About synchronous programming and abstract interpretation*
(Invited Paper)

Nicolas Halbwachs

Verimag** – Miniparc-Zirst
F–38330, Montbonnot, France
e-mail: Nicolas.Halbwachs@imag.fr

1 Introduction

Synchronous programming has been proposed during the last decade, as a new point of view on concurrency. A family of synchronous languages [IEE91, Hal93b] (e.g., ESTEREL, STATECHARTS, ARGOS, LUSTRE, SIGNAL) has been based on this point of view, and devoted to the design of *reactive systems*, i.e., computer systems whose role is to maintain a continuous, real-time, interaction with their environment.

In the synchronous approach, concurrency is not considered as an implementation constraint, but as a programming facility for logical structuring of programs. Since synchronous concurrency has nothing to do with parallel or distributed execution, it can be defined without taking into account physical constraints, like communication or task execution delays. A synchronous process is supposed to be able to react *instantaneously* to incoming events, and to communicate *instantaneously* with other processes. The main consequence is that a synchronous system can be (and will be, in most synchronous languages) *deterministic*, a feature which is highly desirable in reactive programs, and which tremendously simplifies the design, debugging, and verification tasks.

An abundant literature has been devoted to synchronous languages (e.g., [BCG88, BG92, Ber89, BS91, Har87, Mar92, CPHP87, HCRP91, LGLL91, BL90, GGB87, BCGH93]). This paper intends to highlight the connection between synchronous programming and abstract interpretation. First, one of the compiling techniques which have been developed for synchronous languages consists of synthesizing the control structure of the sequential object code as a finite automaton. We will see that this technique is indeed an abstract interpretation. The second point concerns program verification: For reactive systems, experience shows that an important practical goal is to ensure some critical properties, rather than to completely prove program correctness. Moreover, these properties are mainly *safety* properties. Now, the synchronous approach allows

* This work has been partly supported ESPRIT-BRA action "REACT" and by a grant from the French Department of Research.

** Verimag is a joint laboratory of CNRS, Institut National Polytechnique de Grenoble, Université Joseph Fourier and Verilog SA associated with IMAG.

an easy translation of any safety property into an *invariant*: The property is expressed by means of an auxiliary program, called *synchronous observer* [HLR93], which observes the input/output events of the program under verification, and detects any violation of the property. The verification then consists in showing that the parallel composition of the program and its observer never complains about property violation. Finally, invariants can be proved by means of approximate reachability analysis, which is probably the most studied application of abstract interpretation [CC77, CC92]. Numerous specific analysis techniques (e.g., [Kar76, Weg75, CC76, CH78, Gra89, Mas93]) are available, which provide conservative verification tools in this context.

The paper essentially consists of two parts: Section 2 is a brief introduction to synchronous languages, illustrated by aspects of the language LUSTRE. Section 3 concerns the use of abstract interpretation in compiling and verification.

2 Synchronous programming

or *Concurrency can make programming easier*

The subtitle of this section is deliberately provocative: From the very first steps of time-sharing systems, concurrency has been identified as a tremendous increase in programming complexity, because of non-determinism and lack of global notion of *state*. However, after Milner's pioneering works [Mil81, Mil83], the synchronous model was developed, which overcomes these two problems at the expense of losing the idea of concurrent implementation. In synchronous languages, concurrency must be understood as a way of describing the desired behavior of a system, and has nothing to do with concurrent execution.

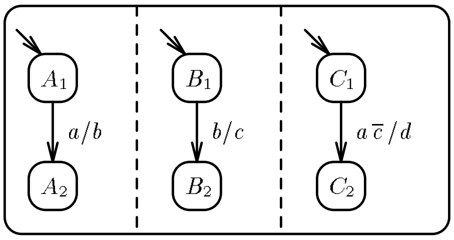
2.1 Basic principles

Basically, a reactive system can be viewed as an input/output automaton, reacting to incoming events by selecting a transition, computing and emitting output events, and changing its state for the next reaction. As a matter of fact, many reactive systems are implemented in that way (for instance on automatic controllers). However, building an automaton by hand is a complex and error-prone task. Automata are even harder to debug and modify, since the slightest modification in the specification may involve a complete change in the structure of the automaton. These problems are mainly due to the lack of structure and modularity. Synchronous languages are nothing but high level, modular, languages to describe automata, exactly as LEX and YACC are languages to describe scanners and parsers.

