Delay analysis in synchronous programs[1]

Nicolas Halbwachs
IMAG Institute and Stanford University[2]

Abstract: Linear relation analysis [CH78, Hal79] has been proposed a long time ago as an abstract interpretation which permits to discover linear relations invariantly satisfied by the variables of a program. Here, we propose to apply this general method to variables used to count delays in synchronous programs. The "regular" behavior of these counters makes the results of the analysis especially precise. These results can be applied to code optimization and to the verification of real-time properties of programs.

1 Introduction

Synchronous programming has been proposed [IEE91] 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 which expresses that something will never happen, and by a *simple logical* property, we mean a property which 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*, i.e., another program

[1] This work was supported by the Department of the Navy, Office of the Chief of Naval Research under Grant N00014-91-J-1901, and by a grant from the Stanford Office of Technology Licensing. This publication does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement of this work should be inferred.

[2] Author's permanent adress: IMAG, B.P. 53X, 38041 Grenoble Cedex, France. This work was performed while the author was on leave in Stanford University.

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.

This paper attempts to solve the problem of taking numerical delays into account in the generation of automata. Let us take a small example, in ESTEREL[3]: We consider a car, about which we know that *(1)* it stops within 4 seconds, and *(2)* 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 process which 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:

[3] 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.

```
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.
```

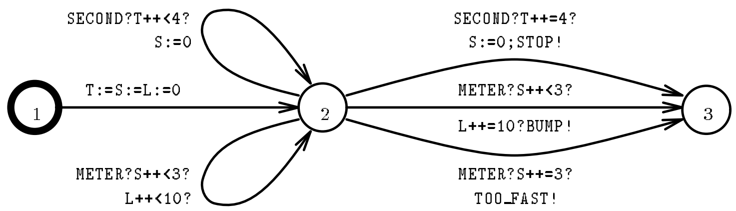


Fig. 1. An interpreted automaton

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. 1 (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 `L` for counting 10 meters (the length). 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 will apply a general method, that was proposed quite a long time ago but little applied, to discover linear relations among numerical variables of a program. After recalling in Section 2 the principles of this method, together with specific optimizations (Section 3), we will see on some examples that it gives particularly precise results when applied to counters (Section 4).

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 program. 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, together with functions defined on the domain. 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 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 ∇, 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 — $\overline{y} = \lim_{i \geq 0} y_i$, with $y_0 = \bot$ (the least element of A) and $y_{i+1} = G(y_i)$ — , we can compute an *ascending approximation sequence* $(y'_i)_{i \geq 0}$: $y'_0 = \bot$, $y'_{i+1} = y'_i \nabla G(y'_i)$, 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* $y''_0 = \widetilde{y}, y''_{i+1} = G(y''_i)$, 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. 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 is very common in sequential program analysis, where K often represents the set of control points. It 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_∇ 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_∇:

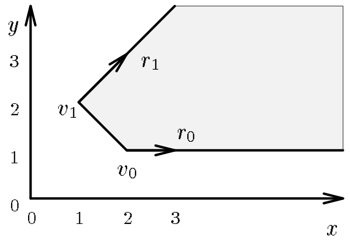
$$\begin{array}{ll}
\forall k \in K, & y'^{(k)}_0 = \bot \\
\forall k \in K_\nabla, & y'^{(k)}_{i+1} = y'^{(k)}_i \nabla G^{(k)}(y^{(1)}_i, y^{(2)}_i, \ldots, y^{(|K|)}_i) \\
\forall k \in K \setminus K_\nabla, & y'^{(k)}_{i+1} = G^{(k)}(y^{(1)}_i, y^{(2)}_i, \ldots, y^{(|K|)}_i)
\end{array}$$

The advantage is that the widening operator, which is the one which looses 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 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 convex polyhedron $P \in \mathcal{P}(\mathbb{Q}^n)$ will represent the set of states $\gamma(P) = \{\langle X, s' \rangle \mid X \in P \cap N^n \ , \ s' \in S'\}$.



$$P = \left\{ (x,y) \mid \begin{pmatrix} y \geq 1 \\ x+y \geq 3 \\ -x+y \leq 1 \end{pmatrix} \right\}$$

