# Verification of Real-Time Systems using Linear Relation Analysis*

Nicolas Halbwachs, Yann-Erick Proy, Patrick Roumanoff
Verimag, Centre Equation
2, avenue de Vignate
F-38610 Gieres,France

**Abstract**

Linear Relation Analysis [CH78] is an abstract interpretation devoted to the automatic discovery of invariant linear inequalities among numerical variables of a program. In this paper, we apply such an analysis to the verification of quantitative time properties of two kinds of systems: synchronous programs and linear hybrid systems.

## 1 Introduction

Embedded systems are generally critical. So, they constitute a privileged application field for formal verification techniques. Among these techniques, automatic verification based on finite state abstractions has been very successfully developed and applied during the last decade.

However, embedded systems interact with their environment in real-time, and quantitative time properties are very common in this field. So, more recently, considerable research efforts have been devoted to the verification of real-time properties. As the addressed problems involve numerical aspects, the approaches based on finite state models no longer apply, in general. Such problems, being undecidable in general, can be approached along three ways:

> Many people propose methods and tools based on general or dedicated theorem proving techniques (e.g., [AL91, Rus94]). Of course this approach is quite general, but requires a significant, and often prohibitive, amount of human help.
>
> Other works have been devoted to the identification of decidable subproblems (e.g., [ACD93, HNSY92]) that can be automatically solved as finite state problems. Although interesting classes of decidable problems have been identified, this approach lacks generality, and is faced with the very high complexity of the verification procedures.
>
> A last track, which is considered in this paper, makes use of approximation.

Approximate verification methods either provide definite results ("yes, the property is satisfied", or "no, it is violated") or fail (i.e., answer "I don't know"). The question, of course, is about the precision of such techniques, i.e., the amount of "I don't know" answers you get. Notice that,

*This work has been partly supported by ESPRIT-BRA action "REACT", by ESPRIT-LTR project "SYRF", and by a grant from Schneider Electric.

from a practical point of view, getting an unconclusive answer is not worse than running out of time or memory with a "complete" but exponential decision procedure.

Approximate verification can benefit from program analysis techniques, developed a long time ago, to extract semantic properties to be used in compilers, either for static consistency checks or for code optimization. Most of these techniques have been unified in the framework of *abstract interpretation* [CC77]. In this paper, we propose to adapt a specific abstract interpretation, called *linear relation analysis*, to the verification of real-time systems.

Linear relation analysis [CH78] is a method to discover linear relationships invariantly satisfied by the numerical variables of a program. It was first designed to check the consistency of array accesses in sequential programs. The analysis associates with each control point of the program, a system of linear inequalities characterizing an upper approximation of the set of states in which the numerical variables can be when the program execution reaches the considered control point. An important consequence is that, when the linear system associated with a control point is unsatisfiable, it means that the control point is not reachable by any program execution. So, the analysis can be straightforwardly used to prove unreachability properties. Now, experience shows that the essential goal, for embedded systems, is generally to ensure some critical *safety properties*, expressing that something bad never happens, i.e., precisely, unreachability properties.

This paper recalls the fundamentals of linear relation analysis, and applies the method, with some extensions, to two classes of real-time systems:

> The first application concerns the verification of programs written in synchronous languages [IEE91, Hal93b], for properties involving delay counters [Hal93a]. Although these counters are bounded integer variables that could be taken into account by classical finite state methods, our analysis avoids the tremendous state explosion that would occur during such an exhaustive state exploration. The analysis can be applied to any numerical programs, but gives particularly good results in this field because of the simple behavior of counters.
>
> The second application concerns linear hybrid automata [ACHH93, AHH93, HPR94, ACH+95], a model that has been proposed to describe systems involving variables evolving continuously along the time. These systems are not finite state, and their verification is undecidable. We show that, thanks to slight extensions, linear relation analysis can cope with such continuous evolutions, and give quite precise results with very good performances.

## 2 Linear Relation Analysis

The linear relation analysis [CH78, Hal79] is an application of the general method of *abstract interpretation* proposed by P. & R. Cousot [CC77, CC92a]. It is an approximate analysis method which discovers invariant linear relations among numerical variables of a dynamic system. We informally recall its principles in this section.

### 2.1 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:

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

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 very 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 $\nabla$, 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 $\overline{y}$.

**Partitioned systems:** Assume the concrete domain $C$ is the powerset of some set $S$ of states, and that $S = K \times S'$, where $K$ is a finite set (typically, a set of *control points*). For each $k \in K$, let $C^{(k)} = \{k\} \times 2^{S'}$, and for each $x \in C$, let $x^{(k)} = x \cap C^{(k)}$. Clearly, for each $x \in C$, the set $\{x^{(k)} \mid k \in K\}$ is a finite partition of $x$. Now, any fixpoint equation $x = F(x)$ can be written as a system of equations:

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

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

This partitioning can be used to make the results more precise, as follows: The partition can obviously be reflected in the abstract domain, by setting $y^{(k)} = \alpha(x^{(k)})$, resulting in an abstract system of equations

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

We will say that $k$ 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 $\mathcal{R}_G$ be this dependence relation on $K$. Let $K_\nabla$ 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 $K_\nabla$:

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

The advantage is that the widening operator, which is the one which looses most information, is applied less frequently.

## 2.2 Convex Polyhedra

The linear relation analysis is used to deal with systems whose states include a numerical part. Let us define the set of states to be $S = N^n \times S'$, where $N$ is a numerical set (e.g., $\mathbb{N}$, $\mathbb{Z}$ or $\mathbb{Q}$). A state $s \in S$ is a pair $\langle X, s' \rangle$, where $X$ is a numerical vector and $s' \in S'$ is the non-numerical part of the state (it can contain a numerical part which is kept out of the analysis).

The concrete domain we consider is $C = 2^S$, and the abstract one is $\mathcal{P}(\mathbb{Q}^n)$, the set of convex polyhedra of $\mathbb{Q}^n$. Any subset $x$ of $S$ will be approximated by a closed convex polyhedron $\alpha(x) \in \mathcal{P}(\mathbb{Q}^n)$, such that

$$\langle X, s' \rangle \in x \implies X \in \alpha(x)$$

and any closed convex polyhedron $P \in \mathcal{P}(\mathbb{Q}^n)$ will represent the set of states

$$\gamma(P) = \{\langle X, s' \rangle \mid X \in P \cap N^n \ , \ s' \in S'\}$$

### 2.2.1 Representations of polyhedra

So, our abstract values are closed convex polyhedra. Let us recall that a closed convex polyhedron $P$ (a polyhedron, for short) has two representations (see Fig. 1):

it is the set of solutions of a *system of linear inequalities*

$$P = \{X \mid AX \geq B\}$$

where $A$ is a $m \times n$-matrix and $B$ is a $m$-vector.

it is the convex closure of a *system of generators*, i.e., two finite sets $V$ and $R$ (respectively for "vertices" and "rays") of $n$-vectors such that

$$P = \{ \sum_{v_i \in V} \lambda_i . v_i + \sum_{r_j \in R} \mu_j . r_j \mid \lambda_i \geq 0, \mu_j \geq 0, \sum_i \lambda_i = 1\}$$

These two representations are dual: If $P$ is a polyhedron, its *polar* is the polyhedron $P^* = \{x \mid \forall y \in P, x.y \leq 1\}$, where $x.y$ denotes the scalar product of $x$ and $y$. $P^*$ always contains the origin, and if $P$ contains the origin, $P^{**} = P$. If $(V, R)$ is a system of generators of $P$, then

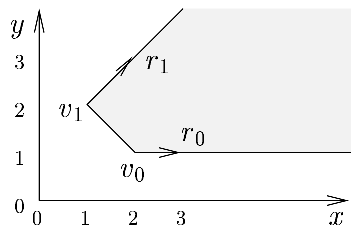
$$\bigwedge_{v \in V} v.x \leq 1 \;\wedge\; \bigwedge_{r \in R} r.x \leq 0$$

is a system of inequalities of $P^*$. Conversely, if $P$ contains the origin, and if its system of inequalities $(AX \leq B)$ is normalized into $A_0 X \leq \vec{0} \;\wedge\; A_1 X \leq \vec{1}$, then the set of rows of the matrices $A_0$ and $A_1$ respectively constitute the set of rays and of vertices of $P^*$.

There exist efficient algorithms [Che68, LeV92] for translating each representation into the other; these algorithms also minimize the representations. The principle of the translation is the following [Che68]: Assume $(AX \geq B)$ is the system of inequalities of a polyhedron $P$. A system of generators of $P$ can be computed iteratively as follows:

Start with $P_0 = \mathbb{Q}^n$, the whole space, a system of generators of which is $V_0 = \{\vec{0}\}$ (the origin) and $R_0 = \{\vec{i_1}, \ldots, \vec{i_n}, -\vec{i_1}, \ldots, -\vec{i_n}\}$, where $\{\vec{i_1}, \ldots, \vec{i_n}\}$ form a basis of $\mathbb{Q}^n$.