The main structuring feature, which exists in all the synchronous languages, is synchronous parallel composition. Intuitively, when two automata are compound in parallel, each of them perceives the outputs of the other as its own inputs. A transition of the whole program, in response to an input event, consists of simultaneous transitions of all active parallel processes in the program, each process reacting to the current event, which is the input event enriched by the outputs of the other processes. So, the current event is the solution of a fixpoint equation

"*current_event* = *input_event* **enriched_by response_to**(*current_event*)"

For instance, we show beside a fragment of an ARGOS program [Mar92] which is the parallel composition of three automata:The first one, in state A_1, waits for an input event where the "signal" a is present. Whenever such an event occurs, it executes its transition to state A_2 and emits the signal b.



Now, the presence of b in the current event involves the reaction $B_1 \longrightarrow B_2$ of the second automaton, which emits the signal c. The third automaton does not react, since, in C_1 it waits for an event containing a but not c. So, in presence of the input event $\{a\}$, the resulting current event is $\{a, b, c\}$ and the whole program moves from the global state (A_1, B_1, C_1) to the global state (A_2, B_2, C_1).

2.2 An example: programming in Lustre

We will use the synchronous data-flow language Lustre, as an example of synchronous language. As for any synchronous program, a behavior of a Lustre program is an infinite sequence of reactions to input events. The data-flow point of view is the following: at each reaction, each variable of the program has a value, so we associate with each variable x an infinite sequence $(x_1, x_2, \ldots, x_n, \ldots)$, where x_n is the value of x at the nth reaction of the program. Such a sequence will be called a *flow*, and a program can be viewed as computing output flows (i.e., flows associated with output variables) from input flows. In fact, any operator in the language can be viewed as such a function on flows: for instance if x and y are variables, respectively associated with flows $(x_1, \ldots, x_n, \ldots)$ and $(y_1, \ldots, y_n, \ldots)$, then $x+y$ denotes the flow $(x_1+y_1, \ldots, x_n+y_n, \ldots)$; that is, "+" is an operator on flows, which returns the pointwise sum of its operands. Usual arithmetic, Boolean, conditional, operators are extended this way to operate on flows; they are called *data operators*. Constants are considered as constant flows. Two specific operators will be used: the "previous" (pre) and the "followed-by" (->) operators. Let $x = (x_1, \ldots, x_n, \ldots)$ and $y = (y_1, \ldots, y_n, \ldots)$ be two flows, then $\mathsf{pre}(x) = (nil, x_1, \ldots, x_{n-1}, \ldots)$ and x->$y = (x_1, y_2, \ldots, y_n, \ldots)$. Now, a program basically consists of a system of equations defining output (and possibly local) variables, as expressions on constant and variables. More precisely, such a system of equations is encapsulated into a *node*, which is a user-defined operator on flows. Let us start with two simple examples which will be used later on.

- The first one is a "switch", which receives two Boolean flows, ON and OFF, and returns a Boolean flow S. Whenever ON is true, S becomes true; otherwise, if OFF is true, S becomes false; when neither ON nor OFF is true, the value of S does not change (its initial value is false):

```
node SWITCH (ON, OFF: bool) returns (S: bool);
let
    S =    if ON then true
           else if OFF then false
           else (false -> pre(S));
tel
```

Once such a node is declared, it can be used as an operator, anywhere in a program.

– The second example is a counter-uncounter, whose result is always the difference between the number of times the input INCR has been true, and the number of times the input DECR has been true:

```
node DIFF (INCR, DECR: bool) returns (DIFF: int);
let
    DIFF = (0 ->pre(DIFF)) +
              (if INCR and not DECR then 1
              else if not INCR and DECR then -1
              else 0);
tel
```

Now, let us consider a (slightly) more realistic example, extracted from a subway traffic regulation system: a train detects beacons that are placed along the track, and receives the "second" from a central clock. Ideally, it should encounter one beacon each second; so the space left between beacons rules the speed of the train. Here, we only consider the way the train detects it is early or late. To avoid shaking, a hysteresis is applied: whenever the number $\#b$ of beacons encountered exceeds the number $\#s$ of seconds received plus 10, the train detects it is early, and remains early until $\#b$ becomes equal to $\#s$. Conversely, whenever $\#b < \#s - 10$, the train detects it is late until $\#b = \#s$.

