Dynamic Partitioning in Analyses of Numerical Properties*

Bertrand Jeannet, Nicolas Halbwachs, Pascal Raymond
{Bertrand.Jeannet,Nicolas.Halbwachs,Pascal.Raymond}@imag.fr

Vérimag** , Grenoble – France

Abstract. We apply linear relation analysis [CH78,HPR97] to the verification of declarative synchronous programs [Hal98]. In this approach, *state partitioning* plays an important role: on one hand the precision of the results highly depends on the fineness of the partitioning; on the other hand, a too much detailed partitioning may result in an exponential explosion of the analysis. In this paper, we propose to dynamically select a suitable partitioning according to the property to be proved.

1 Introduction

Partitioning in abstract interpretation: We address the standard case of abstract interpretation [CC77], where a set S of *concrete states* is considered, and where a complete lattice $(L, \sqsubseteq, \sqcap, \sqcup)$ of *abstract values* is associated with the powerset 2^S by means of a Galois connection (α, γ). In this context, *partitioning* a fixpoint equation $X = F(X)$ — where F is a monotone function from L to L — consists in choosing a finite partition (or, more generally, a finite covering) $\{S_1, \ldots, S_k\}$ of S (i.e., such that $S = \bigcup_{i=1}^{k} S_k$), and by replacing the initial equation by the *system* of equations

$$X_i = F_i(X_1, \ldots, X_k),\ i = 1 \ldots k$$

where $X_i = X \sqcap L_i$ and $L_i = \alpha(S_i)$. In general, the advantages of partitioning are that the component functions $(F_i)_{i=1..k}$ are simpler than F, and that the system can be more efficiently solved, using chaotic iterations. It can be even more interesting in two common cases:

- When a widening is necessary to ensure the termination of iterations, the widening only needs to be applied on some equations (cutting dependence loops [Bou92b,CC92b]). Thus, less information is lost because of the widening.

* This work was partially supported by the ESPRIT-LTR project "SYRF".

** Verimag is a joint laboratory of Université Joseph Fourier (Grenoble I), CNRS and INPG associated with IMAG.

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

Fig. 1. An example "program"

– In lattices where the least upper bound (*lub*) operation $\sqcup$ loses information, i.e., when

$$\gamma(X) \cup \gamma(Y) \subsetneq \gamma(X \sqcup Y)$$

partitioning allows some operands of a *lub* operation to be considered separately (i.e., considering X_i and X_j instead of $X_i \sqcup X_j$), thus representing exact union of concrete values ([Bou92a] introduces the term "dynamic partitioning" in this context).

Examples of such cases are intervals [CC76] or linear relations [CH78] analyses.

In the analysis of imperative sequential programs, a natural partitioning is given by the control structure of the program. A concrete state is a pair $\langle c, v\rangle$, where c is a control location, and v is a valuation of variables (memory state), and concrete states are generally partitioned according to their c components.

We are interested in the verification of synchronous data-flow programs, using abstract interpretation. Such programs don't present any control structure, but the trivial one: a single loop around a unique control location, which results in no partitioning at all. In order to get a more interesting partitioning, we used the *control automaton*, produced by some compilers by performing a partial evaluation [JGS93] of all Boolean variables of the program [Hal98].

Let us consider a trivial example: the "program" shown on Fig. 1 is, basically, a system of equations defining the dynamic behavior of a set of Boolean ("b1, b2") and integer ("x, y") variables. Their possible initial states are specified by a formula ("initial"), then each equation defines the "next" (primed) value of a variable as an expression of the current values. Fig. 2 (a) and (b) show two imperative versions of this program: Fig. 2.a is the trivial control structure, while Fig. 2.b shows the automaton obtained by a partial evaluation of Boolean variables.