At step $k$, intersect $P_{k-1}$ with the $k$th inequality of $P$, say "$aX \geq b$": (1) any vertex $v \in V_{k-1}$ (resp. ray $r \in R_{k-1}$) such that $av \geq b$ (resp. $ar \geq 0$) belongs to $V_k$ (resp. $R_k$); (2) for any pair $(v, v')$ of vertices in $V_{k-1}$ such that $av > b$ and $av' < b$, their convex combination $\frac{b-av'}{av-av'}.v - \frac{b-av}{av-av'}.v'$ belongs to $V_k$; (3) for any pair $(v, r) \in V_{k-1} \times R_{k-1}$ such that either $av > b$ and $ar < 0$, or $av < b$ and $ar > 0$, their positive combination $v + \frac{b-av}{ar}.r$ belongs to $V_k$; (4) for any pair $(r, r')$ of rays in $R_{k-1}$ such that $ar > 0$ and $ar' < 0$, their positive combination $(ar').r - (ar).r'$ belongs to $R_k$.




$$P = \left\{ (x,y) \;\middle|\; \left( \begin{array}{rcl} y & \geq & 1 \\ x+y & \geq & 3 \\ -x+y & \leq & 1 \end{array} \right) \right\}$$

$$V = \left\{ v_0 \begin{pmatrix} 2 \\ 1 \end{pmatrix}, v_1 \begin{pmatrix} 1 \\ 2 \end{pmatrix} \right\} \quad R = \left\{ r_0 \begin{pmatrix} 1 \\ 0 \end{pmatrix}, r_1 \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\}$$

Figure 1: A convex polyhedron and its 2 representations

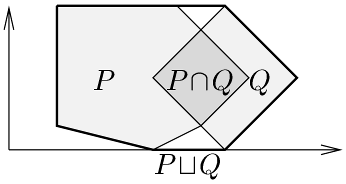



Figure 2: Intersection and convex hull

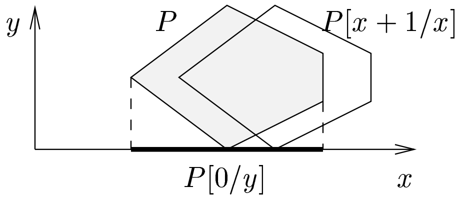



Figure 3: Linear transformations

The system of generators computed by the algorithm above is not minimal. It can be minimized as follows, using the system of inequalities of $P$: Let us say that a vertex $v$ (resp. a ray $r$) *saturates* an inequality $ax \geq b$ if $av = b$ (resp. $ar = 0$). Let $Sat(v)$ (resp., $Sat(r)$) denote the set of inequalities of $P$ saturated by the vertex $v$ (resp., the ray $r$). Then $v$ is *redundant* if there exists an other vertex $v'$ with $Sat(v) \subset Sat(v')$ (and similarly for rays). Two vertices $v$ and $v'$ are *mutually redundant* if $Sat(v) = Sat(v')$. A minimal system of generator can be extracted from $(V, R)$ by discarding all redundant vertices and rays, and keeping only one representative in each subset of mutually redundant vertices or rays. [LeV92] proposes an simpler and more efficient way to minimize a system of generators during its construction, based on the following remark: Let $n_1$ be the dimension of the least hyperplane containing $P$, and $n_2$ be the dimension of the greatest hyperplane contained in $P$. Then a point $v$ (resp., a vector $r$) is an actual vertex (resp., ray) of $P$ if and only if it saturates $n_1 - n_2$ (resp., $n_1 - n_2 - 1$) inequalities of $P$.

#### 2.2.2 Operations on polyhedra

We will use the following basic operations on polyhedra (see Fig. 2 and 3):

**Intersection:** The intersection of two convex polyhedra $P$ and $Q$ is a convex polyhedron whose system of linear inequalities is the conjunction of those of $P$ and $Q$.

**Convex hull:** The convex hull of two polyhedra $P$ and $Q$ (noted $P \sqcup Q$) is the least convex polyhedron containing both $P$ and $Q$. Its system of generators is the union of those of $P$ and $Q$. The convex hull is used as an upper approximation of union, since generally the union of two convex polyhedra is not convex.

**Linear transformation:** We will use linear transformations resulting of the substitution of linear expressions to variables. Let us define a *linear assignment* to be a pair $(A, B), A \in \mathbb{Q}^{n\times n}, B \in \mathbb{Q}^n$ defining the function $\lambda x.Ax + B$ from $\mathbb{Q}^n$ to $\mathbb{Q}^n$. The image of a polyhedron $P$ by a linear assignment $(A, B)$ is $\{Ax + B \mid x \in P\}$. If $(V, R)$ is a system of generators of $P$, then $(V' = \{Av + B \mid v \in V\}, R' = \{Ar \mid r \in R\})$ is a system of generators of the image of $P$ by $(A, B)$.

**Test for emptyness:** A polyhedron is empty if and only if it has no vertices.

**Test for inclusion and equality:** A polyhedron $P$, with system of generators $(V, R)$, is included in a polyhedron $Q$, defined by the system of inequalities $AX \geq B$, if and only if

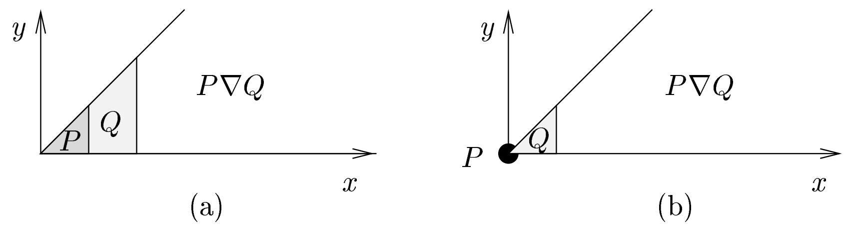



Figure 4: Widening operation

$\forall v \in V,\ Av \geq B$ and $\forall r \in R,\ Ar \geq 0$ . The equality of two polyhedra is decided by showing the double inclusion.

**Widening:** While the basic operations on abstract values are determined by the choice of the abstract domain, the design of a widening operator is based on heuristics. The following widening operator (hereafter called *standard widening*) was proposed in [Hal79]. Let $P$ and $Q$ be two polyhedra. Roughly speaking, the widening $P \nabla Q$ is obtained by removing from the system of $P$ all the inequalities that are not satisfied by $Q$. Fig. 4.a shows an example where $P = \{(x, y) \mid 0 \leq y \leq x \leq 1\}$, $Q = \{(x, y) \mid 0 \leq y \leq x \leq 2\}$ and $P \nabla Q = \{(x, y) \mid 0 \leq y \leq x\}$. The intuition is clear: whenever an inequality is translated or rotated, it can do so infinitely many times, so it is removed. This operator clearly satisfies the properties of a widening: the result contains both the operands, and since the system of inequalities of $P \nabla Q$ is a subset of the one of $P$, the widening cannot be infinitely iterated without convergence.

The actual operator is a bit more complicated: first, whenever $P$ is empty, $P \nabla Q = Q$. Moreover, if $P$ is included in a strict subspace of $\mathbb{Q}^n$, its minimal system of inequalities is not canonical. It should be first rewritten into an equivalent system maximizing the number of inequalities satisfied by $Q$, and thus kept in the result. For instance, consider:

$$P = \{(x, y) \mid x = 0 \land y = 0\} \quad , \quad Q = \{(x, y) \mid 0 \leq y \leq x \leq 1\}$$

The system of inequalities of $P$ can be first rewritten into $P = \{(x, y) \mid 0 \leq y \leq x \leq 0\}$ before performing the widening, which evaluates to $P \nabla Q = \{(x, y) \mid 0 \leq y \leq x\}$ (see Fig. 4.b) instead of $\{(x, y) \mid 0 \leq y \land 0 \leq x\}$, which would be obtained without rewriting. This optimization preserves the widening properties. It is performed using the following algorithm [Hal79]: All the inequalities of $P$ satisfied by $Q$ are kept in the result, together with all the inequalities of $Q$ that are mutually redundant with an inequality of $P$, i.e., saturated by the same vertices and rays of $P$. In the above example $y \leq x$ is an inequality of $Q$ that is mutually redundant with the inequality $0 \leq x$ of $P$.

# 3 First Application: Delays in Synchronous Programs

## 3.1 Synchronous Programs and their Verification

Synchronous programming has been proposed [IEE91, Hal93b] as a useful approach to describe real-time control kernels. A synchronous program is supposed to *instantly* and *deterministically* react to events coming from its environment. All synchronous languages share the same abstract notion of time: the notion of physical (chronometric) time is replaced by a simple order among events; the only relevant notions are the simultaneity and precedence of events. Physical time does not play any special role; it is handled as an external event, exactly as any other event coming from the program environment. This is called the *multiform notion of time*: Simply by counting events, one can express delays counted in "meters" as well as in "seconds".

The advantages of this approach have been pointed out elsewhere. Synchronous languages are simple and clean, they have been given simple and precise formal semantics, they allow especially elegant programming style. They can be compiled into a very efficient sequential code, using a specific compiling technique: The control structure of the object code is a finite automaton which is synthesized by an exhaustive simulation of a finite abstraction of the program.

