/30	118/350	9m30s	140
lift5-3	38/13	3/0	6	?	5.0e6	6	10/6	19/44	9m57s	310
lift5-4	38/13	3/0	6	?	5.3e6	6	17/7	27/68	11m13s	310

Table 3: Verification of the lift system

All these properties have been proved, and they are quite involved. Such proofs require indeed to separate successive states of the controller and its environment: closing the door, going up or down floor by floor, opening the door, etc. Observe also that the Boolean automaton reduced by bisimulation grows very quickly whereas the control structures generated by our tool are much smaller. Moreover the automaton generation for $N = 4$ took 40 minutes! We were clearly unable to build it for $N = 5$. Here the addition of one floor does not induce a too large increase of the memory consumption, although the verification time grows more quickly. One can observe once again that properties that are intuitively easier are more easily proved by our tool (properties 1 vs. 2 and 3 vs. 4).

It should be noted that such verifications are not affordable by others techniques, as far as we know. The timed automata model could represent such a model (counters are never frozen here and could be implemented by clocks) but verification tools like KRONOS [DOTY96] or UPPAAL [LPY97]

[3]For $N = 6$, BDDs become very large, even with the use of reordering techniques, and we failed to build the initial control structure with its sequences.

cannot handle so many control states (when $N = 5$ for instance). We did not try techniques combining proof and model checking like [GS97, BLO98], which partitions the state space according to some guards appearing in the program, use a theorem prover to remove transitions, apply possibly compilation analysis techniques, and finally model-check the resulting graphs. They would probably fail on such an example:

- If only guards of *numerical* transition functions are considered, the obtained partition is too rough because the same counters are used many times in delay properties, and the formula guarding the incrementation of those counters have to be decomposed in order to distinguish their different uses.
- If guards of *all* transition functions are considered, the resulting control structure is at least as big as the Boolean control automaton reduced by bisimulation.

9 Conclusion

The initial goal of this work was to solve both the state explosion problem and the unprecision problem that are met in the verification of synchronous programs that have both complex Boolean control and non trivial numerical one.

This work is based on two main assumptions:

1. Being able to symbolically handle both Boolean and numerical properties is required, as well as the possibility to establish relations between the two kinds of properties. Indeed, the behavior of the two kinds of variables may influence each other and symbolic techniques become necessary when the number of variables increases.
2. Using approximations to keep the verification problem under a tractable complexity is required as well, whereas they are often used only to solve undecidability problems.

Abstract Interpretation is thus a natural theoretical framework. The first requirement is fulfilled on one hand by using as an abstract lattice the reduced product of the Boolean lattice and any abstract numerical lattice, and on the other hand by using the control structure to establish precise relations between the two kinds of properties. The second requirement is implemented by the use of the control structure to finely control the tradeoff

between precision and efficiency. An initial and small control structure is analyzed and refined progressively using refinement heuristics, according to the needs of verification. The principle of the refinement is to separate states according to their past or future behavior, in order to obtain a more precise knowledge about locations accessibility and to cut paths in the control structure. During this process the precision of the analysis is increased while the considered state space is considerably reduced, which allows acceptable performances to be achieved.

The programs we were able to verify show experimental evidence of the power of this technique. They justify our conviction that approximation is a very efficient way to increase the size of the systems for which it is possible to apply automatic verification methods, *if* it is possible to improve them in order to verify a wide spectrum of properties. The main observation is that properties generally involve only small parts of a program, which have to be analyzed with precision, whereas other parts can be roughly abstracted. This observation is especially true for big programs.

Further work. A source of complexity we did not handle in this work is the number of numerical variables: polyhedra operations have an exponential complexity w.r.t. that number. A possibility that began to be studied is to decompose or to approximate high dimension polyhedra with cartesian products of lower dimension polyhedra. Another direction would be to use other numerical lattices, less precise but also less costly, such as the interval lattice [CC76] or, more interesting, the lattice of zones [HNSY92, LLPY97] that is used in the verification of timed automata, and its generalization described in [Min00]. The last two lattices exhibit "only" cubic time and quadratic space complexity.

Another promising experiment is to apply the dynamic partitioning to the verification of linear hybrid systems, similar to those handled by the tool HyTech [HHWT95]. Such a work has been already started and requires only the addition of the time elapse operator, as described in [HPR97].