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

$$L = \emptyset$$

Fig. 2. A convex polyhedron and its 2 representations

So, our abstract values are convex polyhedra. Let us recall that a convex polyhedron P (a polyhedron, for short) has two representations (see Fig. 2):

• 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., three finite sets V, R, and L (respectively for "vertices," "rays," and "lines") of n-vectors such that

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

These two representations are dual. There exist efficient algorithms [Che68, LeV92] for translating each representation into the other; these algorithms also minimize the representations. We will use the following basic operations on polyhedra (see Fig. 3 and 4):

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 a linear expression to a variable. Here, we consider only very simple cases — variable reset, increment, and decrement — but the general case is similar.

• Let $P[0/X_i] = \{(X_1, X_2, \ldots, X_{i-1}, 0, X_{i+1}, \ldots, X_n) \mid X \in P\}$ be the result of resetting to zero the i-th variable in a polyhedron P. The vertices (respectively, the rays, the lines) of $P[0/X_i]$ are obtained by setting to 0 the i-th coordinate of the vertices (resp., the rays, the lines) of P.

• The result $P[X_i{+}1/X_i] = \{(X_1, X_2, \ldots, X_{i-1}, X_i{+}1, X_{i+1}, \ldots, X_n) \mid X \in P\}$ of incrementing the i-th variable in a polyhedron P can be computed easily on both representations: if $AX \geq B$ is the system of inequalities of P, then $AX \geq (B + A^{(i)})$ (where $A^{(i)}$ is the i-th column of A) is the system of constraints of $P[X_i{+}1/X_i]$. The vertices of $P[X_i{+}1/X_i]$ are obtained by incrementing the

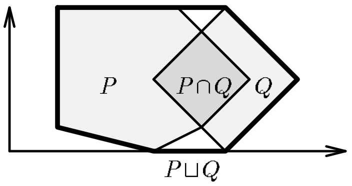


Fig. 3. Intersection and convex hull

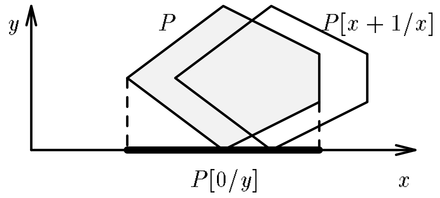


Fig. 4. Linear transformations

i-th coordinate of those of P, the rays and the lines don't change. Variable decrement is performed symmetrically.

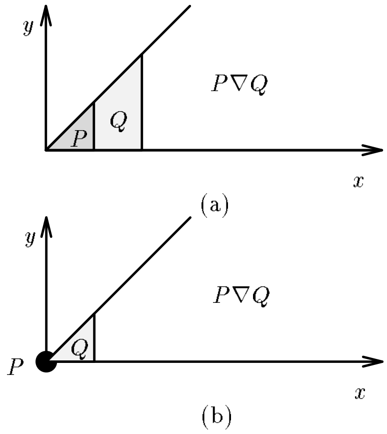
Test for emptyness: A polyhedron is empty if and only if it has no vertices.

Test for inclusion and equality: A polyhedron P, with system of generators (V, R, L), is included in a polyhedron Q, defined by the system of inequalities $AX \geq B$, if and only if $\forall v \in V,\ Av \geq B \quad \wedge \quad \forall r \in R,\ Ar \geq 0 \quad \wedge \quad \forall \ell \in L,\ A\ell = 0$. 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 which are not satisfied by Q. Fig. 5.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 a constraint 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 can be first rewritten into an equivalent system maximizing the number of inequalities satisfied by Q, and thus kept in the result. For instance, consider:

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



(a)

(b)

Fig. 5. Widening operation

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. 5.b) instead of $\{(x, y) \mid 0 \leq y \wedge 0 \leq x\}$, which would be obtained without rewriting. This optimization preserves the widening properties. An efficient algorithm has been proposed for it [Hal79].