Concerning program verification, it has been argued [BS91, HLR92] that the practical goal, for real-time programs, is generally to verify some simple logical safety properties: By a *safety* property, we mean, as usual, a property that expresses that something will never happen, and by a *simple logical* property, we mean a property that depends on logical dependences between events, rather than on complex relations between numerical values. For the verification of such properties also, the synchronous approach has some advantages: Since the parallel composition is synchronous, the desired properties of a program can be easily and modularly expressed by means of an *observer* [HLR93], i.e., another program which observes the behavior of the first one and decides whether it is correct. The verification then consists in checking that the parallel composition of the program and its observer never causes the observer to complain. This verification can often be performed by traversing the finite control automaton built by the compiler. Moreover, the automaton is generally much smaller than in the asynchronous case, where non-deterministic interleaving of processes often results in state explosion.

However, the claim that usual critical properties of a real-time system do not depend on numerical variable values can be disputed in one important aspect: they often depend on the values of the *delays* involved in program control. Now, the finite automata built by the compilers and considered in the verification do not reflect these delays: Delays are counted by means of integer variables, described in the interpretation associated with the automaton. For instance, the ESTEREL compiler doesn't know that the statement "await 5 SECOND" takes more time than "await 3 SECOND", and neither does any proposed verification tool. In that sense, one can argue that these tools have nothing to do with the verification of "real-time" properties.

In this section, we show how the Linear Relation Analysis can be combined with the usual verification methods to take numerical delays into account in the generation of automata. Let us take a small example, in ESTEREL[1]: We consider a car, about which we know that

- it stops within 4 seconds,
- if it doesn't stop before 10 meters, it bumps into an obstacle.

This simple behavior can be described as follows in ESTEREL:

```
trap END in
   await 4 SECOND; emit STOP; exit END;
||
   await 10 METER; emit BUMP; exit END;
end.
```

This small program is made of two parallel processes embedded into a "trap" block. The first

[1] All the examples will be given in ESTEREL, on the one hand, because it is probably the best-known synchronous language, and on the other hand, because it contains specific statements to deal with delays. However, the method described here can be applied to other languages.

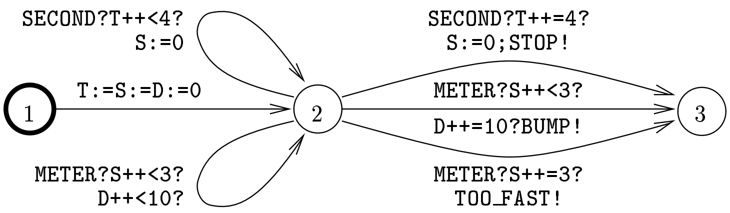



Figure 5: An interpreted automaton

process that stops waiting, instantaneously emits a signal and performs an "exit END", which terminates the whole block, thus killing the other process.

Now, assume we know also that the speed of the car is at most 2m/s. We can express this knowledge in the program, by signaling an exception whenever 3 meters are perceived within a second. The full program is as follows:

```
module car:
input METER, SECOND;
relation METER # SECOND;
output BUMP, STOP, TOO_FAST;
trap END in
      loop
         await 3 METER; emit TOO_FAST; exit END
      each SECOND
   ||
      do
         await 10 METER; emit BUMP; exit END
      ||
         await 4 SECOND; emit STOP; exit END
      upto TOO_FAST
end.
```

The "loop ... each SECOND" is started again each second. Thus, the exception TOO_FAST is only raised if three METER signals are received between two successive SECOND signals. In that case, the whole program terminates because of the "exit END" statement.

From this program, the ESTEREL compiler builds an interpreted automaton similar to that of Fig. 5 (where X++ denotes the value of X after incrementing it). It introduces 3 counters: T for counting 4 seconds (the time), S for counting 3 meters each second (the speed), and D for counting 10 meters (the distance). The structure of the automaton doesn't show that the emission of BUMP is impossible.

Now, this automaton is a sequential program, dealing with 3 bounded integer variables. An exhaustive simulation can be performed, which leads to a detailed, non interpreted, automaton with 49 states and 146 transitions, on which the property can be checked. This solution has an obvious drawback: The size of the detailed automaton clearly increases as the product of the delays. Counting a time delay in milliseconds rather than in seconds will tremendously increase the size of the automaton. So, our goal is to detect that some transitions of the interpreted automaton cannot occur because of delay counting, without considering the detailed automaton. For that, we apply our linear relation analysis.

```
Location 1 ....................... ........................ {P_1}
  T=0; D=0; S=0; goto Location 2 ..................... {P_1,1,2}

Location 2 ................................................ {P_2}
  if SECOND then S=0;
    if T++=4 then emit STOP; goto Location 3 ....... {P_1,2,3}
    end;
    goto Location 2................................... {P_1,2,2}
  end;
  if METER then
    if S++=3 then emit TOO_FAST; goto Location 3 .. {P_2,2,3}
    end;
    if D++=10 then emit BUMP; goto Location 3....... {P_3,2,3}
    end;
    goto Location 2................................... {P_2,2,2}
  end;
  goto Location 2..................................... {P_3,2,2}

Location 3 ................................................ {P_3}
   goto Location 3.................................... {P_1,3,3}
```

Figure 6: The code of the automaton, with associated polyhedra

## 3.2 Delay Analysis

### 3.2.1 Interpreted Automata

The linear relation analysis is applied to automata produced by synchronous languages compilers. Such an automaton is a finite set of states — which will be called *locations* to distinguish them from the global state of the automaton, involving also *valuations* of variables —, each of which being associated with a piece of sequential code. This code is made of three kinds of statements:

> *Assignments:* Those which do not assign counter variables will be ignored in the analysis. An assignment to a counter variable either increments it, or decrements it, or resets it to zero.
>
> *Tests* select statements to be performed according to some conditions. The only conditions that will be taken into account in the analysis are comparisons of counter variables with integers.
>
> *Branching statements* select the next location of the automaton. These statements terminate the code executed in a location.

Fig. 6 gives the code of the automaton shown in Fig. 5.

We will take advantage of this control structure to get a partitioned system. A state of the program is a triple $(\ell, X, Y)$, where $\ell$ is a location of the automaton, $X$ is a vector of counter values, and $Y$ is a vector of values of other variables (e.g., those giving the presence of external signals) which will be ignored. With each location $\ell$ of the automaton, we will associate a polyhedron $P_\ell$, which will be an approximation of the set

$$\{X \mid \exists Y,\ (\ell, X, Y) \text{ is a reachable state of the program}\}$$

Since we are interested in determining what transitions can occur and what locations can be reached, we will also associate a polyhedron with each branching statement, which will approximate the set of possible valuations of the counters when executing those statements: Let $P_{i,\ell',\ell}$ be the polyhedron associated with the $i$-th "goto location $\ell$" statement appearing in the code of location $\ell'$. Fig. 6 shows the polyhedra to be computed for our small example.

Clearly $P_\ell$ is the convex hull of all the $P_{i,\ell',\ell}$, and $P_{i,\ell',\ell}$ is computed from $P_{\ell'}$ according to the statements executed along the branch leading to the $i$-th "goto location $\ell$" appearing in the code of $\ell'$. The transformation of polyhedra resulting from assignments is straightforward, using linear transformations of polyhedra. For tests, three cases occur: Let $F_t$, $F_f$ be the transformations corresponding respectively to entering the "then" and "else" branches of a test, Then,

- if the condition is not a linear expression of the counters, it is ignored, and both $F_t$ and $F_f$ are the identity function $\lambda P.P$.
- if the condition is of the form "$X_i \leq k$", where $X_i$ is a counter and $k$ is an integer constant, then $F_t = \lambda P.P \cap \{X \mid X_i \leq k\}$ , $F_f = \lambda P.P \cap \{X \mid X_i \geq k+1\}$.
- if the condition is of the form "$X_i = k$", then

$$
\begin{aligned}
&F_t = \lambda P.P \cap \{X \mid X_i = k\}\\
&F_f = \lambda P.(P \cap \{X \mid X_i \geq k+1\}) \ \sqcup \ (P \cap \{X \mid X_i \leq k-1\})
\end{aligned}
$$

Notice that we take advantage of the fact that counters are integer variables, by setting $\neg(X_i \leq k) \equiv (X_i \geq k+1)$ and that the non-convex set $P \cap \{X \mid X_i \neq k\}$ is approximated by the convex hull of the two polyhedra $P \cap \{X \mid X_i \geq k+1\}$ and $P \cap \{X \mid X_i \leq k-1\}$.