The program makes use of the nodes defined above. We use a local variable advance which counts the difference $(\#b - \#s)$ by means of the node DIFF. The thresholds on advance are used to control two switches, which respectively compute the "early" and "late" states.

```
node train (beacon, second: bool) returns (early, late: bool);
var advance: int;
let
    advance = DIFF(beacon, second);
    early = SWITCH(advance>10, advance=0);
    late = SWITCH(advance<-10, advance=0);
tel
```

Notice that the three equations can be written in any order, as in mathematics. They can also be considered as three parallel processes, synchronously composed: at each reaction, the process computing advance instantaneously computes its result and broadcasts it to the other processes.

2.3 Natural semantics of Lustre

Let us give (a simplified version of) the semantics of Lustre, in structural operational semantics [Plo81], in order to show that a Lustre program behaves as an automaton. In particular, we have to identify the notion of *state* of a program. Let *Var* ($\ni x$) be the set of variable identifiers appearing in a program. A *memory* is a function σ from *Var* to the set *Val* ($\ni v$) of values[3]. Let σ_{In} and σ_{Out} respectively denote the restriction of a memory σ to input and output variables. Basically, a reaction of a program computes an output memory σ_{Out} from an input memory σ_{In}; but, for doing so, it needs also some information about the past execution, to compute the results of the operators "pre" and "->". Without loss of generality, let us assume that any "pre" operator in a program, is applied to a single variable. Then, the value of "pre(x)" at a given reaction is the value of x at the previous reaction, i.e., in the previous memory. Thus, we have to keep track of the previous memory (noted π) to compute the current reaction (initially, the previous memory associates the undefined value *nil* with each identifier). Moreover, the result of "e_1 ->e_2" is the result of e_1 if the reaction is the first one, otherwise it is the result of e_2. So, we also need a Boolean flag, say *init*, which is true only at the first reaction of the program. For simplicity, assume *init* is an auxiliary variable, whose value is given by the memory. This value is true only in the initial memory. Now, the past memory π constitutes the state of the program. So, a reaction of the program will be noted $\pi \xrightarrow[\sigma_{Out}]{\sigma_{In}} \sigma$ to express that, in the state π the reaction to the input memory σ_{In} computes the whole memory σ, whose projection on outputs is σ_{Out}. A behavior of the program will be a sequence $(\langle\sigma^1_{In}, \sigma^1_{Out}\rangle, \langle\sigma^2_{In}, \sigma^2_{Out}\rangle, \ldots)$, where the sequence of memories $(\sigma^1, \sigma^2, \ldots)$ satisfies

$$\sigma^0 \xrightarrow[\sigma^1_{Out}]{\sigma^1_{In}} \sigma^1 \text{ and } \sigma^n \xrightarrow[\sigma^{n+1}_{Out}]{\sigma^{n+1}_{In}} \sigma^{n+1} \quad \forall n \geq 1$$
$$\text{where } \sigma^0(x) = nil, \forall x \neq init \text{ and } \sigma^0(init) = true$$

We start from a simplified abstract syntax:

$$\begin{aligned}
\mathsf{program} &::= \underline{\mathsf{node}}\ (\mathsf{In})(\mathsf{Out})\ \mathsf{equations} \\
\mathsf{equations} &::= \mathsf{equation} \ \mid \ \mathsf{equations}\ ;\ \mathsf{equation} \\
\mathsf{equation} &::= \mathsf{x} = \mathsf{exp} \\
\mathsf{exp} &::= \mathsf{x} \ \mid \ \mathsf{k} \ \mid \ \mathsf{op}(\mathsf{exp}_1, \ldots, \mathsf{exp}_n) \ \mid \ \underline{\mathsf{pre}}(\mathsf{x}) \ \mid \ \mathsf{exp}_1\ \texttt{->}\mathsf{exp}_2
\end{aligned}$$

where In and Out are lists of input and output identifiers, x stands for any identifier, k for any constant, and op for any n-ary data operator.

