

# Temporal Abstract Domains

Julien Bertrane  
 Département d'Informatique  
 École Normale Supérieure  
 Paris, France  
 bertrane@di.ens.fr

**Abstract**—The specifications of the control units driving embedded systems often involve temporal properties. We aim at certifying them statically using the Abstract Interpretation framework and introduce several Abstract Domains dedicated to proving such temporal properties. This work defines the specificity of such domains, that we call Temporal Abstract Domains.

We introduce a continuous-time abstraction, since this abstract and continuous representation of the time allows a fast computation of abstract invariants that are furthermore more precise than with discrete time. This also enables the definition of a canonical reduced-product between the domains. We finally present new abstract domain transformers that build more precise new domains with a reasonable additional cost with respect to the initial domain. An example of such a generic transformer introduces temporally-local disjunctions and is thus specific to temporal abstract domains.

**Keywords**-Static Analysis; Abstract Interpretation; Embedded systems; Temporal specifications; Reduced Product.

## I. INTRODUCTION

The control units driving embedded systems often have temporal specifications. This is because reactive systems do not compute a *final result* like classical programs. On the contrary, the computation is infinite and returns updated results repeatedly. The time needed for any part of this computation is therefore of huge importance. For example, it can be very useful to specify that an alarm has to raise at most  $k$  seconds after such or such failure. The time is thus intrinsically related to the abstract properties that we try to prove. In Temporal Abstract Domains, we make sure that it is treated as a special information, and not just as a regular variable as it is often the case in non-temporal analysis.

The control units are frequently driven by computers programmed using synchronous languages, which usually have a discrete semantics. The static analysis aiming at proving these specifications have thus often been performed in discrete frameworks. As a consequence, clocks desynchronization problems as well as communication delays between units, which are usually asynchronous, are often ignored during the certification pro-

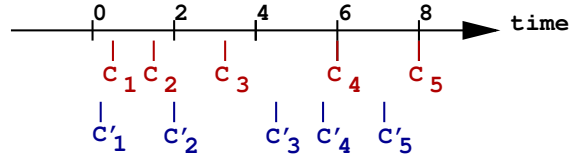


Figure 1: Two clocks  $C$  and  $C'$  with the same period interval as parameter

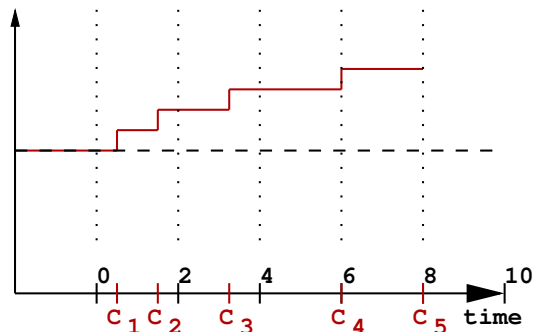


Figure 2: One signal, clocked by one imprecise clock  $C$  of period interval  $[1.6; 2.4]$ , *i.e.*  $2, \pm 2\%$

cess. Alternatively, these hardware-related imperfections are encoded in the synchronous framework. However, certification then relies on a big case analysis that cannot be comprehensively analyzed. In our framework, we allow clocks to desynchronize, as long as their period remains inside the *period interval*, which should be given as a parameter by the people designing the hardware. For example, the clocks  $C$  and  $C'$  depicted in Fig. 1 satisfy the parameter  $[1.6; 2.4]$ . However, infinitely many other clocks also have the same parameter, including a perfectly synchronous clock that ticks every 2 time units.

The continuous-time semantics doesn't only allow the expression of the hardware imperfections and of specifications aiming at being certified. It also enables a precise and inexpensive abstraction. This may be because these systems were in fact designed in a continuous world (differential equations) in order to remain as close as possible to the environment, space and time, that are

continuous objects. In addition, using a continuous-time semantics enables the use of existing mathematical theories about continuous numbers rarely used in static analysis.

We use now the semantics formally defined with more details in [5]. This semantics focuses on the temporal aspects of the computation and its main non-standard point is that it considers time as a continuous notion. Our semantics is indeed not based on a *program counter* that updates the state of the systems. It is not either based on sequences of values like the classical *Lustre* semantics. We rather give a value to each variable of the system at each instant. In the following, we call *signal* such a function connecting each time instant to a value. For example, in Fig. 2, we depict a signal clocked by an imprecise clock.