Here are the definitions of the polyhedra corresponding to our example:

$$
\begin{aligned}
P_1 &= \mathit{true} \ \ \text{(initial location)}\\
P_2 &= P_{1,1,2} \sqcup P_{1,2,2} \sqcup P_{2,2,2} \sqcup P_{3,2,2}\\
P_3 &= P_{1,2,3} \sqcup P_{2,2,3} \sqcup P_{3,2,3} \sqcup P_{1,3,3}\\
P_{1,1,2} &= P_1[0/T][0/S][0/D]\\
P_{1,2,3} &= P_2[0/S][T{+}1/T] \cap \{(T,S,D) \mid T = 4\}\\
P_{1,2,2} &= (P_2[0/S][T{+}1/T] \cap \{(T,S,D) \mid T \leq 3\})\\
&\qquad \sqcup (P_2[0/S][T{+}1/T] \cap \{(T,S,D) \mid T \geq 5\})\\
P_{2,2,3} &= P_2[S{+}1/S] \cap \{(T,S,D) \mid S = 3\}\\
P_{3,2,3} &= Q[D{+}1/D] \cap \{(T,S,D) \mid D = 10\}\\
P_{2,2,2} &= (Q[D{+}1/D] \cap \{(T,S,D) \mid D \leq 9\}) \sqcup (Q[D{+}1/D] \cap \{(T,S,D) \mid D \geq 11\})\\
\text{with } Q &= (P_2[S{+}1/S] \cap \{(T,S,D) \mid S \leq 2\}) \sqcup (P_2[S{+}1/S] \cap \{(T,S,D) \mid S \geq 4\})\\
P_{3,2,2} &= P_2\\
P_{1,3,3} &= P_3
\end{aligned}
$$

### 3.2.2 Widening strategies

The points where the widening is performed are selected among location entry points. Although the ESTEREL compiler generates a dummy transition looping on each location, we do not have to perform a widening in each of these loops where no action is performed. So, we consider only the transitions containing actions on counters, and we select a location in each loop of such transitions. In our example, we select location 2, which belongs to any loop, and change the equation of $P_2$:

$$P_2 \ = \ P_2 \ \nabla \ (P_{1,1,2} \sqcup P_{1,2,2} \sqcup P_{2,2,2} \sqcup P_{3,2,2})$$

Moreover, our experimentations show that both the precision and the performances of the analysis are improved by the following modifications:

**Widening "up to":** One can choose a fixed set of linear inequalities, say $M$, and define a new "widening up to $M$" operator $\nabla_M$ as follows: $P \nabla_M Q$ is the intersection of the standard widening $P \nabla Q$ with all the inequalities in $M$ that are satisfied by both $P$ and $Q$. For instance, if a counter $x$ is declared to be of subrange type 0..10, if the domain of $x$ is first $\{x = 0\}$ and then $\{0 \leq x \leq 1\}$, it is reasonable to widen this domain to $\{0 \leq x \leq 10\}$ instead of $\{0 \leq x\}$. It is a way of guessing an invariant — a guess that can be found false at a next step. This heuristic changes neither the property of the widening nor the correctness of the result. In many cases, not only it avoids the necessity of the decreasing sequence — since the increasing sequence reaches a fixpoint — but also it provides a more precise result. In the case of our counters, a set of inequalities $M$ is associated with each widening location. This set is selected to be all the linear relations which make the control remain in the location. The intuition behind this choice is the following: Assume $\ell$ is a location whose only outgoing transition is guarded by the condition "x++=10" and that $\ell$ is entered with x=0. Then, since the control remains in $\ell$ (possibly incrementing or decrementing x) unless x becomes equal to 10, x is likely to remain smaller than 9 as long as the control is in $\ell$. In our example, the location 2 is left when either T++=4 or S++=3 or D++=10. The set of inequalities limiting the widening is

$$\{T \leq 3 \ , \ T \geq 4 \ , \ S \leq 2 \ , \ S \geq 3 \ , \ D \leq 9 \ , \ D \geq 10\}$$

**Non-regular behavior:** Any widening operator is chosen under the assumption that a program behaves regularly: When we get $\{x = y = 0\}$ at the first step, and $\{0 \leq y \leq x \leq 1\}$ at the second step, this assumption of regularity consists of guessing that we are likely to get $\{0 \leq y \leq x \leq 2\}$ at the third step, and so on; this is why the standard widening extrapolates the limit to $\{0 \leq y \leq x\}$. Now, the assumption of regularity is obviously abusive in one case: when a path in the loop becomes possible at step $n$, the effect of this path is obviously out of the scope of the extrapolation before step $n$ (since the actions performed on this path have never been taken into account). So, if the polyhedron associated with a widening point depends on some polyhedra which become non-empty at step $n$, the extrapolation performed before can be questioned. In such a case, the extrapolation will be performed from the first non-empty solution: In our example, if one of $P^{(n)}_{1,1,2}, P^{(n)}_{1,2,2}, P^{(n)}_{2,2,2}, P^{(n)}_{3,2,2}$ is not empty whereas it was at step $n-1$, we will take

$$P_2^{(n+1)} = P_2^{(1)} \ \nabla \ (P^{(n)}_{1,1,2} \sqcup P^{(n)}_{1,2,2} \sqcup P^{(n)}_{2,2,2} \sqcup P^{(n)}_{3,2,2})$$

because $P_2^{(1)}$ is the first non-empty version of $P_2$.

## 3.3 Examples

### 3.3.1 The "car" example

Let us detail the analysis of the very simple program we considered so far. The system of equations has been given in §3.2.1. Let us recall (cf. §3.2.2) that the only widening location is Location 2, and that the widening is performed up to the following inequalities:

$$M = \{T \leq 3\ ,\ T \geq 4\ ,\ S \leq 2\ ,\ S \geq 3\ ,\ D \leq 9\ ,\ D \geq 10\}$$

The successive computation steps are the following:

*Step 0:* Initially, all the polyhedra are empty.

*Step 1:* The first iteration in the loop provides:

$$\begin{array}{rcl}
P_2^{(1)} & = & P_{1,1,2}^{(1)} = \{T = S = D = 0\} \\
P_{1,2,2}^{(1)} & = & P_2^{(1)}[T+1/T][0/S] \cap \{T \leq 3\}) \sqcup (P_2^{(1)}[T+1/T][0/S] \cap \{T \geq 5\}) \\
& = & \{T = 1, S = D = 0\} \\
Q & = & (P_2^{(1)}[S+1/S] \cap \{|\ S \leq 2\}) \sqcup (P_2^{(1)}[S+1/S] \cap \{S \geq 4\}) \\
& = & \{S = 1, T = D = 0\} \\
\\
P_{2,2,2}^{(1)} & = & (Q[D+1/D] \cap \{D \leq 9\}) \sqcup (Q[D+1/D] \cap \{D \geq 11\}) \\
& = & \{S = D = 1, T = 0\}
\end{array}$$

and so $P_{1,1,2}^{(1)} \sqcup P_{1,2,2}^{(1)} \sqcup P_{2,2,2}^{(1)} = \{T \geq 0, S = D \geq 0, S + T \leq 1\}$

*Step 2:* The widening is applied, and we get:

$$\begin{array}{rcl}
P_2^{(2)} & = & \{T = S = D = 0\} \nabla_M \{T \geq 0, S = D \geq 0, S + T \leq 1\} \\
& = & \{0 \leq S = D \leq 2, 0 \leq T \leq 3\} \\
P_{1,2,2}^{(2)} & = & P_2^{(2)}[T+1/T][0/S] \cap \{T \leq 3\}) \sqcup (P_2^{(2)}[T+1/T][0/S] \cap \{T \geq 5\}) \\
& = & \{S = 0, 0 \leq D \leq 2, 1 \leq T \leq 3\} \\
Q & = & (P_2^{(2)}[S+1/S] \cap \{|\ S \leq 2\}) \sqcup (P_2^{(2)}[S+1/S] \cap \{S \geq 4\}) \\
& = & \{1 \leq S = D + 1 \leq 2, 0 \leq T \leq 3\} \\
P_{2,2,2}^{(2)} & = & (Q[D+1/D] \cap \{D \leq 9\}) \sqcup (Q[D+1/D] \cap \{D \geq 11\}) \\
& = & \{1 \leq S = D \leq 2, 0 \leq T \leq 3\}
\end{array}$$

and $P_{1,1,2}^{(2)} \sqcup P_{1,2,2}^{(2)} \sqcup P_{2,2,2}^{(2)} = \{0 \leq S \leq D \leq 2T + S, D \leq 2, T \leq 3\}$

*Step 3:*

$$\begin{array}{rcl}
P_2^{(3)} & = & \{0 \leq S \leq D \leq 2T + S, T \leq 3, D \leq 2\} \\
P_{1,2,2}^{(3)} & = & \{S = 0, 0 \leq D \leq 2, 1 \leq T \leq 3\} \\
P_{2,2,2}^{(3)} & = & \{1 \leq S \leq D \leq 2T + S, T \leq 3, D \leq 3, S \leq 2\}
\end{array}$$

and $P_{1,1,2}^{(3)} \sqcup P_{1,2,2}^{(3)} \sqcup P_{2,2,2}^{(3)} = \{0 \leq S \leq D \leq 2T + S, D \leq S + 2, T \leq 3, D \leq 3, S \leq 2\}$

```
Location 1
  T=0; D=0; S=0;
  goto Location 2.......................{T = S = D = 0}

Location 2............ {0 ≤ S ≤ D ≤ 2T + S, T ≤ 3, S ≤ 2}
  if SECOND then S=0;
    if T++=4 then
      emit STOP;
      goto Location 3 ......... {S = 0, 0 ≤ D ≤ 8, T = 4}
    end;
    goto Location 2....... {S = 0, 0 ≤ D ≤ 2T, 1 ≤ T ≤ 3}
  end;
  if METER then
    if S++=3 then
      emit TOO_FAST;
      goto Location 3 ....{S = 3, 2 ≤ D ≤ 2T + 2, T ≤ 3}
    end;
    if D++=10 then
      emit BUMP;
      goto Location 3...............................{∅}
    end;
    goto Location 2...{1 ≤ S ≤ D ≤ 2T + S, T ≤ 3, S ≤ 2}
  end;

Location 3
{3T + S ≤ 12 ≤ 3T + 4S, 2S ≤ 3D ≤ 6T + 2S, S ≤ 3}
    goto Location 3
```

Figure 7: Results of the analysis of the "car" example

*Step 4:*

$$\begin{aligned}
P_2^{(4)} &= \{0 \leq S \leq D \leq 2T + S, T \leq 3, S \leq 2\} \\
P_{1,2,2}^{(4)} &= \{S = 0, 0 \leq D \leq 2T, 1 \leq T \leq 3\} \\
P_{2,2,2}^{(4)} &= \{1 \leq S \leq D \leq 2T + S, T \leq 3, S \leq 2\}
\end{aligned}$$

and since $P_{1,1,2}^{(4)} \sqcup P_{1,2,2}^{(4)} \sqcup P_{2,2,2}^{(4)} \sqcup P_{3,2,2}^{(4)} \sqcup P_{1,3,2}^{(4)} = P_2^{(4)}$ the sequence converges on a fixpoint.