3 Application to delay analysis

3.1 Automata

We will apply the linear relation analysis to automata produced by synchronous languages compilers. Such an automaton is a finite set of states, each of which being associated with a piece of sequential code. The sequential code executed in a state is linear, in the sense that it contains neither loop nor recursion. It 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 state of the automaton. These statements terminate the code executed in a state.

Fig. 6 gives the code of the automaton shown in Fig. 1.

We will take advantage of this control structure to get a partitioned system. A state of the program is a triple (s, X, Y), where s is a state 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 state s of the automaton, we will associate a polyhedron P_s, which will be an approximation of the set $\{X \mid \exists Y,\ (s, X, Y) \text{ is a reachable state of the program}\}$.

```
State 1 ........................................................ {P1}
  T=0; L=0; S=0; goto State 2 .................................. {P1,1,2}

State 2 ........................................................ {P2}
  if SECOND then S=0;
    if T++=4 then emit STOP; goto State 3 ...................... {P1,2,3}
    end;
    goto State 2 ............................................... {P1,2,2}
  end;
  if METER then
    if S++=3 then emit TOO_FAST; goto State 3 .................. {P2,2,3}
    end;
    if L++=10 then emit BUMP; goto State 3 ..................... {P3,2,3}
    end;
    goto State 2 ............................................... {P2,2,2}
  end;
  goto State 2 ................................................. {P3,2,2}

State 3 ........................................................ {P3}
   goto State 3 ................................................ {P1,3,3}
```

Fig. 6. The code of the automaton, with associated polyhedra

Since we are interested in determining what transitions can occur and what states can be reached, we will also associate a polyhedron with each branching statement, which will approximate the set of reachable states of the counters when executing those statements: Let $P_{i,s',s}$ be the polyhedron associated with the i-th "`goto state` s" statement appearing in the code of state s'. Fig. 6 shows the polyhedra to be computed for our small example. Clearly P_s is the convex hull of all the $P_{i,s',s}$, and $P_{i,s',s}$ is computed from $P_{s'}$ according to the statements executed along the branch leading to the i-th "`goto state` s" appearing in the code of s'. The transformation of polyhedra resulting from assignments is straightforward. 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 $F_t = \lambda P.P \cap \{X \mid X_i = k\}$, $F_f = \lambda P.(P \cap \{X \mid X_i \geq k+1\}) \sqcup (P \cap \{X \mid X_i \leq k-1\})$

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:

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

3.2 Widening strategies

The points where the widening is performed are selected among state entry points. Although the ESTEREL compiler generates a dummy transition looping on each state, 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 state in each loop of such transitions. In our example, we select state 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 constraints, say M, and define a new "widening up to M" operator ∇_M as follows: $P \nabla_M Q$ is the intersection of the standard widening $P \nabla Q$ with all the constraints 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 constraints M is associated with each widening state. This set is selected to be all the linear relations which make the control remain in the state. The intuition behind this choice is the following: Assume s is a state whose only outgoing transition is guarded by the condition "`x++=10`" and that s is entered with `x=0`. Then, since the control remains in s (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 s. In our example, the state 2 is left when either `T++=4` or `S++=3` or `L++=10`. The set of constraints limiting the widening is $\{T \leq 3\ ,\ T \geq 5\ ,\ S \leq 2\ ,\ S \geq 4\ ,\ L \leq 9\ ,\ L \geq 11\}$.

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.

4 Examples

4.1 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 §3.1. Let us recall (cf. §3.2) that the only widening state is State 2, and that the widening is performed up to the following constraints: $M = \{T \leq 3\ ,\ T \geq 5\ ,\ S \leq 2\ ,\ S \geq 4\ ,\ L \leq 9\ ,\ L \geq 11\}$. The successive computation steps are the following:

Step 0: Initially, all the polyhedra are empty.