The problem is that the automaton produced by partial evaluation may have an exponential number of control locations (Boolean states). Of course, partitioning a fixpoint equation into a system with millions of equations is not realistic. Moreover, the obtained control structure is not always suitable to obtain precise enough results: in some cases, control locations should be split again, to delay the application of a *lub* operator. For instance, in the example above, if our goal is to prove, by linear relation analysis, that x is always greater or equal to y, the

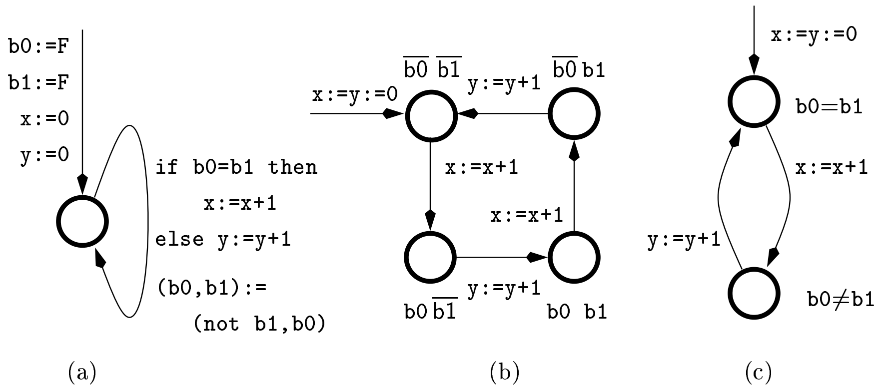


Fig. 2. Control structures for the example program

control structure of Fig. 2.a is too coarse (the behavior of Boolean variables is ignored); conversely, the partitioning of Fig. 2.b is too much detailed: we could complexify the program, with n Boolean variables, and get a ring of 2^n locations, while the property can indeed be shown on an automaton with only 2 locations (Fig. 2.c).

We use abstract interpretation — and, more precisely, Linear Relation Analysis — to verify safety properties of program, i.e., to show that some "bad" states cannot be reached during the program execution. In this context, and as a solution to the above problem, we propose to dynamically choose the partitioning, according to the property we want to prove. Dynamic partitioning consists in starting from a very coarse control structure (basically, distinguishing only between initial states, bad states, and others), and to refine it according to heuristics, until the unreachability of bad states from initial ones has been shown. Our goal is to approach the simplest control structure, in order to prove the property.

2 Definitions and Notations

Let us note $\mathcal{B} = \{T, F\}$ the set of Boolean values, and $\mathcal{N}$ the set of numerical values (which may be integer and/or rational numbers). For a program with p Boolean variables and q numerical variables, the set S of concrete states is $\mathcal{B}^p \times \mathcal{N}^q$. The program defines a set S_{init} of initial states, and a transition relation[1] $\rightarrow \subseteq S \times S$. Moreover, for the sake of verification, the program is supposed to be equipped with an "invariant", i.e., a formula specifying the property one want to prove. We note S_{error} the set of "bad" states that violate the invariant. The goal is to show that there is no path $s_0, s_1, \ldots, s_n$, with $s_0 \in S_{init}$, $s_n \in S_{error}$,

[1] In our implementation, we consider deterministic programs with inputs, instead of the non-deterministic ones which are considered here, for simplicity.

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

Fig. 3. A program with a property

and for all $i = 1..n$, $s_{i-1} \rightarrow s_i$. For instance, Fig. 3 shows our example program augmented with an "invariant", which becomes false whenever x<y.

The lattice of abstract values we consider is $L = 2^{\mathcal{B}^p} \times L_{\mathcal{N}}$, where $L_{\mathcal{N}}$ is a set of abstract values for numerical domains: there is a Galois connection $(\alpha_{\mathcal{N}}, \gamma_{\mathcal{N}})$ between $2^{\mathcal{N}^q}$ and $L_{\mathcal{N}}$. Throughout the paper, $L_{\mathcal{N}}$ may be either the lattice $\mathcal{I}(\mathcal{N})^q$ of tuples of intervals (interval analysis), or the lattice $\mathcal{P}(\mathcal{N}^q)$ of convex polyhedra of $\mathcal{N}^q$ (linear relation analysis). We assume the existence of standard operations $\sqcap_{\mathcal{N}}, \sqcup_{\mathcal{N}}, \nabla_{\mathcal{N}}$ in $L_{\mathcal{N}}$.