Fig. 1 gives the semantic rules, defining the following predicates:

program : $\pi \xrightarrow[\sigma_{Out}]{\sigma_{In}} \sigma$, which expresses that $\pi \xrightarrow[\sigma_{Out}]{\sigma_{In}} \sigma$ is a reaction of program.

[3] We do not detail the type rules, which are standard.

$$\frac{\mathsf{equations} \,:\, \pi \longrightarrow \sigma}{\underline{\mathsf{node}}\ (\mathsf{In})(\mathsf{Out})\ \mathsf{equations} \,:\, \pi \xrightarrow[\sigma_{Out}]{\sigma_{In}} \sigma}$$

$$\frac{\mathsf{equations} \,:\, \pi \longrightarrow \sigma \quad , \quad \mathsf{equation} \,:\, \pi \longrightarrow \sigma}{\mathsf{equations};\mathsf{equation} \,:\, \pi \longrightarrow \sigma} \qquad \frac{\mathsf{exp} \,:\, (\pi, \sigma) \rightarrow \sigma(x)}{\mathsf{x} = \mathsf{exp} \,:\, \pi \longrightarrow \sigma}$$

$$\mathsf{x} \,:\, (\pi, \sigma) \rightarrow \sigma(x) \qquad \mathsf{k} \,:\, (\pi, \sigma) \rightarrow k \qquad \frac{\mathsf{exp}_i \,:\, (\pi, \sigma) \rightarrow v_i \ ,\ i = 1..n}{\mathsf{op}(\mathsf{exp}_1, \ldots, \mathsf{exp}_n) \,:\, (\pi, \sigma) \rightarrow op(v_1, \ldots, v_n)}$$

$$\underline{\mathsf{pre}}(\mathsf{x}) \,:\, (\pi, \sigma) \rightarrow \pi(x) \qquad init \,:\, (\pi, \sigma) \rightarrow false$$

$$\frac{\pi(init) = true \quad , \quad \mathsf{exp}_1 \,:\, (\pi, \sigma) \rightarrow v_1}{\mathsf{exp}_1\text{->}\mathsf{exp}_2 \,:\, (\pi, \sigma) \rightarrow v_1} \qquad \frac{\pi(init) = false \quad , \quad \mathsf{exp}_2 \,:\, (\pi, \sigma) \rightarrow v_2}{\mathsf{exp}_1\text{->}\mathsf{exp}_2 \,:\, (\pi, \sigma) \rightarrow v_2}$$

Fig. 1. Semantic rules

equations $: \pi \longrightarrow \sigma$, which expresses that σ is a current memory consistent with the evaluation of equations in the state π.

exp $: (\pi, \sigma) \rightarrow v$, which expresses that in the state π, v is the value of the expression exp, evaluated in the current memory σ.

Lustre static semantics (which is not described here) ensures that these rules define a deterministic semantics: in each state π, the current memory σ is a *function* of the input memory σ_{In} (and thus, so are the current outputs and the next state).

3 Compilation and verification

or *Why synchronous programming and abstract interpretation nicely fit together*

3.1 Control structure synthesis

In this section, we consider the problem of compiling synchronous languages. The code of a reactive system is generally subject to dramatic constraints, concerning either the reaction time (real-time systems), or the size of the code (embedded systems), or both. So, a wide range of compiling techniques is needed, so as to adjust the size and performances of the code to the requirements.

In the preceding section, we have seen that a synchronous program behaves as a deterministic automaton. There are two extreme ways for implementing such an automaton:

- It can be implemented as a single function, taking the current state and input as parameters, and returning the current output and the next state. In this solution, the code executed at each reaction is the same. The structure of the object code is a single infinite loop, whose body consists of 3 steps: (1) waiting for inputs, (2) computing outputs, (3) computing and storing

```
init := true;
loop
    get(ON,OFF);
    if init then
        S := if ON then true else false;
        init:=false;
    else
        S := if ON then true elsif OFF then false else pre_S
    endif
    emit(S); pre_S := S;
endloop
```

Fig. 2. Single loop code of the node SWITCH

<table>
<tr><th>State_1
init</th><th>State_2
not init and pre_S</th><th>State_3
not init and not pre_S</th></tr>
<tr>
<td>get(ON,OFF);
if ON then
    S := true;
    goto State_2
else
    S := false;
    goto State_3</td>
<td>get(ON,OFF);
if ON then
    S := true;
    goto State_2
elsif OFF then
    S := false;
    goto State_3
else
    S := true;
    goto State_2</td>
<td>get(ON,OFF);
if ON then
    S := true;
    goto State_2
else
    S := false;
    goto State_3</td>
</tr>
</table>

Fig. 3. Detailed automaton code of the node SWITCH

the memory for the next step. As a very simple example, consider the node SWITCH presented in §2.2. The single loop code could have the structure shown in Fig. 2.