The polyhedra which do not belong to the loop evaluate to:

$$\begin{aligned}
P_{1,2,3} &= \{S = 0, 0 \leq D \leq 8, T = 4\} \\
P_{2,2,3} &= \{S = 3, 2 \leq D \leq 2T + 2, T \leq 3\} \\
P_{3,2,3} &= \emptyset
\end{aligned}$$

The final results are shown on Fig. 7. From the fact that $P_{3,2,3}$ is empty, we conclude that the corresponding transition cannot occur, and that the BUMP signal is never emitted in the ESTEREL program. This analysis needs 0.1 sec of CPU time, on a SUN-Sparc 10 workstation.

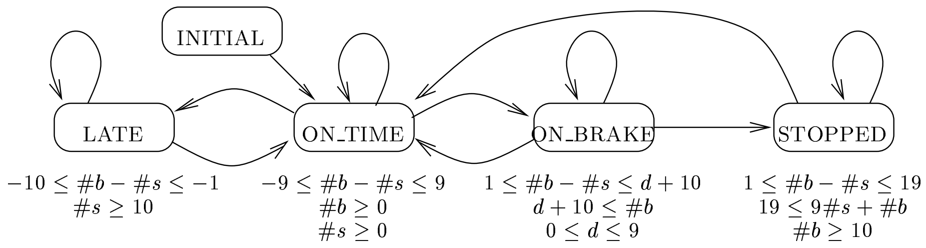



Figure 8: Result of the subway example, with one train

### 3.3.2 A subway speed regulation system

Our next 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.
>
> 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.

The results of the analysis of a simulation program for one train are shown in Fig. 8. Notice that the absolute difference $|\#b - \#s|$ is shown to be bounded. Notice also that the bound 19 has been discovered, although it does not appear in any condition of the program. For two trains, the analysis shows that the difference $\#b_1 - \#b_2$ of the number of beacons encountered by each train remains in the interval $[-29, +29]$. So, if they are initially separated by more than 29 beacons, no collision can occur. The analysis for one train needs 0.4 sec of CPU time, while the one for two trains takes 8.2 sec.

## 4 Second Application: Linear Hybrid Systems

Our second application field concerns hybrid automata. These automata have been proposed [MMP91, ACHH93] to model systems involving both discrete and continuous variables. A hybrid automaton is a finite automaton associated with a finite set of variables continuously varying in dense time. Each transition of the automaton can be guarded by a condition on these variables, and can perform an action modifying their values (discrete change). In each location[2] of the automaton, the variables continuously vary, according to a system of differential equations associated with the location.

[2] As in the preceding section, since we deal with interpreted automata, the control states of these automata will be called *locations*, to distinguish them from the global *state* of the system, made of a location and a *valuation* of the variables.

In this section, we consider a class of *linear hybrid automata* (which has been identified elsewhere [KPSY93, ACHH93, AHH93]), where guards are (general) linear conditions, actions are assignments of linear expressions, and in each location of which variable derivatives are (symbolic) constants subject to linear inequalities. This class is general enough to handle many practical problems, and the linear restrictions allow easy symbolic representation. The linear relation analysis is particularly suitable for the analysis of linear hybrid systems, on one hand because it copes with continuous variables, and on the other hand, because the behavior of these variables is naturally linear.

## 4.1 Linear Hybrid Automata

Before defining linear hybrid automata, we make precise some notions: Let $Var = \{x_1, \dots, x_n\}$ be the set of variables. A *linear constraint* is given by a triple $(a, \rho, b)$ where $a \in \mathbb{Z}^n$, $b \in \mathbb{Z}$ and $\rho \in \mathcal{R} = \{<, \leq, =, \geq, >\}$. It characterizes the subset of $\mathbb{Q}^n$ made of all the vectors $X$ satisfying $aX \; \rho \; b$. Notice that, for obvious computational reasons, we restrict ourselves to integer coefficients and rational values. A conjunction of $m$ linear constraints, given by a triple $(A, R, B) \in (\mathbb{Z}^n)^m \times \mathcal{R}^m \times \mathbb{Z}^m$, will be simply called a *linear system*. A *linear assignment* is given by a pair $(A, B), A \in \mathbb{Z}^{n \times n}, B \in \mathbb{Z}^n$ and defines the function $\lambda X.AX + B$.

**Definition:** A *linear hybrid automaton* $H = \{Loc, Var, Init, Trans, D, Inv\}$ consists of 6 components:

A finite set *Loc* of *locations*

A finite set *Var* of *variables*, supposed to be functions of dense time . Let $n = Card(Var)$. A *valuation* $X$ is a vector of $\mathbb{Q}^n$, giving the value of each variable in *Var*. A *state* is a pair $(\ell, X)$ where $\ell$ is a location and $X$ is a valuation.

A labeling function *Init* which assigns to each location $\ell$ a linear system $Init(\ell)$, specifying the set of *initial valuations*, if the automaton is started in location $\ell$. If $Init(\ell)$ is unsatisfiable, the automaton cannot be started in $\ell$.

A finite set *Trans* of *transitions*. Each transition $\tau = (\ell, \gamma, \alpha, \ell')$ consists of a *source location* $\ell \in Loc$, a *target location* $\ell' \in Loc$, a *guard* $\gamma$, which is a linear system over *Var*, and an *action* $\alpha$, which is a linear assignment to *Var*. A transition $\tau = (\ell, \gamma, \alpha, \ell')$ is *enabled* in a state $(\ell, X)$ if and only if $\gamma(X)$ holds. The state $(\ell', \alpha(X))$ is then the *transition successor* of $(\ell, X)$ by $\tau$.

A labeling function $D$ which assigns to each location $\ell$ a linear system $D(\ell)$ constraining variable's derivatives: If the automaton reaches a location $\ell$ with valuation $X$, after staying in $\ell$ for a delay $\delta$, the valuation will be $X + \delta\dot{X}$ where the vector of derivatives $\dot{X}$ satisfies the system $D(\ell)$.

A labeling function *Inv* which assigns to each location $\ell$ a linear system $Inv(\ell)$ constraining variables: The automaton can only stay in location $\ell$ as long as the current valuation satisfies $Inv(\ell)$.

At any instant, the state of the automaton is given by a control location and a valuation of the variables. The state can change in two ways:

an enabled *discrete* transition can instantaneously change both the control location and the current variable valuation;

a *time delay* can change only the valuation according to a vector of derivatives satisfying the constraints on derivatives associated with the current location; such a delay can only take place as long as the valuation satisfies the invariant associated with the location.

More formally, a *run* of the automaton is a finite or infinite sequence

$$s_0 \rightarrow^{t_0}_{\dot{X}_0} s_1 \rightarrow^{t_1}_{\dot{X}_1} s_2 \rightarrow^{t_2}_{\dot{X}_2} \dots$$

of states $s_i = (\ell_i, X_i)$, nonnegative reals $t_i$ and vectors of derivatives $\dot{X}_i$ ($\dot{X}_i$ satisfying $D(\ell_i)$) such that $X_0$ satisfies $Init(\ell_0)$ and for all $i \geq 0$,

for all $t \in [0, t_i[$, $X_i + \dot{X}_i t$ satisfies $Inv(\ell_i)$

the state $s_{i+1}$ is a transition successor of the state $s'_i = (\ell_i, X_i + \dot{X}_i t_i)$.

An infinite run diverges if the infinite sum $\sum_{i \geq 0} t_i$ diverges. With such a divergent run, we can associate a *behavior*, which is a total function $\beta$ from time to valuations defined as follows:

$$\beta(t) = X_{i(t)} + \dot{X}_{i(t)}(t - \sum_{j<i(t)} t_j), \quad \text{where } i(t) = \min\{\, k \mid \sum_{j=0}^{k} t_j > t \,\}$$

A state $(\ell, X)$ is *reachable* iff there exists a divergent run and an instant $t$ such that

$$\ell = \ell_{i(t)} \text{ and } X = X_{i(t)} + \dot{X}_{i(t)}(t - \sum_{j<i(t)} t_j)$$

Let *Reach* denote the set of reachable states.