In synchronous languages, programs are executed according to a clock, performing computation at each cycle of this clock. In real systems, these clocks may be a little imprecise. An example of a simple counter is defined in synchronous languages by the equation  $n = 0 - > (1 + pre\ n)$ <sup>1</sup>. When given an imperfect clock, the semantics is the behavior that consist in, at each cycle of this clock, performing two discrete actions:

- 1) check if the current cycle is the first.
- 2) • if it is the first cycle, then return 0
  - else return previous value of  $n$  incremented by 1

Our semantics can render these imprecisions and in Fig. 3, we show two behaviors for the two imprecise clocks  $\mathcal{C}$  and  $\mathcal{C}'$  presented in Fig.1, whose periods are *approximately* 2 time units, that belong to the semantics of this counter  $n$ . The semantics is continuous-time: it gives values at any time, but it also gives a value to each point of the program. In this case, there are four points of interest in the syntax:  $n$ ,  $pre\ n$ ,  $1 + pre\ n$ , and  $0 - > (1 + pre\ n)$ . Each of them is then connected in each behavior of the system to one signal. In this example, the two behaviors, *i.e.* the two coherent sets of signals, differ a lot following the imprecisions of the clocks.

When a system is made of several communicating sub-systems and if moreover its clocks are allowed to be imprecise, this kind of imprecision does not only induce an eventual delay, but may lead in completely a different behavior for the global system. The number of possible behaviors is then too large to be checked y hand. Our semantics emphasize on *similarities between* behaviors, that may potentially be abstracted by a single abstract element. For example, in Fig. 3, the two behaviors are very close between times 6 and 8, but differ a lot between times 2 and 4. A discrete semantics may not make this difference, and would simply consider them different.

<sup>1</sup> $a - > b$  means  $a$  at first cycle, then  $b$ .  $pre\ c$  means  $c$ , but delayed by one cycle

- We consider the *Lustre* node  $n = 0 - > (1 + pre\ n)$
- with  $[1.6; 2.4]$  as parameter for its clock.
- It has among others this two behaviors in its semantics:

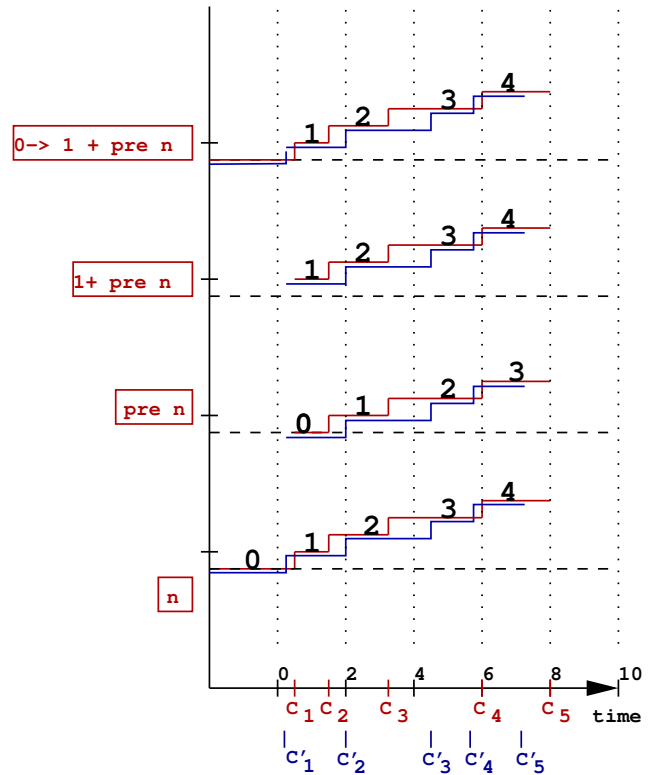


Figure 3: A single imperfectly-clocked synchronous system has many possible behaviors. Two of them are depicted here, depending on their clocks  $\mathcal{C}$  and  $\mathcal{C}'$ .

Continuous-time semantics does not only eases finding similarities. It also allows the definition of abstract operators that reason automatically on these similarities, forgetting temporary dissimilarities, and that prove global properties of all the behaviors in a fast way.