- Another solution is to produce a different function for each state that can be reached by the automaton (provided this set of states is finite). The current input is the only parameter of each function, which returns the current output and selects the function to be applied at the next reaction. For the SWITCH example, we get 3 reachable states, so 3 functions[4] as shown in Fig. 3.

[4] Notice that the code executed in states 1 and 3 is the same, so the automaton has actually only 2 states. Algorithms to compile synchronous programs into *minimal* automata are available [BFH+92].

The latter solution produces a more efficient code, but can only be applied when the set of reachable states is finite. Moreover, even in the finite case, the number of states (and thus, the size of the code) grows exponentially with the number of variables. Intermediate solutions can be defined as follows: Let $\mathcal{S}$ be a set of *state variables*, with finite domains (generally Boolean), selected by the user.

Let $\sigma_{\mathcal{S}}$ be the restriction of the memory σ to $\mathcal{S}$, and $[\sigma]_{\mathcal{S}}$ be the equivalence class of σ for the relation $\quad \sigma \overset{\mathcal{S}}{\approx} \sigma' \iff \sigma_{\mathcal{S}} = \sigma'_{\mathcal{S}}$.

The idea is to distinguish the code to be executed according to the value of state variables — i.e., to the class $[\pi]_{\mathcal{S}}$ of the previous memory —, while the other variables are stored in the actual memory. Let $\mathcal{D} = \mathit{Var} \setminus \mathcal{S}$. The code corresponding to a class $[\pi]_{\mathcal{S}}$ is a function of $(\pi_{\mathcal{D}}, \sigma_{In})$ returning the output σ_{Out}, the next class $[\sigma]_{\mathcal{S}}$ and the memory $\sigma_{\mathcal{D}}$. Ideally, this code should be generated only for reachable classes, and we would have a transition $[\pi]_{\mathcal{S}} \longrightarrow [\sigma]_{\mathcal{S}}$ if and only if

$$\exists \pi' \overset{\mathcal{S}}{\approx} \pi,\ \exists \sigma' \overset{\mathcal{S}}{\approx} \sigma,\ \text{with } \pi' \xrightarrow[\sigma'_{Out}]{\sigma'_{In}} \sigma'$$

But, in general, the reachability of a class cannot be determined statically, without taking fully into account the behavior of variables in $\mathcal{D}$. So, the reachability is only approached, and some unreachable states can be dealt with. The construction of the control automaton makes use of an abstract interpretation of the program, by abstracting a memory π into

$$\alpha(\pi) = \lambda x. \begin{cases} \pi(x) & \text{if } x \in \mathcal{S} \\ \top & \text{if } x \in \mathcal{D} \end{cases}$$

where $\top$ is a non-determined value, with respect to which most operators are strict.

For instance, Fig. 4 shows the resulting code for the program "train" when the state variables are all the Boolean variables. The control automaton has 5 states: the initial state (State_0), the states where the train is on time (State_4), early (State_2), late (State_3), and a clearly unreachable state where it is both early and late (State_1). Also, many transitions are clearly irrelevant. In this example, state variables highly depend on numerical variables, so the structure of the resulting control automaton is rather poor. Notice that all the lines marked with a "$*$" would be removed by a straightforward numerical analysis of path condition satisfiability.