Step 1: The first iteration in the loop provides:

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

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

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

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

and $P^{(2)}_{1,1,2} \sqcup P^{(2)}_{1,2,2} \sqcup P^{(2)}_{2,2,2} = \{0 \leq S \leq L \leq 2T + S, L \leq 2, T \leq 3\}$

Step 3:

$$\begin{aligned}
P^{(3)}_{2} &= \{0 \leq S \leq L \leq 2T + S, T \leq 3, L \leq 2\} \\
P^{(3)}_{1,2,2} &= \{S = 0, 0 \leq L \leq 2, 1 \leq T \leq 3\} \\
P^{(3)}_{2,2,2} &= \{1 \leq S \leq L \leq 2T + S, T \leq 3, L \leq 3, S \leq 2\}
\end{aligned}$$

and $P^{(3)}_{1,1,2} \sqcup P^{(3)}_{1,2,2} \sqcup P^{(3)}_{2,2,2} = \{0 \leq S \leq L \leq 2T + S, L \leq S + 2, T \leq 3, L \leq 3, S \leq 2\}$

Step 4:

$$\begin{aligned}
P^{(4)}_{2} &= \{0 \leq S \leq L \leq 2T + S, T \leq 3, S \leq 2\} \\
P^{(4)}_{1,2,2} &= \{S = 0, 0 \leq L \leq 2T, 1 \leq T \leq 3\} \\
P^{(4)}_{2,2,2} &= \{1 \leq S \leq L \leq 2T + S, T \leq 3, S \leq 2\}
\end{aligned}$$

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

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

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

```
State 1
  T=0; L=0; S=0;
  goto State 2.............................................{T = S = L = 0}

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

State 3.................{3T + S ≤ 12 ≤ 3T + 4S, 2S ≤ 3L ≤ 6T + 2S, S ≤ 3}
   goto State 3
```

Fig. 7. Results of the analysis of the "car" example

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.

4.2 The "train-gate controller" revisited

Let us complexify an example considered in [Alu91], introducing some multiform-time problems: It is an automatic controller that opens and closes a gate at a railway track intersection (We do not pretend it corresponds to a realistic system!).

When the train approaches the gate (see Fig. 8), it pushes a pedal. 800 meters farther, it crosses a signal:

• if the signal is red, the train puts on the brakes, and stops within 550m;

• otherwise, it crosses the gate 770m after the signal, and pushes an exit pedal 1000m after the gate.

Two seconds after receiving a signal from the entry pedal, the controller commands the gate closure. Normally, the gate closes within 7s, but it may fail. So, 8s after the closure command:

• if the gate is not closed, the controller switches the signal on red, to stop the train;

• otherwise, it waits for the signal from the exit pedal and commands the gate openning.

The maximum speed of the train is 69m/s.

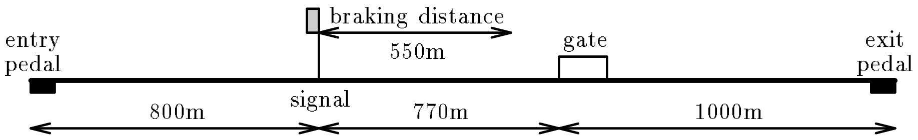


Fig. 8. The train-gate example

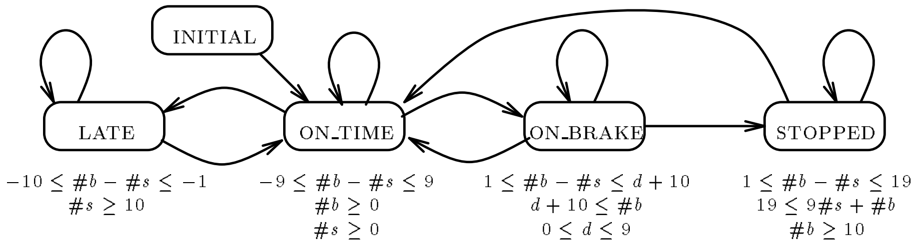


Fig. 9. Result of the subway example, with one train

From a program simulating this system, the ESTEREL compiler generates an automaton with 104 states and 451 transitions. The analysis finds that 82 of these states are unreachable. Removing the unreachable part, we get an automaton with 22 states and 56 transitions. Every transition on which `ON_GATE` is emitted also emits `GATE_CLOSED`.

So, this example shows that the analysis results can be used not only to prove program properties, but also to reduce the size of the automaton and optimize the object code.