Fortunately, the designers try to design robust systems for which the global behaviors remain similar whatever happens. Proving this robustness of designed systems despite hardware imprecisions is our main goal in the following. The main contributions of this paper thus are:

- We first introduce (Sect. II) a formalization of a subclass of abstract domains with additional characteristics and capacities, the temporal abstract domains. The temporal domain of abstract constraints is presented as an example of a temporal domain.
- We then define (Sect. III) the temporally-local disjunction, an automatic abstract transformer for temporal abstract domains that enables disjunctions of elements and thus defines automatically similar yet more precise domains. The fact that the

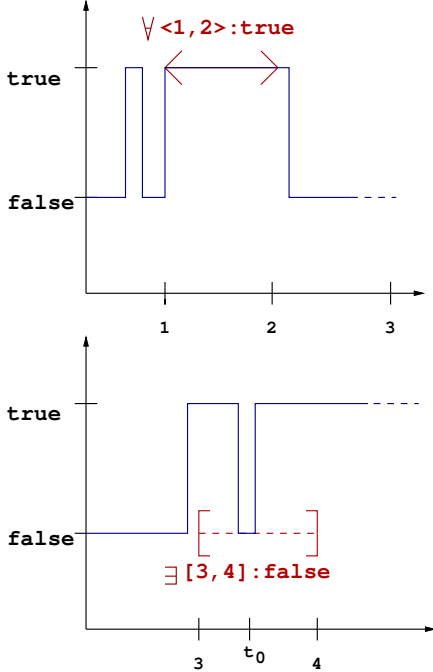


Figure 4: For the two kinds of constraints, one of the signals satisfying them

disjunctions are local bounds the cost of using this new domain with respect to the base domain.

- An automatic reduced-products for temporal abstract domains is introduced in Sect. V. The temporal aspect is at the core of this definition.

## II. TEMPORAL ABSTRACT DOMAIN

### A. An example

As explained in [5], Abstract Domains are the bricks of a static analyzer based on Abstract Interpretation, and encode the abilities of a theory in analyzing programs. A typical example of a *temporal* abstract domain, introduced in [2], is the domain of Abstract Constraints. It manipulates elements representing abstractly sets of signals that are *temporally similar* i.e. that take the same *Boolean* value  $b$  **around** a time  $t$ . There are *universal* and *existential* constraints that describe sets of signals denoted  $s$  in a generic way, and depicted in Fig. 4:

- An universal constraint is the data defined by a time interval  $[a; b]$  and a Boolean  $x$ , denoted  $\forall\langle a; b \rangle : x$ . It describes the property for signals  $s$  to take the value  $x$  during the whole time interval  $[a; b]$ . In the upper figure on the right, the Boolean signal satisfies the universal constraint  $\forall\langle 1; 2 \rangle : t^2$ .

- An existential constraint is the data defined by a time interval  $[a; b]$  and a Boolean  $x$ , denoted  $\exists[a; b] : x$ .

<sup>2</sup>In following formulas,  $t$  denotes Boolean value *true*, and  $f$  denotes *false*

It describes the property for signals  $s$  to take the value  $x$  at least once during the time interval  $[a; b]$ . In the lower figure on the right, the Boolean signal satisfies the existential constraint  $\exists[3; 4] : f$ . It takes the value  $f$ , among other time instants, at time  $t_0$ .

The concretization inside this abstract domain are presented in detail in [3]. But it is clear in this simple example, that the concretization  $\gamma$  of a single constraints is the set of signals satisfying the existential or universal constraint. In the case of several constraints, we may express more precise properties:

$$\begin{aligned} \bullet \gamma \left( \left\{ \begin{array}{l} \exists[0; 1] :: t \\ \exists[0; 1] :: f \end{array} \right\} \right) &= \left\{ \begin{array}{l} f : \mathbb{R}^+ \mapsto \mathbb{B} \text{changes} \\ \text{its value between} \\ t = 0 \text{ and } t = 1 \end{array} \right\} \\ \bullet \gamma \left( \left\{ \begin{array}{l} \forall\langle 0; 1 \rangle :: t \\ \forall\langle 0.5; 1.5 \rangle :: f \end{array} \right\} \right) &= \emptyset. \end{aligned}$$

The abstract information gathered at some point of the system is propagated abstractly by *abstract transfer functions*. The transfer functions for a lustre-like synchronous languages are also detailed in [3]. For example, let's assume that a variable  $v$  is known to have a semantics satisfying  $\forall\langle a; b \rangle : x$ . If later  $\neg v$  is defined and computed in the program, it is the role of the transfer function to compute automatically the abstract effect of the negation operation, and in this case it should return  $\forall\langle a; b \rangle : \neg x$ . Similarly, at the cost of only two additions, a communication channel transmitting information in a serial way with at least  $\alpha$  and at most  $\beta$  time units as delay, given an input satisfying the  $\exists[a; b] : x$  constraint may safely output  $\exists[a + \alpha; b + \beta] : x$ . These transfer functions are defined by hand and should be proved. We will see in Sect.III-B that abstract transformers may define some of them automatically when they create a new domain from an existing one.