An abstract value is thus a pair (f, D), where f is a Boolean formula representing a subset of $\mathcal{B}^p$, and $D \in L_{\mathcal{N}}$. It represents the set of concrete states

$$\gamma(f, D) = \{\langle \boldsymbol{b}, \boldsymbol{v} \rangle \mid \boldsymbol{b} \models f,\ \boldsymbol{v} \in \gamma_{\mathcal{N}}(D)\ \}$$

Operations on abstract values are derived straightforwardly:

$$\begin{aligned}
(f_1, D_1) \sqcap (f_2, D_2) &= (f_1 \wedge f_2, D_1 \sqcap_{\mathcal{N}} D_2) \\
(f_1, D_1) \sqcup (f_2, D_2) &= (f_1 \vee f_2, D_1 \sqcup_{\mathcal{N}} D_2) \\
(f_1, D_1) \nabla (f_2, D_2) &= (f_1 \vee f_2, D_1 \nabla_{\mathcal{N}} D_2)
\end{aligned}$$

Notice that the *lub* operator $\sqcup$ loses information, not only because $\sqcup_{\mathcal{N}}$ loses information in the numerical lattices we consider, but also because the Boolean and numerical parts are considered separately: $(b \wedge x \geq 0) \sqcup (\,\overline{b} \wedge x < 0) = (T, T)$ instead of $(b \equiv (x \geq 0))$.

With these tools, we are able to perform both forward and backward analyses on our programs.

3 Control Structures

A control structure is an automaton $(Q, Q_0, Q_F, \rightsquigarrow, \mathit{def})$, where

- Q is a finite set of *locations*, Q_0 and Q_F are subsets of Q (respectively called sets of initial and final locations).

- def is a function from Q to L, such that $\bigcup_{q \in Q} \gamma(def(q)) = S$. In other words, $\{\gamma(def(q)) \mid q \in Q\}$ is a covering of S. Moreover, Q_0 and Q_F are such that

$$S_{init} \subseteq \bigcup_{q \in Q_0} \gamma(def(q)) \text{ and } S_{error} \subseteq \bigcup_{q \in Q_F} \gamma(def(q))$$

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

$$\Big(\exists s \in \gamma(def(q)), \exists s' \in \gamma(def(q')) \text{ such that } s \to s'\Big) \Rightarrow q \rightsquigarrow q'$$

In other words, a control structure is an abstraction of the program. Each location q represents a set of concrete states $\gamma(def(q))$ exactly represented by an abstract value $def(q)$, and the relation transition is an upper approximation of the program transition relation. The important point is that, if there is no path in the automaton from an initial to a final location, then no bad state can be reached from an initial state in the program.

3.1 Analysis on a Control Structure

We use forward and backward analyses on control structures. Forward analysis computes, for each location q, an upper approximation $reach(q)$ of the set of states in $\gamma(def(q))$ that are reachable from an initial state of the program. Backward analysis computes, for each location q, an upper approximation $coreach(q)$ of the set of states in $\gamma(def(q))$ from which one can reach a bad state (these states are said to be "coreachable" from bad states).

Since we are interested in states that are both reachable from initial states, and co-reachable from bad states, we associate with each location q, an upper approximation $danger(q) \in L$ of the set of these states in $\gamma(def(q))$. Our goal is to show that, for each location q (or, equivalently, for each location q in Q_0), $danger(q) = \bot_L$. $danger(q)$ is computed by standard combination of forward and backward analyses: for any $X \in L$ let us note

$post(X)$ the upper-approximation in L of the set
$\{s' \in S \mid \exists s \in X \text{ such that } s \to s'\}$
$pre(X)$ the upper-approximation in L of the set
$\{s \in S \mid \exists s' \in X \text{ such that } s \to s'\}$

In a first step, forward analysis computes $reach(q), q \in Q$ as an upper approximation of the least solution of the system

$$reach(q) = \bigsqcup_{q' \rightsquigarrow q} post(reach(q')) \sqcap def(q)$$

After convergence of this step, one can update the relation "$\rightsquigarrow$" as:

$$q \rightsquigarrow q' \Leftrightarrow post(reach(q)) \sqcap def(q') \neq \bot_L$$