4.3 A subway speed regulation system

Our last 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. 9. 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.

5 Conclusion

The symbolic program analysis methods were strongly studied in the late seventies; in program verification, they were afterward almost completely abandoned because of the first success of enumerative model-checking. Now, in view of the development of boolean symbolic model-checking, thanks to Binary Decision Diagrams, many people are trying again to apply symbolic techniques (e.g.,[DKK91, MPS92, Cor92]) to numerical systems.

For taking delays into account in the analysis of synchronous program, our first idea was to adapt the model-checking techniques developed for timed automata [ACD90, ACH$^+$92]. Unfortunately, timed automata do not take into ac-

[4] The size of the detailed automata were determined by a state graph traversal using the tool Murphi [DDHY92].

Program	number of variables	Control automaton		Detailed automaton[4]		Result automaton		Analysis time
		#states	#trans.	#states	#trans.	#states	#trans.	
Car	3	3	6	49	146	3	5	0.3s
Train	8	104	451	445,900	972,372	22	56	1.8s
Subway 1 train	4	8	41	66	103	5	27	0.7s
Subway 2 trains	7	37	297	4536	9340	17	71	16.6s

Table 1. Performances of the analysis

count the multiform time handled by synchronous programs [JMO93], and particularly the dependence relations existing between different time scales.

So, we decided to apply an approximate method. The approximations we make seem well-suited for that application field. As a matter of fact, 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 (Notice that our computations are done in rational numbers, because of the cost of linear programming in integer variables; once again, this constitutes an upper approximation).

The proposed method is a combination of automata-based methods with abstract interpretation: On the one hand, the analysis can reduce the size of the automaton, and on the other hand the automaton is used for partitioning the analysis. So, it would be interesting to combine the delay analysis and the automaton generation, both to avoid the generation of unreachable states and to choose the best partition.

Up to now, only a rough prototype has been implemented, for experimentation purposes. It is based on the very efficient implementation of Chernikova's algorithm provided by [LeV92]. The Table 1 gives the analysis times for the considered examples. All our examples have been presented in ESTEREL, but as soon as the analyzer will be fully interfaced with the common intermediate code of synchronous languages [PS87], it will be applicable to any language using this code.

References

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model checking of real-time systems. In *Fifth IEEE Symposium on Logic in Computer Science, Philadelphia*, 1990.

[ACH+92] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi. Minimization of timed transition systems (extended abstract). In *CONCUR'92, Stony Brook*. LNCS 630, Springer Verlag, August 1992.

[Alu91] R. Alur. Techniques for automatic verification of real-time systems. Phd thesis, Stanford University, August 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*, January 1977.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. Research Report LIX/RR/92/08, Ecole Polytechnique, March 1992. (to appear in the Journal of Logic Programming, special issue on Abstract Interpretation).

[CC92b] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. Research Report LIX/RR/92/09, Ecole Polytechnique, June 1992.

[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, 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.

[Cor92] J. C. Corbett. Verifying general safety and liveness properties with integer programming. In G. Bochmann, editor, *4th Int. Workshop on Computer Aided Verification, Montreal*, 1992.

[DDHY92] D. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *1992 IEEE Int. Conference on Computer Design: VLSI in Computers and Processors*, 1992.

[DKK91] S. Devadas, K. Kreutzer, and A. S. Krishnakumar. Design verification ansd reachability analysis using algebraic manipulation. In *ICCD'91*, 1991.

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

[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), 1992.

[IEE91] Another look at real-time programming. *Special Section of the Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[JMO93] M. Jourdan, F. Maraninchi, and A. Olivero. Verifying quantitative real-time properties of synchronous programs. In *Fifth Int. Workshop on Computer Aided Verification, Elounda (Crete)*, July 1993.

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

[MPS92] E. Macii, B. Plessier, and F. Somenzi. Verification of systems containing counters. In *ICCAD'92*, 1992.

[PS87] J. A. Plaice and J-B. Saint. The LUSTRE-ESTEREL portable format. Unpublished report, INRIA, Sophia Antipolis, 1987.

This article was processed using the LaTeX macro package with LLNCS style