3.2 Program verification

Synchronous programming and abstract interpretation nicely cooperate for reactive system verification. This is due to 3 reasons, that we will develop in turn:

1. In the field of reactive systems, the main goal, in practice, is to prove some critical *safety* properties;

State_0: init

```
  if beacon then
      if second then advance := 0
      else advance := 1
  elsif second then advance := −1
  else advance := 0;
  if advance < −10 then late := true;
*     if advance > 10 then
*         early := true; goto State_1
      else early := false; goto State_3
  else late := false
      if advance > 10 then
          early := true; goto State_2
      else early := false; goto State_4
```

State_1: pre_early and pre_late

```
  if beacon then
      if not second then advance++
  elsif second then advance-- ;
  if advance = 0 then
*     if advance < −10 then
*         if advance > 10 then goto State_1
*         else early := false; goto State_3
      else late := false
*         if advance > 10 then goto State_2
          else early := false; goto State_4
  else goto State_1
```

State_2: pre_early and not pre_late

```
  if beacon then
      if not second then advance++
  elsif second then advance-- ;
  if advance < −10 then late := true;
*     if advance > 10 then goto State_1
*     elsif advance = 0 then
*         early := false; goto State_3
      else early := true; goto State_1
  elsif advance > 10 then goto State_2
  elsif advance = 0 then
          early := false; goto State_4
  else goto State_2
```

State_3: not pre_early and pre_late

```
  if beacon then
      if not second then advance++
  elsif second then advance-- ;
  if advance = 0 then
*     if advance < −10 then
*         if advance > 10 then
*             early := true; goto State_1
*         else goto State_3
      else late := false;
*         if advance > 10 then
*             early := true; goto State_2
          else goto State_4
  elsif advance > 10 then
      early := true; goto State_1
  else goto State_3
```

State_4: not pre_early and not pre_late

```
  if beacon then
      if not second then advance++
  elsif second then advance-- ;
  if advance < −10 then late := true;
*     if advance > 10 then
*         early := true; goto State_1
      else goto State_3
  elsif advance > 10 then
      early := true; goto State_2
  else goto State_4
```

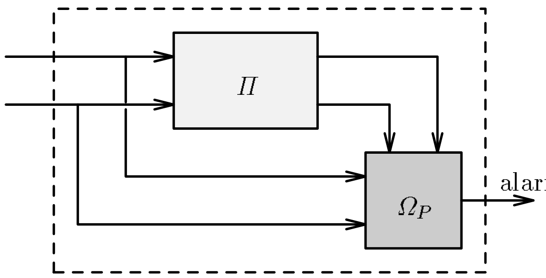
Fig. 4. Control structure of the “train” example

2. In addition to being well-suited to reactive system programming, synchronous programming allows an easy and modular expression of safety properties in terms of *invariants*;
3. Abstract interpretation allows an approximate, conservative, verification of invariance properties.

Safety properties of reactive systems: It is a commonplace to say that reactive systems are those whose reliability is the most critical, because of the dramatic consequences that failures can have in this field. However, experience shows that, in practice, the most important goal is to ensure some critical properties, rather than to completely prove program correctness. Moreover, it was noted elsewhere [HLR92, Pnu92] that, almost always, these properties are *safety properties*, i.e., properties expressing that something bad never happens. As a matter of fact, in non-reactive systems, *liveness properties* often result from the abstraction of a real-time property; in reactive systems, real-time properties cannot be abstracted. For instance, nobody cares that a train *eventually* stops; it must stop within a given delay or distance.

Translating safety into invariance: Let us recall that, if a program behavior violates a safety property, it does so at a precise step: one can identify the first reaction which violates the property.

As a consequence, given a safety property P about the behavior of a program Π, one can write another program Ω_P — called an *observer* of P —, which takes as input the input/output variables of Π, and emits an "alarm" whenever P is violated (see the figure beside).



Now, instead of proving P about Π, we can prove that the parallel composition of Π and Ω_P never emits an alarm. In other words, the safety property P has been changed into an invariant. Notice that the synchronous composition mechanism allows this translation to be *modular*. In an asynchronous model, in order to ensure that the observer accurately perceives the behavior of Π, one would have to add explicit synchronizations in Π: it would not be modular, and, worse, it could modify the behavior of Π, thus invalidating the result of the proof.