Perspectives. Dynamic partitioning has more general applications than the verification of synchronous programs. Actually, it allows to combine Boolean properties, at low cost, with any *other* abstract interpretation, in order to limit the state explosion problem that always happens in concurrent systems. For instance, a suitable abstract lattice to represent queue contents could enable the use of dynamic partitioning to verify automata communicating by unbounded FIFO channels [BG96, ABJ98].

On the other hand, the problem of the approximations induced by the least upper bound is very common in abstract interpretation. In this context, dynamic partitioning allows to improve automatically the precision of an analysis whose goal is the proof of an invariance property.

Acknowledgment

I thank Nicolas Halbwachs and Pascal Raymond for their valuable advices and for many helpful discussions and suggestions.

References

[ABJ98] P. Aziz Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In *Computer Aided Verification, CAV'98*, volume 1427 of *LNCS*, July 1998.

[ACD+92] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. Minimization of timed transition systems. In *Proceedings of the Third Conference on Concurrency Theory CONCUR '92*, Lecture Notes in Computer Science 630, pages 340–354. Springer-Verlag, 1992.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science B*, 138:3–34, January 1995.

[BBM97] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1), 1997.

[BdS92] A. Bouali and R. de Simone. Symbolic bisimulation minimisation. In *Computer Aided Verification, CAV'92*, volume 663 of *LNCS*, July 1992.

[BFH+92] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

[BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In

Computer Aided Verification, CAV'96, volume 1102 of *LNCS*, July 1996.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Computer Aided Verification, CAV'98*, volume 1427 of *LNCS*, July 1998.

[BLP+99] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In *Computer Aided Verification, CAV'99*, volume 1633 of *LNCS*, July 1999.

[BLS96] S. Bensalem, Y. Lakhnech, and H. Saïdi. Powerful techniques for the automatic generation of invariants. In *Computer Aided Verification, CAV'96*, volume 1102 of *LNCS*, July 1996.

[Bou92] F. Bourdoncle. Abstract interpretation by dynamic partitionning. *Journal of Functional Programming*, 2(4), 1992.

[Bou93] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *International Conference on Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, 1993.

[Bou99] L. Du Bousquet. Test fonctionnel statistique de systèmes spécifiés en LUSTRE. Thesis Université Joseph Fourier, Grenoble, 1999.

[CAM] The CAML language. http://caml.inria.fr/.

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, 1989.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd Int. Symp. on Programming*. Dunod, Paris, 1976.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or

approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages, POPL'77*, Los Angeles, January 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th ACM Symposium on Principles of Programming Languages, POPL'79*, San Antonio, January 1979.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 1992.

[CC92b] 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*, Leuven (Belgium), January 1992. LNCS 631, Springer Verlag.

[CGJ+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, CAV'2000*, volume 1855 of *LNCS*, July 2000.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages, POPL'78*, Tucson (Arizona), January 1978.

[Cou77] P. Cousot. Asynchronous iterative methods for solving a fixpoint system of monotone equations. Technical Report IMAG-RR-88, Université scientifique et médicale de Grenoble, 1977.

[CU98] M. A. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Computer Aided Verification, CAV'98*, volume 1427 of *LNCS*, July 1998.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, volume 1066 of *LNCS*, 1996.

[DWT95] D. Dill and H. Wong-Toï. Verification of real-time systems by successive over and under approximations. In *Seventh Conference on Computer-Aided Verification, CAV'95*, Liège (Belgium), July 1995. LNCS 939, Springer Verlag.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Conference on Computer Aided Verification CAV'97, Haifa*, volume 1254 of *LNCS*, June 1997.

[Hal98] N. Halbwachs. About synchronous programming and abstract interpretation. *Science of Computer Programming, Special Issue on SAS'94*, 31(1), May 1998.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HHWT95] T. Henzinger, P. Ho, and H. Wong-Toï. HyTech: The next generation. In *RTSS'95*, 1995.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[HNSY92] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model-checking for real-time systems. In *LICS'92*, June 1992.

[HPR97] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[Jea] B. Jeannet. The convex polyhedra library NEW POLKA. http://www-verimag.imag.fr/~bjeannet/newpolka-english.html.

[JGS93] N. D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 1993.