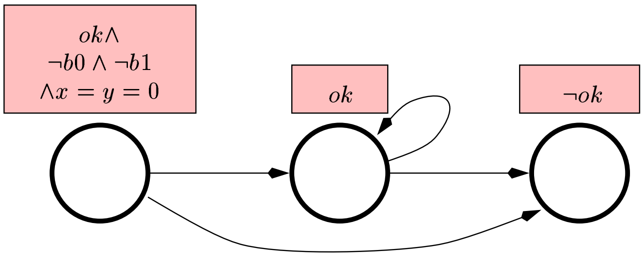


Fig. 4. Initial control structure

Then, backward analysis can be applied to compute $danger(q)$, for each $q \in Q$, as an upper approximation of the least solution of the system

$$danger(q) = \bigsqcup_{q \rightsquigarrow q'} pre(danger(q')) \sqcap reach(q)$$

Again, the relation "$\rightsquigarrow$" is updated by:

$$q \rightsquigarrow q' \;\Leftrightarrow\; pre(danger(q')) \sqcap reach(q) \neq \bot_L$$

The definitions of locations are also updated, by taking $def(q) = danger(q)$, since only dangerous states are still of interest. Then a new forward analysis can be applied to the updated structure.

Forward and backward analyses are performed this way, in alternation, until convergence.

3.2 Initial Control Structure

We start from a very coarse control structure, which distinguishes initial states, final (bad) states, and others; ideally, Q_0 (resp. Q_F) should cover exactly initial (resp. bad) states

$$\bigcup_{q \in Q_0} \gamma(def(q)) = S_{init} \quad , \quad \bigcup_{q \in Q_F} \gamma(def(q)) = S_{error}$$

Under reasonable assumptions about initial and bad states, this covering is possible. Otherwise, on can always add an extra initial and/or final state to the program, which be characterized by a purely Boolean formula. For instance, in the program of Fig 3, the invariance property "x≥y" has been "delayed" into a Boolean state variable ok, and "bad" states are those in which ok=false. In this initial control structure, before any analysis, the transition relation $\rightsquigarrow$ is assumed to be complete, except that locations in Q_0 and Q_F are, respectively, sources and sinks. Fig. 4 shows the initial control structure for our example program. On this Figure, the definition $def(q)$ of each location q is shown in a grey box.

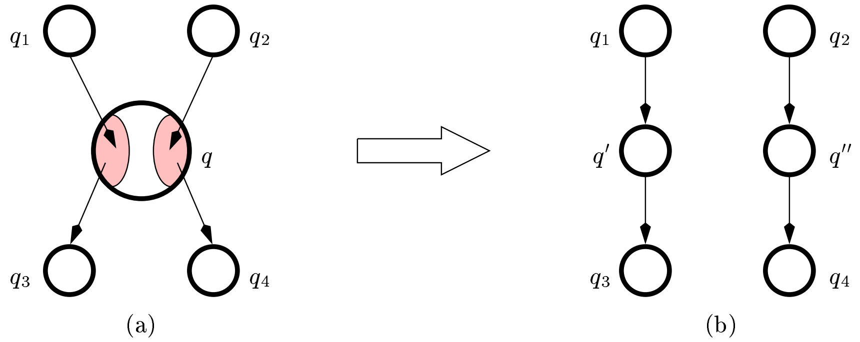


Fig. 5. Refining a location

Notice that such a detailed covering of initial and bad states results from a deliberate choice: on one hand, it starts the analysis with a precise separation of relevant states; on the other hand, the influence of this separation on the analysis cost will not be dramatic: since locations in Q_0 and Q_F never belong to loops, they will not be involved in iterations.

4 Control Structure Refinement

4.1 Principles

Let us give the intuition of the kind of refinement we want to perform. Once the initial control structure is built, the analysis sketched in section 3.1 will associate with each location q an abstract value $danger(q)$ — upper approximation of the states in $def(q)$ that are both reachable from S_{init} and coreachable from S_{error} —, and possibly reduce the transition relation $\rightsquigarrow$. The goal is to obtain

$$\forall q, danger(q) = \bot_L, \text{ and } \rightsquigarrow = \emptyset$$

and, of course, this goal is seldom achieved on the initial control structure. Refining a location q consists in noticing that concrete states in $def(q)$ don't show all the same behavior, in terms of accessibility.