For instance, consider the property of our program "train", stating that it cannot move from a state where it is "early" to a state where it is "late" without passing in a state where it is "on time". An observer of this property will take as input the variables early and late computed by the program train, and compute a variable alarm as follows: alarm is true whenever late is true and on_time has never been true since the last time early was true:

```
node observer (early, late: bool) returns (alarm: bool);
var ontime, never_ontime_since_early: bool;
let
    alarm = late and never_ontime_since_early;
    never_ontime_since_early = SWITCH(early, ontime);
    ontime = not (late or early);
tel
```

The verification of the property comes down to proving that the following program never returns a true value:

```
node verify (beacon, second: bool) returns (alarm: bool);
var early, late: bool;
let
    (early, late) = train(beacon, second);
    alarm = observer(early, late);
tel
```

Approximate verification of invariants: Let Π be a program, considered as a transition system $(S, \mathcal{I}, \rightarrow)$ (where S is a set of states, $\mathcal{I} \subseteq S$ is the set of initial states, and $\rightarrow \subseteq S \times S$ is the transition relation). Let *pre* and *post* be the classical *precondition* and *postcondition* functions, from 2^S to 2^S:

$$pre = \lambda X.\{s \in S \mid \exists s' \in X,\ s \rightarrow s'\}\ ,\ post = \lambda X.\{s' \in S \mid \exists s \in X,\ s \rightarrow s'\}$$

If X is a set of states, let $\mathcal{R}(X)$ be the set of states which are reachable from X, and $\mathcal{A}(X)$ be the set of states from which X is reachable. We have:

$$\mathcal{R}(X) = \mu Y.X \cup post(Y) \quad , \quad \mathcal{A}(X) = \mu Y.X \cup pre(Y)$$

To prove that a given set $\mathcal{P}$ of states is an invariant of a program, we can show that the set $\mathcal{R}(\mathcal{I})$ of reachable states of the program is included in $\mathcal{P}$ (forward verification) or, equivalently, that the set $\mathcal{I}$ of initial states of the program does not intersects the set $\mathcal{A}(\neg\mathcal{P})$ of states which can lead outside of $\mathcal{P}$ (backward verification). In general, neither $\mathcal{R}(\mathcal{I})$ nor $\mathcal{A}(\neg\mathcal{P})$ can be automatically computed. However, both are least fixpoints that can be upper-approximated using available abstract interpretations [Kar76, Weg75, CC76, CH78, Gra89, Mas93]. Let $\widehat{\mathcal{R}(\mathcal{I})}$ and $\widehat{\mathcal{A}(\neg\mathcal{P})}$ be upper approximations of $\mathcal{R}(\mathcal{I})$ and $\mathcal{A}(\neg\mathcal{P})$, respectively; then the invariance of $\mathcal{P}$ is ensured as soon as either $\widehat{\mathcal{R}(\mathcal{I})} \subseteq \mathcal{P}$ or $\mathcal{I} \cap \widehat{\mathcal{A}(\neg\mathcal{P})} = \emptyset$. Obviously, these conditions are only necessary conditions, so the verification method is only partial, and nothing can be inferred when it fails. However the results can be strengthened by combining both approaches: For instance, when $\widehat{\mathcal{R}(\mathcal{I})}$ intersects $\neg\mathcal{P}$, one can compute an upper approximation of $\mathcal{A}(\widehat{\mathcal{R}(\mathcal{I})} \cap \neg\mathcal{P})$, and show that it does not intersect $\mathcal{I}$, and so on.

Several authors [HLR92, BS91] simply use the control automaton built by synchronous language compilers, to prove logical properties of programs. This approach fails in proving our example property (§3.2), because it strongly depends on the behavior of numerical variables.

Here, we show a more powerful technique [Hal93a], combining the control automaton with the abstract interpretation proposed in [CH78]: In this interpretation, a set of numerical states (vectors) is approximated by its *convex hull*, i.e., the least convex polyhedron containing it. With each state s of the control automaton is associated a polyhedron P_s, such that (with the notations of §3.1),

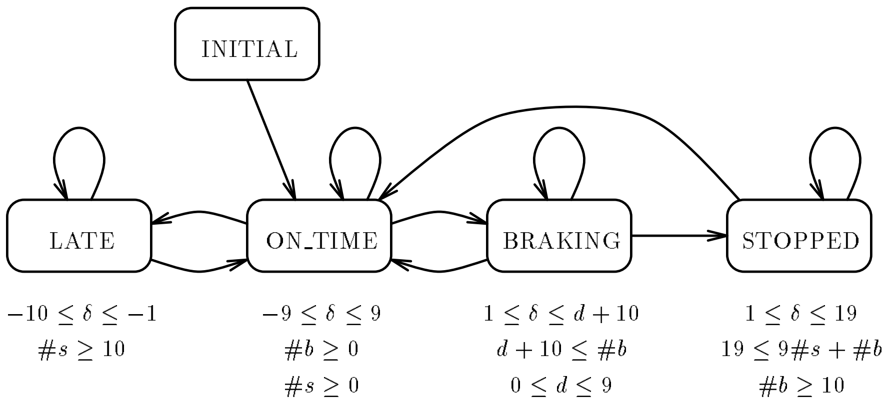
$$\text{for each reachable memory } \pi,\ \pi_{\mathcal{S}} = s \ \Rightarrow\ \pi_{\mathcal{D}} \in P_s$$