[JHR99] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symposium, SAS'99*, volume 1694 of *LNCS*, Venezia (Italy), September 1999.

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[LGLL91] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[LLPY97] Fredrik Larsson, Kim G. Larsen, Paul Pettersson, and Wang Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *Proc. of the 18*th *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 1997.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1(1/2), 1997.

[Mau96] C. Mauras. Calcul symbolique et automates interprétés. Technical Report 10, IRCyN, November 1996.

[Min00] A. Miné. Representation d'ensembles de contraintes de somme ou de difference de deux variables et application a l'analyse automatique de programmes. Master's thesis, Laboratoire d'Informatique de l'École Normale Supérieure de Paris, July 2000.

[MLAH99] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Computer Science Logic*, The IT University of Copenhagen, Denmark, September 1999.

[Ray91] P. Raymond. *Compilation efficace d'un langage déclaratif synchrone : Le générateur de code* LUSTRE-*V3*. Thesis, Institut National Polytechnique de Grenoble, November 1991.

[RHR91] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. In *ACM-SIGSOFT'91 Conference on Software for Critical Systems*, New Orleans, December 1991.

[Som] F. Somenzi. CUDD: Colorado University Decision Diagram Package. ftp://vlsi.colorado.edu/pub.

[STA98] R. F. Lutje Spelberg, W. J. Toetenel, and M. Ammerlaan. Partition refinement in real-time model checking. In *5th International Symposium on Formal Techniques in Real-Time and*

Fault-Tolerant Systems, pages 143–157. LNCS 1486, Springer Verlag, 1998.

[SUM96] H. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. In *8th International Conference on Computer Aided Verification, CAV'96*, volume 1102 of *LNCS*, July 1996.

[YL93] M. Yanakakis and D. Lee. An efficient algorithm for minimizing real-time transition systems. In *Fifth Conference on Computer-Aided Verification, CAV'93*, Elounda (Greece), July 1993. LNCS 697, Springer Verlag.

Recent BRICS Report Series Publications

RS-00-38 **Bertrand Jeannet.** ***Dynamic Partitioning in Linear Relation Analysis: Application to the Verification of Synchronous Programs*****. December 2000. 44 pp.**

RS-00-37 **Thomas S. Hune, Kim G. Larsen, and Paul Pettersson.** ***Guided Synthesis of Control Programs for a Batch Plant using*** **UPPAAL. December 2000. 29 pp. Appears in Lai, editor,** ***International Workshop in Distributed Systems Validation and Verification. Held in conjunction with 20th IEEE International Conference on Distributed Computing Systems (ICDCS '2000),*** **DSVV '00 Proceedings, 2000.**

RS-00-36 **Rasmus Pagh.** ***Dispersing Hash Functions*****. December 2000. 18 pp. Preliminary version appeared in Rolim, editor,** ***4th International Workshop on Randomization and Approximation Techniques in Computer Science,*** **RANDOM '00, Proceedings in Informatics, 2000, pages 53–67.**

RS-00-35 **Olivier Danvy and Lasse R. Nielsen.** ***CPS Transformation of Beta-Redexes*****. December 2000. 12 pp.**

RS-00-34 **Olivier Danvy and Morten Rhiger.** ***A Simple Take on Typed Abstract Syntax in Haskell-like Languages*****. December 2000. 25 pp. To appear in** ***Fifth International Symposium on Functional and Logic Programming,*** **FLOPS '01 Proceedings, LNCS, 2001.**

RS-00-33 **Olivier Danvy and Lasse R. Nielsen.** ***A Higher-Order Colon Translation*****. December 2000. 17 pp. To appear in** ***Fifth International Symposium on Functional and Logic Programming,*** **FLOPS '01 Proceedings, LNCS, 2001.**

RS-00-32 **John C. Reynolds.** ***The Meaning of Types — From Intrinsic to Extrinsic Semantics*****. December 2000. 35 pp.**

RS-00-31 **Bernd Grobauer and Julia L. Lawall.** ***Partial Evaluation of Pattern Matching in Strings, revisited*****. November 2000. 48 pp.**

RS-00-30 **Ivan B. Damgård and Maciej Koprowski.** ***Practical Threshold RSA Signatures Without a Trusted Dealer*****. November 2000. 14 pp.**