We will represent a hybrid automaton by means of a directed graph, whose nodes represent the locations and edges represent transitions. Let us illustrate the use of the model by means of some classical examples.

## 4.2 Examples

### 4.2.1 A water-level monitor:

The water level in a tank is controlled through a monitor, which continuously senses the water level and turns a pump on and off. The water level changes as a piecewise-linear function over time: when the pump is off, the water level, denoted by the variable $w$, falls by 2 inches per second; when the pump is on, the water level rises by 1 inch per second. Suppose that initially the water level is 1 inch and the pump is turned on. We wish to keep the water between 1 and 12 inches. But from the time that the monitor signals to change the status of the pump to the time that the change becomes effective, there is a delay of 2 seconds. Thus the monitor must signal to turn the pump on before the water level falls to 1 inch, and it must signal to turn the pump off before the water level reaches 12 inches.

The hybrid automaton of Figure 9 describes a water level monitor that signals whenever the water level passes 5 and 10 inches, respectively. The automaton has four locations: in locations $\ell_0$ and $\ell_1$, the pump is turned on; in locations $\ell_2$ and $\ell_3$, the pump is off. The clock $x$ is used to specify the delays: whenever the automaton control is in location $\ell_1$ or $\ell_3$, the signal to switch

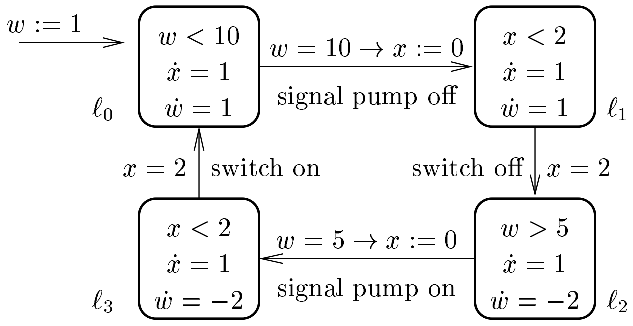



Figure 9: Water-level monitor

the pump off or on, respectively, was sent $x$ seconds ago. On each transition, we give the guard and the action (if any). In each location, we give the label, the invariant, and the constraints on derivatives. For instance, the invariant associated with location $\ell_3$ is $x < 2$, and the derivatives of $x$ and $w$ are 1 and $-2$, respectively.

#### 4.2.2 Fischer mutual exclusion protocol:

This example describes a parameterized multi-rate timed system. It is a timing-based algorithm that implements mutual exclusion for a distributed system with skewed clocks. Consider an asynchronous shared-memory system that consists of two processes $P_1$ and $P_2$ with atomic read and write operations. Each process has a critical section and at each instant, at most one of the two processes is allowed to be in its critical section. Mutual exclusion is ensured by a version of Fischer's protocol [Lam87], which we describe first in pseudocode. The code executed by process $P_i (i = 1, 2)$ is shown beside.

```
repeat
    repeat
        await k = 0
        k := i
        delay b
    until k = i
    Critical section
    k := 0
forever
```

The two processes $P_1$ and $P_2$ share a variable $k$, and process $P_i$ is allowed to be in its critical section iff $k = i$. Each process has a private clock. The statement "**delay** $b$" delays a process for at least $b$ time units as measured by the process's local clock. Furthermore, each process takes at most $a$ time units, as measured by the process's clock, for a single write access to the shared memory (i.e., for the assignment $k := i$). The values of $a$ and $b$ are the only information we have about the timing behavior of processes. Clearly, the protocol ensures mutual exclusion only for certain values of $a$ and $b$. If both private processor clocks proceed at precisely the same rate, then mutual exclusion is guaranteed iff $a < b$.

To make the example more interesting, we assume that the private clocks of the processes $P_1$ and $P_2$ proceed at different rates, namely, the rate of the local clock of $P_2$ is between 0.9 and 1.1 times the rate of the clock of $P_1$.

The resulting system could be modeled by the product of two hybrid automata, each of which modeling one process. Instead, for clarity in the further treatment of this example, we give an

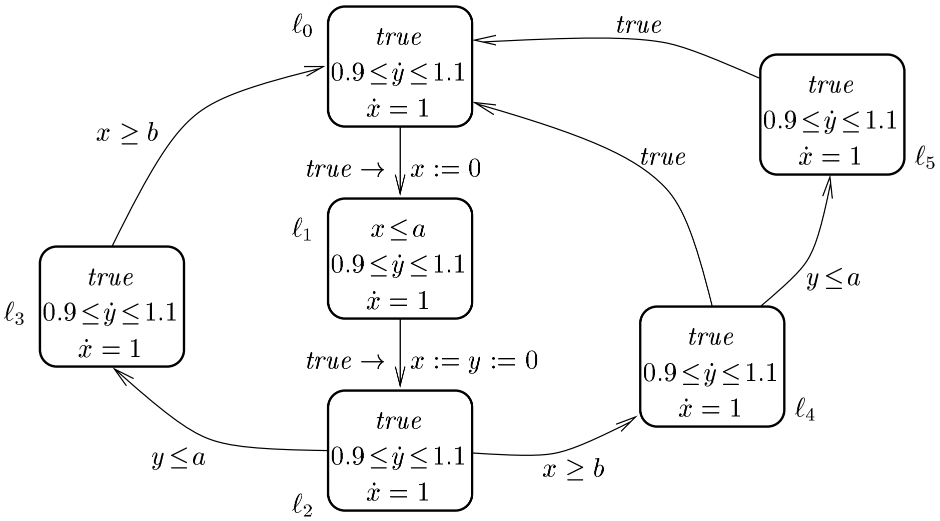



Figure 10: Fischer protocol (simplified)

abstract view of the system, where the behavior of $P_2$ is abstracted. Fig. 10 gives the behavior of $P_1$ with only the relevant interactions with $P_2$. Variables $x$ and $y$ are used to count delays, with respect to $P_1$'s and $P_2$'s local clock, respectively. In location $\ell_0$, $P_1$ is idle, in $\ell_1$, it has read $k = 0$. On the transition from $\ell_1$ to $\ell_2$, $P_1$ is supposed to set $k$ to 1, so it is the last time $P_2$ can read $k = 0$. In $\ell_2$, $P_1$ waits for $b$; two transitions may occur:

either the delay $b$ expires, and $P_1$ enters the critical section ($\ell_4$),

or $P_2$ sets $k$ to 2 ($\ell_3$) thus forbidding $P_1$ to enter the critical section.

In $\ell_4$, $P_1$ is in the critical section; if $P_2$ may set $k$ to 2, it may also enter the critical section, and the mutual exclusion is violated (location $\ell_5$).

#### 4.2.3 A scheduler:

Our last example is a task scheduler. We consider two classes of tasks, activated by interrupts. Interrupt $I_1$ (resp. $I_2$) occurs at most once each 10 (resp. 20) time units and activates a task of the first (resp. second) class, which takes 4 (resp. 8) time units. Tasks of the second class have priority, and can preempt other tasks. We want to show that a task of the second class never waits.

We use two timers, $c_i (i = 1, 2)$, to count the delay elapsed since the last interrupt $I_i$. The assumptions about interrupt frequencies can be expressed by the automaton of Fig. 11.(a). Concerning tasks, (see Fig. 11.(b)), we use two other timers, $x_i (i = 1, 2)$, to count the execution time of tasks, and two counters $k_i (i = 1, 2)$, to count the number of pending tasks in each class (these counters are discrete variables, their derivative is supposed to be 0 in any location).

A typical behavior of this automaton is to start in location $\ell_0$ (idle), then receiving an interrupt $I_1$ and activating a task 1 (in location $\ell_1$) for 4 time units — counted by the timer $x_1$. If, during this execution, an interrupt $I_2$ occurs, the task 1 is suspended, and the scheduler executes a task 2 (in location $\ell_2$) for 8 time units — counted by $x_2$. Notice that the timer $x_1$