So, if $P_s = \emptyset$, the state s is not reachable. Without explaining the technique in more detail (see [Hal93a]), we show the results on the complete subway example.

The complete subway example: Let us come back to our subway traffic regulation system. We assume that

- when a train is early, it puts on brake. Continuously braking stops the train before encountering 10 beacons.
- the "second" signal is broadcast from a central clock. A late train signals it to the central clock, which does not emit the "second" as long as at least one train is late.

From a Lustre program simulating the whole system, the compiler generates a control automaton with 9 states. The approximate convex analysis shows that only 5 states are reachable, and gives the following results (where $\#b$ and $\#s$ are the numbers of beacons and of seconds, and where δ stands for $\#b - \#s$):



The property expressed in §3.2 is obvious on this automaton. Moreover, the absolute difference $|\#b - \#s|$ is shown to be bounded. With two trains, the analysis shows that the absolute difference $|\#b_1 - \#b_2|$ between the numbers of beacons encountered by each train is also bounded, which means that if the number of beacons initially separating the trains is greater than the bound, then the trains cannot collide.

The whole approach can be viewed as a compound abstract interpretation: First, the control structure is generated, which takes into account the behavior of (some) Boolean variables, and provides a finite partition to the numerical analysis. The more detailed is the partition, the more precise is the numerical analysis.

References

[BCG88] G. Berry, P. Couronné, and G. Gonthier. Synchronous programming of reactive systems, an introduction to ESTEREL. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*. Elsevier Science Publisher B.V. (North Holland), 1988. INRIA Report 647.

[BCGH93] A. Benveniste, P. Caspi, P. Le Guernic, and N. Halbwachs. Data-flow synchronous languages. In REX *Symposium "Ten Years of Concurrency, Reflections and Perspectives"*. LNCS 803, Springer-Verlag, June 1993.

[Ber89] G. Berry. Real time programming: Special purpose or general purpose languages. In *IFIP World Computer Congress*, San Francisco, 1989.

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

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BL90] A. Benveniste and P. LeGuernic. Hybrid dynamical systems theory and the SIGNAL language. *IEEE Transactions on Automatic Control*, 35(5):535–546, May 1990.

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

[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*, Los Angeles, January 1977.

[CC92] 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).

[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.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, Munchen, January 1987.

[GGB87] T. Gauthier, P. Le Guernic, and L. Besnard. Signal, a declarative language for synchronous programming of real-time systems. In *Proc. 3rd. Conf. on Functional Programming Languages and Computer Architecture*. LNCS 274, Springer Verlag, 1987.

[Gra89] P. Granger. Static analysis of arithmetical congruences. *International Journal on Computer Mathematics*, 30:165–190, 1989.

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

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

[Har87] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8(3), 1987.

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

[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.

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

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

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

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

[Mas93] F. Masdupuy. Semantic analysis of interval congruences. In *Int. Conf. on Formal Methods in Programming and their Applications.* LNCS, Springer Verlag, July 1993.

[Mil81] R. Milner. On relating synchrony and asynchrony. Technical Report CSR-75-80, Computer Science Dept., Edimburgh Univ., 1981.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *TCS*, 25(3), July 1983.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Lecture notes, Aarhus University, 1981.

[Pnu92] A. Pnueli. How vital is liveness? Verifying timing properties of reactive and hybrid systems. In *CONCUR'92, Stony Brook.* LNCS 630, Springer Verlag, August 1992.

[Weg75] B. Wegbreit. Property extraction in well-founded property sets. *IEEE Trans. on Software Engineering*, SE-1(3):270–285, September 1975.