For instance, consider the fragment of structure shown on Fig. 5.(a), and assume (as shown on the Figure) that:

$$\begin{aligned} post\left(post(def(q_1)) \sqcap def(q)\right) \sqcap def(q_4) &= \bot_L \\ post\left(post(def(q_2)) \sqcap def(q)\right) \sqcap def(q_3) &= \bot_L \end{aligned}$$

In such a case, splitting q into q' and q'', as shown in Fig. 5.(b), with

$$post(def(q_1)) \subseteq \gamma(def(q')) \text{ and } post(def(q_2)) \subseteq \gamma(def(q''))$$

clearly improves our knowledge about location accessibility. Now such refinements must be performed with care: arbitrary refinements may uselessly complexify the control structure, and even result in infinite computations.

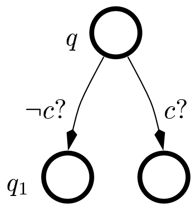
4.2 Refinement Heuristics

In this section, we explain the kind of refinements implemented in our tool, which is specialized in Linear Relation Analysis. To avoid the risk of infinite refinements, we adopt the following restriction: locations will only be split according to conditions (Boolean formulas, or linear constraints) appearing in the program.

Let q and q_1 be two locations such that $q \rightsquigarrow q_1$. Assume there exists a condition c appearing in the program, such that

$$(\mathit{def}(q) \sqcap \mathit{pre}(q_1)) \;\sqcap\; (\mathit{def}(q) \sqcap \alpha(c)) = \bot_L$$

Then q can be safely split into two locations q' and q'', with $\mathit{def}(q') = \mathit{def}(q) \sqcap \alpha(c)$ and $\mathit{def}(q'') = \mathit{def}(q) \sqcap \alpha(\neg c)$, and $q' \not\rightsquigarrow q_1$.



Notice that this case is rather common, since, in general, transitions are guarded by conditions, which naturally separate preconditions.

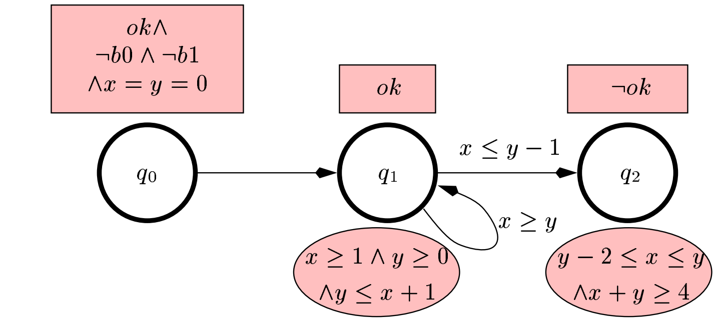
Our global refinement scheme consists in splitting one location at a time, performing (forward and backward) analysis until convergence, and repeating this process until either reaching the goal (all $\mathit{danger}(q)$ empty), or obtaining the most detailed structure according to our set of conditions. The choice of not splitting many locations at each step is to avoid unnecessary increasing of the structure: a single refinement, followed by an analysis, can eliminate the apparent need of other refinements. Of course, this choice may increase the number of steps before termination.

This simple heuristic is surprisingly efficient: on one hand, in many examples, it splits locations enough to prove the property, and on the other hand, it does not generate many useless locations.