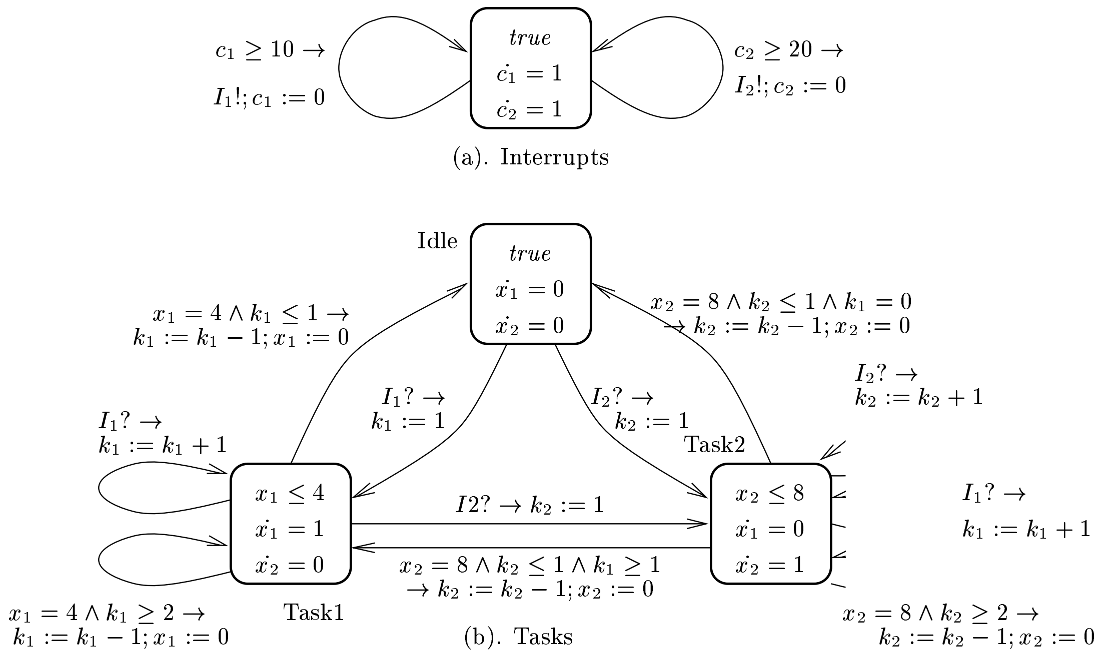



Figure 11: The scheduler

is only frozen in $\ell_2$, so, on termination of all the pending tasks 2, the suspended task 1 can be completed. The occurrence of an interrupt which does not have priority upon the active task only results in incrementing the corresponding counter.

## 4.3 Linear relation analysis and linear hybrid automata

### 4.3.1 Forward collecting semantics of linear hybrid automata

Let us recall that *Reach* denotes the set of reachable states. For any location $\ell$, we note $Reach_\ell$ the set of reachable valuations at location $\ell$:

$$Reach_\ell \;=\; \{X \mid (\ell, X) \in Reach\}$$

We will characterize the tuple $(Reach_\ell)_{\ell \in Loc}$ by means of a system of fixpoint equations. This system will be constructed in a "forward" way, in the sense that, for each location $\ell$, $Reach_\ell$ will be defined as a function of the sets $Reach_{\ell'}$, where $\ell'$ runs over the source location of transitions incoming to $\ell$:

$$Reach_\ell = F_\ell\left(\{Reach_{\ell'} \mid (\ell', \gamma, \alpha, \ell) \in Trans\}\right)$$

We first introduce some operations on sets of valuations. Let $\alpha = (A, B)$ be a linear assignment, and $S$ be a set of valuations. We note $\alpha(S)$ the image of $S$ by $\alpha$:

$$\alpha(S) \;=\; \{AX + B \mid X \in S\}$$

Let $D$ be a linear system (supposed to be a domain of derivatives), and $S$ be a set of valuations. We note $S \nearrow D$ the set of valuations that can be obtained by letting the variables continuously evolve, according to a constant derivative belonging to $D$, and starting from a valuation belonging to $S$:

$$S \nearrow D \;=\; \{X + td \mid X \in S,\; d \in D\,, t \in \mathbb{Q}^{\geq 0}\}$$

This operator will be called the *time elapse operator*.
Now, we are able to define the set $Reach_\ell$:

$$Reach_\ell \;=\; \left( \left( Init(\ell) \cup \bigcup_{(\ell',\gamma,\alpha,\ell) \in Trans} \alpha(Reach_{\ell'} \cap \gamma) \cap Inv(\ell) \right) \nearrow D(\ell) \right) \cap Inv(\ell)$$

This equation expresses that a reachable valuation in location $\ell$ satisfies the invariant $Inv(\ell)$, and is obtained by letting the time elapse from either an initial valuation or from an incoming valuation satisfying the invariant (notice that, since the invariant defines a convex domain, and since the continuous time-elapsing transformation is linear, any time-elapsing behavior starting from a valuation satisfying the invariant and leading to a valuation satisfying the invariant, continuously satisfies the invariant). An incoming valuation is obtained from a valuation associated with the source location of an incoming transition, satisfying the guard of the transition, as the result of the action of the transition.

We will use this characterization of $Reach_\ell$ as a basis to apply the linear relation analysis. First, we need some new features:

The time-elapse operator on polyhedra must be realized;

Since we work on continuous domains, we can no longer limit ourselves to closed polyhedra: strict inequalities must be taken into account.

### 4.3.2 Extension of the analysis

**Time elapse operator:** Let us recall that, if $D$ be a polyhedron representing a domain of derivatives, and $P$ is a polyhedron, then

$$P \nearrow D \;=\; \{X + td \mid X \in P,\; d \in D\,, t \in \mathbb{Q}^{\geq 0}\}$$

This operator is easily implemented as follows: Let $(V, R)$, $(V', R')$ be the respective systems of generators of $P$ and $D$. Then $(V\;,\; R \cup V' \cup R')$ is a system of generators of $P \nearrow D$.

**Strict inequalities:** The case of strict inequalities was not considered in previous applications of the method, since only discrete integer variables were considered. In fact, strict inequalities can easily be handled, by adding an auxiliary variable, say $\varepsilon$, with $0 \leq \varepsilon \leq 1$, replacing any strict inequality $aX > b$ by $aX - \varepsilon \geq b$, and checking that $\varepsilon$ is not null when checking for polyhedron emptyness and inclusion. More precisely, if $AX \geq B \;\wedge\; A'X > B'$ is the system of inequalities of a polyhedron $P$ of $\mathbb{Q}^n$, let $\widehat{P}$ be the polyhedron of $\mathbb{Q}^{n+1}$ defined by the system

$$AX \geq B \;\wedge A'X - \varepsilon \geq B' \;\wedge\; 0 \leq \varepsilon \leq 1$$

We note $\widehat{X}$ the extended vector of variables, and $\widehat{A}\widehat{X} \geq \widehat{B}$ the system of constraints of $\widehat{P}$. Let $\widehat{V}$ be the set of vertices of $\widehat{P}$, and $\widehat{V}^{>0}$, the subset of those vertices whose $\varepsilon$-component is strictly positive. Then, *Test for emptyness:* A polyhedron $P$ is empty if and only if $\widehat{V}^{>0}$ is empty.

*Test for inclusion:* We note $\widehat{P}$ the closure of $P$, i.e., the polyhedron defined by the same system of constraints as $P$ where all the inequalities are considered to be loose. If $v \in \widehat{V}^{>0}$, we note $v\!\downarrow$ its projection onto $\mathbb{Q}^n$ according to $\varepsilon$ (i.e., the result of removing the $\varepsilon$ component of $v$). Then $P$ is included in another polyhedron $Q$ if and only if $\widehat{P} \subseteq \widehat{Q}$ and $\forall v \in \widehat{V}^{>0},\ v\!\downarrow \in Q$ .

## 4.4 Applications to examples

Let us illustrate the use of our method on the examples given in section 4.2. We will give some details about the analysis of the water-level monitor, and simply the results obtained for the other two examples.

### 4.4.1 Water-level monitor

The system of equations corresponding to the water-level monitor is the following:

$$\begin{aligned}
P_{\ell_0} &= ((\{w = 1\} \sqcup (P_{\ell_3} \cap \{x = 2\} \cap \{w < 10\})) \nearrow \{\dot{x} = \dot{w} = 1\}) \cap \{w < 10\} \\
P_{\ell_1} &= (((P_{\ell_0} \cap \{w = 10\})[x := 0] \cap \{x < 2\}) \nearrow \{\dot{x} = \dot{w} = 1\}) \cap \{x < 2\} \\
P_{\ell_2} &= (((P_{\ell_1} \cap \{x = 2\} \cap \{w > 5\})) \nearrow \{\dot{x} = 1, \dot{w} = -2\}) \cap \{w > 5\} \\
P_{\ell_3} &= (((P_{\ell_2} \cap \{w = 5\})[x := 0] \cap \{x < 2\}) \nearrow \{\dot{x} = 1, \dot{w} = -2\}) \cap \{x < 2\}
\end{aligned}$$

We choose $\ell_0$ as the only widening location, thus replacing $P_{\ell_0}$'s definition by

$$P_{\ell_0} = (P_{\ell_0} \nabla ((\{w\!=\!1\} \sqcup (P_{\ell_3} \cap \{x\!=\!2\} \cap \{w\!<\!10\})) \nearrow \{\dot{x}\!=\!\dot{w}\!=\!1\})) \cap \{w\!<\!10\}$$