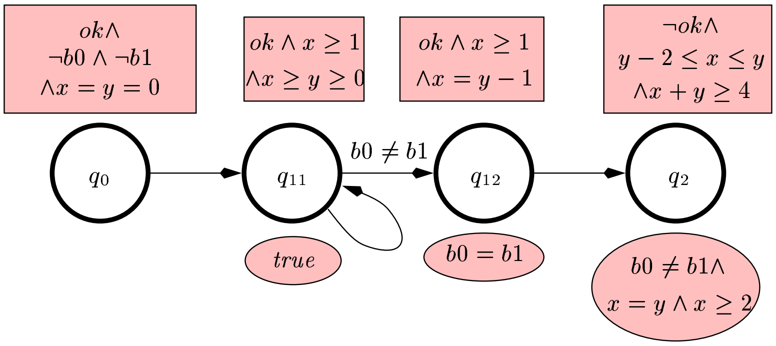
5 Examples

5.1 The simple example

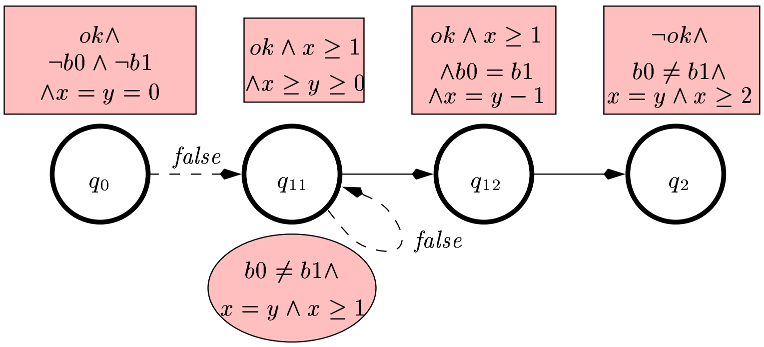
Let us illustrate the proposed technique by detailing the analysis of our simple example (Fig. 3). The initial control structure was shown by Fig. 4. Fig. 6.(a) shows the results of a forward analysis on this structure: the result for each location appears in an ellipse. Backward analysis does not give any additional information. Then, location q_1 is split according to the condition $x \geq y$ (guard of a transition, appearing in the program). Fig. 6.(b) shows the result of forward analysis on the refined structure. Finally, backward analysis disconnects the



(a) Forward analysis on the initial structure



(b) Refinement + forward analysis



(c) Backward analysis

Fig. 6. Analysis of the simple example

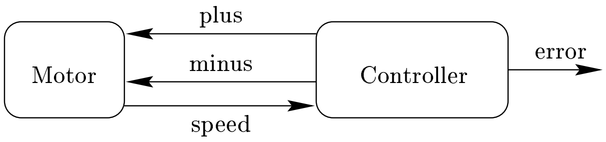


Fig. 7. CD controller

initial location (Fig 6.(c)). The subsequent forward analysis leaves the structure unchanged, thus completing the analysis.

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

5.2 A compact-disk controller

As a more realistic example, we consider a (simplified) compact-disk controller, connected to a motor as shown by Fig. 7.
The motor reacts to two logical signals, plus and minus, which make it run faster or slower, respectively. When neither of these two signals is on, the speed varies within two bounds (here, -4 and $+4$). A sensor gives the current speed. The controller tries to maintain the motor speed within a specified range (here $[8, 12]$), by sending appropriate signals plus and minus. Whenever the speed has been outside the range for a given delay (here, 15), counted as a number of steps, an error signal is sent. With our notations for "programs", Fig. 8 gives the assumptions on the motor, and the program of the controller.

```
Motor:
    speed'-speed <= 4;
    speed'-speed >= -4;
    plus => (speed'-speed > 0);
    minus => (speed'-speed < 0);
```

```
Controller:
    plus' = (speed <= 9);
    minus' = (speed >= 11);
    count' = if speed'< 8 or speed'> 12
                then count+1 else 0;
    error = (count >= 15);
```

Fig. 8. The CD controller example

We want to verify that the error signal is never sent. This means that, if the speed is initially 0, it must reach the correct range in time, and be properly controlled afterwards. The system has 2 Boolean and 2 numerical state variables. A partial evaluation of Boolean variables provides a structure with 4 locations, on which the property cannot be proved. The reason is that, in this example, the "relevant control" essentially comes from numerical thresholds. A further complete refinement according to these thresholds gives a structure with $4 \times 5 \times 2 = 40$ locations (4 Boolean states, 5 relevant intervals for speed, and 2 intervals for count). Our tool starts with an initial structure with 3 locations, and performs 2 splits before proving the property, thus getting a structure with 5 locations. In fact, a manual examination shows that this is the minimal structure allowing the property to be verified.

6 Related works

The problem of the approximation due to the *lub* operation on an abstract lattice L is well-known and a general solution has already been proposed in the literature. A natural solution is to use the disjunctive completion of L [CC92a] which basically consists in manipulating non-redundant unions of elements of L. [Bou92a] proposes refined versions of this solution — with application to the computation of minimal function graphs —, where the number of disjunctions is limited by application of suitable widening operators. It is however not obvious that such solutions would be efficient in our case: decomposing synchronous programs into smaller functions and applying interprocedural analysis leads to too rough results. We have then to consider the whole program as a big function and thus post- and pre-condition computations of abstract states involve themselves many unions, which can produce a combinatorial explosion if no abstract union is applied between two steps of the fixpoint computation. Moreover this solution is proposed in a context where the only goal is to compute good abstractions of concrete elements; computations are not guided by a property to be proved.

[BGP97,BGL99] address the verification of CTL formulas on similar systems. Sets of states are represented by disjunctions of conjunctions of Boolean and numerical domains. Although numerical domains are represented by Presburger arithmetic formulas, which are closed under set union, the separation of the Boolean and the numerical parts leads to the manipulation of explicit unions. This technique requires the partitioning of the state space of the program in such a way that on each partition post- and pre-condition can be separately computed on the Boolean and the numerical parts. This partition is coarser than our finest partition because numerical constraints in numerical terms are handled directly in Presburger arithmetic, but remains still potentially explosive. Here also, the partition does not depend of the considered property.

7 Conclusion and Further Work

This work was motivated by our attempt to apply abstract interpretation techniques to the verification of synchronous data-flow programs; these programs don't provide any control structure to be used for partitioning. Our first idea [Hal98,HPR97] was to consider the control structure implicitly represented by Boolean (or, more generally, finite state) variables. In practice, this solution is not satisfactory: on one hand, for real-size programs, it often produces unmanageable control structures (combinational explosion); on the other hand, the obtained control structure sometimes happens to be too coarse to allow the desired property to be proved. In this paper we propose to take into account the property in the selection of the control structure. An initial, very coarse, control structure is progressively refined, according to the needs of the verification.

Another formulation of our problem is that we want to combine properties on numbers, with properties on Boolean variables. For this, we could have designed a new lattice of abstract values, integrating, e.g., BDDs and polyhedra, like in [Mau96]. Instead, our proposal consists in entrust the control structure with the Boolean part. An advantage is that it allows Boolean properties to be combined, at low cost, with *any* other abstract interpretation.

For us, this work is a first confirmation of the practical usefulness of dynamic partitioning. Of course it must be completed: more experiments should be driven, which will probably suggest some refinements of our heuristics. Also, it should be applied to other interpretations, which may have specific features. For instance, in interval analysis, one can easily evaluate the *distance* between $\gamma(I \sqcup J)$ and $\gamma(I) \cup \gamma(J)$, thus measuring the approximation involved by each *lub* operation. This measure could obviously be used in the refinement heuristics.

References

[BGL99] T. Bultan, R. Gerber, and C. League. Composite model checking: Verification with type-specific symbolic representations. Technical Report TRCS99-02, University of California, Santa Barbara, January 1999.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmeti c. In *Computer Aided Verification, CAV'97*. LNCS 1254, Springer Verlag, June 1997.

[Bou92a] F. Bourdoncle. Abstract interpretation by dynamic partitionning. *Journal of Functional Programming*, 2(4), 1992.

[Bou92b] F. Bourdoncle. Sémantique des langages impératifs d'ordre supérieur et interprétation abstraite. Thesis Ecole Polytechnique, Paris, 1992.

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

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages, POPL'77*, Los Angeles, January 1977.

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

[CC92b] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *PLILP'92*, Leuven (Belgium), January 1992. LNCS 631, Springer Verlag.

[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.

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

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

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

[Mau96] Ch. Mauras. Symbolic simulation of interpreted automata. In *3rd Workshop on Synchronous Programming*, Dagstuhl (Germany), December 1996.