The resolution converges after 3 iterations, needs 0.3 seconds of CPU time, and provides the following results:

$$\begin{aligned}
P_{\ell_0} &= \{1 \le w < 10\} \\
P_{\ell_1} &= \{w = x + 10 \wedge 0 \le x < 2\} \\
P_{\ell_2} &= \{2x + w = 16 \wedge 4 \le 2x < 11\} \\
P_{\ell_3} &= \{2x + w = 5 \wedge 0 \le x < 2\}
\end{aligned}$$

from which we can easily conclude that

$1 \le w < 10$ in location $\ell_0$ $\quad$ $10 \le w < 12$ in location $\ell_1$

$5 < w \le 12$ in location $\ell_2$ $\quad$ $1 < w \le 5$ in location $\ell_3$

### 4.4.2 Fischer mutual exclusion protocol

We give the results of the analysis of the mutual exclusion protocol of §4.2.2:

$$\begin{aligned}
P_{\ell_0} &= \{a \geq 0 \wedge b \geq 0\} \\
P_{\ell_1} &= \{b \geq 0 \wedge 0 \leq x \leq a\} \\
P_{\ell_2} &= \{a \geq 0 \wedge b \geq 0 \wedge 9x \leq 10y \leq 11x\} \\
P_{\ell_3} &= \{a \geq 0 \wedge b \geq 0 \wedge 9x \leq 10y \leq 11x\} \\
P_{\ell_4} &= \{a \geq 0 \wedge b \geq 0 \wedge 9x \leq 10y \leq 11x \wedge b \leq x\} \\
P_{\ell_5} &= \{0 \leq b \leq x \wedge 9x \leq 10y \leq 11x \wedge 9b \leq 10a \wedge 10a + 11x \geq 10y + 11b\}
\end{aligned}$$

These results are obtained after two iterations, and the analysis takes 0.4 seconds of CPU time. Remember that $\ell_5$ is the location where the mutual exclusion can be violated. Since, in $P_{\ell_5}$, we have the constraint $10a \geq 9b$, this location is not reachable if $9b > 10a$. So, the analysis shows that $9b > 10a$ is a sufficient condition for the mutual exclusion to hold. In fact, this condition is also necessary, which shows that the analysis is precise. This example shows how, by dealing with symbolic constants (here the delays $a$ and $b$), the analysis can be used to adjust some parameters to ensure a desired property.

### 4.4.3 The scheduler

For the scheduler (§4.2.3), the analysis gives (in 4 iterations and 0.6 sec.) the following results:

in the "idle" location: $c_1 \geq 0 \wedge c_2 \geq 0 \wedge x_1 = x_2 = k_1 = k_2 = 0$

"task 1" running: $k_1 \geq 1 \wedge 0 \leq x_1 \leq 4 \wedge c_1 \geq 0 \wedge c_2 \geq 0 \wedge k_2 = x_2 = 0$

"task 2" running: $0 \leq c_2 = x_2 \leq 8 \wedge 0 \leq x_1 \leq 4 \wedge c_1 \geq 0 \wedge x_1 \leq 4k_1 \wedge k_2 = 1$

So the analysis succeeds in showing that $k_2 \leq 1$, which means that a task of the second class never waits. But it fails to show that $k_1 \leq 2$, a fact that clearly appears from a straightforward simulation. Notice that this imprecision is not due — as it often happens — to the widening, but comes from the convex hull approximation[3].

## 5 Implementation: The Polka Tool

This work has been implemented into a tool, named Polka, which offers the following services:

A library of operations on polyhedra. The computations are made in rational numbers.

A "desk calculator", built on top of the library, allows the available operations to be interactively invoked.

A prototype analyzer of hybrid automata. Automata are described in a hybrid extension of the Argos formalism [Mar92].

[3]For the same example, [HH94] solves the problem by changing the control structure of the automaton, by distinguishing the locations according to the values of $k_1$ and $k_2$. They get exact results, but of course, such a change in the structure could not be found automatically, and may involve an explosion of the size of the automaton.

# 6 Conclusion

In this paper, we adapted Linear Relation Analysis to the verification of two classes of real-time systems, which appeared to be particularly good application fields for this technique:

In the delay analysis presented in Section 3, the considered variables are counters. Now, from the reduced set of operations allowed on counter variables, the range of these variables is very likely to be a convex polyhedron — or, more precisely, the set of points with integer coordinates that belong to a convex polyhedron.

Hybrid systems, considered in Section 4, are also good application field: Linear Relation Analysis is well-suited to dense domains, and, of course, the restriction to linear hybrid automata fits well with the capabilities of the analysis.

For many classical examples, the analysis succeeds in proving the relevant properties, and often provides precise results. In case of failure, the precision can be improved by several means:

It is often the case that the widening operation looses too much information. In such a case, the results can be improved by delaying the application of the widening: The widening approximation is only applied after $n$ iterations, where $n$ is a parameter of the analysis tool, which allows a convenient compromise to be chosen between the precision of the result and the complexity of the analysis.

When the convex approximation is responsible of the failure (as in the scheduler example, §4.4.3), the results can be improved by choosing a more detailed control structure. This aspect deserves further investigations: In many cases, the number of control states is prohibitive. So, the choice of a suitable control structure with respect to the property to be proved, is an important problem; the use of symbolic BDD-based techniques for the control part [DWT95] should be considered for that.

A last solution, when the verification fails, is to apply *backward analysis*: Let $Bad_0$ be the set of states that violate the property and have been found reachable by the analysis. One can compute backward (using a precondition function) an upper approximation of the set $Bad_1$ of states that can lead to $Bad_0$, and show that $Bad_1$ does not intersect the set $Init$ of initial states. Otherwise, the process can be iterated, by forward computing an upper approximation of the states reachable from $Bad_1 \cap Init$ and so on. Such forward/backward approximations have been applied by [HH94].

An important goal of the paper was to show that approximate analysis by abstract interpretation is an interesting alternative (or complement) both to (finite) state exploration and to theorem proving.

In the finite state case, approximate analysis can avoid state explosion: from a practical point of view, a theoretically terminating decision procedure is of little interest if its high complexity limits its usage to toy problems. This idea of using approximation to speed up the convergence of verification based on state exploration encounters some success in the field of timed and hybrid systems [HH94, WTD95, DWT95].

For undecidable problems, approximate analysis provides fully automatic tools for conservative verification. Even when the verification fails, the synthesized invariants can be used in a theorem proving approach.

*Acknowledgements: We are indebted to regretted Hervé Leverge for his very efficient algorithm computing the convex hull, which is the basis of our implementation. We thank also Pascal Raymond, who gave us the basic idea for dealing with strict inequalities, and Florence Maraninchi, who adapted the Argos compiler which allowed significant examples to be analyzed.*

## References

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993. Preliminary version appears in the Proc. of 5th LICS, 1990.

[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.

[ACHH93] R. Alur, C. Courcoubetis, T. A. Henzinger, and Pei-Hsin Ho. Hybrid automata: an algorithmic approach to the specification and analysis of hybrid systems. In *Workshop on Theory of Hybrid Systems*, Lyngby, Denmark, October 1993. LNCS 736, Springer Verlag.

[AHH93] R. Alur, T. A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. In *RTTS93*, 1993.

[AL91] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In J.W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, REX *Workshop on Real-Time: Theory in Practice, DePlasmolen (Netherlands).* LNCS 600, Springer Verlag, June 1991.

[BS91] F. Boussinot and R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[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.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(1–4):103–179, 1992. (Also, Research Report LIX/RR/92/08, Ecole Polytechnique).

[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.

[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.

[Che68] N. V. Chernikova. Algorithm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.

[DWT95] D. Dill and H. Wong-Toi. Verification of real-time systems by successive over- and under-approximations. In P. Wolper, editor, *7th International Conference on Computer Aided Verification, CAV'95*, Liege (Belgium), July 1995. LNCS 939, Springer Verlag.

[Hal79] N. Halbwachs. Détermination automatique de relations linéaires vérifiées par les variables d'un programme. Thèse de 3e cycle, University of Grenoble, March 1979.

[Hal93a] N. Halbwachs. Delay analysis in synchronous programs. In *Fifth Conference on Computer-Aided Verification, CAV'93*, Elounda (Greece), July 1993. LNCS 697, Springer Verlag.

[Hal93b] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.

[HH94] T. A. Henzinger and P.-H. Ho. Model checking strategies for hybrid systems. In *Conference on Industrial Applications of Artificial Intelligence and Expert Systems*, 1994.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 29(6/7):523–543, 1992.

[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.

[HPR94] N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In B. LeCharlier, editor, *International Symposium on Static Analysis, SAS'94*, Namur (Belgium), September 1994. LNCS 864, Springer Verlag.

[IEE91] Another look at real-time programming. *Special Section of the Proceedings of the IEEE*, 79(9), September 1991.

[KPSY93] Y. Kesten, A. Pnueli, J. Sifakis, and S. Yovine. Integration graphs: a class of decidable hybrid systems. In *Workshop on Theory of Hybrid Systems*, Lyngby, Denmark, October 1993. LNCS 736, Springer Verlag.

[Lam87] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[LeV92] H. LeVerge. A note on Chernikova's algorithm. Research Report 635, IRISA, February 1992.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR'92*, Stony Brook, August 1992. LNCS 630, Springer Verlag.

[MMP91] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In REX *Workshop on Real-Time: Theory in Practice*, DePlasmolen (Netherlands), June 1991. LNCS 600, Springer Verlag.

[Rus94] J. Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *13th ACM Symp. on Principles of Distributed Computing, PODC'94*, Los Angeles, August 1994.

[WTD95] H. Wong-Toi and D. Dill. Aproximations for verifying timing properties. In *Theories and Experiences for Real-Time System Development*, chapter 7. World Scientific, 1995.