An interesting property of this domain is that each basic constraint does not interfere at any time earlier than the left-limit of the time interval of the constraint or at any time later than its right limit. This means that if two constraints have their time interval not overlapping, the concretization of their conjunction is never empty. This kind of reasoning is automatized in the next section through the definition of the temporal support of any element of any temporal abstract domain.

The development [4] of two more continuous-time temporal domains in order to prove statically and automatically the properties of imperfectly clocked synchronous systems was successful, but developing each abstract domain is a lot of work.

However, these successive implementations showed similarities and we thought that we could take advantage of these similarities. We therefore introduce now a subclass of abstract domains that we call temporal abstract domains.

## B. Definition of Temporal Abstract Domains

Non-temporal abstract domains focus on having precise and fast abstract conjunction, disjunction and transfer functions. A temporal abstract domain additionally provides three additional temporal functions:

- the *temporal support*, that evaluates how early in time and how late an abstract element of the domain may have consequences. It will thus help us determine if two abstract elements, even from *different abstract domains* may interact or not, which is very useful for the writing of transfer functions.

- the *slicing function* defined for each element of the temporal domain and that rewrites an abstract element into several elements whose union overapproximates the initial element while having each a narrower temporal support and thus being easier to deal with.

- the *temporal interaction* function for couple of abstract elements, which tells, when two elements have overlapping temporal supports, how they may interact.

1) *Temporal support*: Dealing with temporal properties is difficult because time in a non-denumerable continuous set. It is therefore of the highest importance to limit the consequences of an element on the temporal point of view. This is provided by computing the *temporal support* function to any element of a temporal domain. The *temporal support* is an interval outside of which an abstract element has no influence.

**Support** A support of an element  $a^\#$  of a temporal abstract domain  $A^\#$  with concretization function  $\gamma$  is a time interval denoted  $S(a^\#)$  such that if  $S(a^\#) = [\alpha; \beta]$  then<sup>3</sup> if  $s_{[\alpha; \beta]} = s'_{[\alpha; \beta]}$  then  $s \in \gamma(a^\#) \Leftrightarrow s' \in \gamma(a^\#)$ .

We thus assume that the temporal abstract domain provides two functions  $\mathbf{r\_b}$  (for *right border*), and  $\mathbf{l\_b}$  (for *left border*) which are the bounds of this interval. For example, considering the abstract constraints domains:

$$\mathbf{r\_b} : \begin{cases} \forall \langle a; b \rangle : x \mapsto b \\ \exists \langle a; b \rangle : x \mapsto b \end{cases} \quad \mathbf{l\_b} : \begin{cases} \forall \langle a; b \rangle : x \mapsto a \\ \exists \langle a; b \rangle : x \mapsto a \end{cases}$$

Once defined on each temporal abstract domain, the support may be used for simplifying reasoning about abstract elements:

**Support of conjunction** A temporal support of a conjunction  $e_1 \wedge^\# e_2$  is the convex union of those of  $e_1$  and  $e_2$ , even if  $e_1$  and  $e_2$  are the canonical images in a reduced product of two elements of different abstract domains.

2) *Slicing*: The support of an abstract element is sometimes too wide for it to be easily handled. In that case, we would like to perform a temporal slicing and to

<sup>3</sup>with  $s_{[\alpha; \beta]}$  meaning  $s$  restricted to time interval  $[\alpha; \beta]$

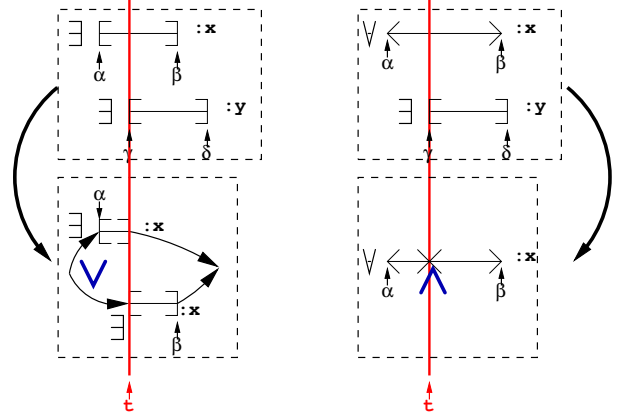


Figure 5: Isolating the part of the two upper constraints that interact with the lower constraints, with the help of the temporal slicing function at time  $t$ .

work on several elements whose conjunction is equivalent to initial element. Each of their temporal supports is then smaller.

**Slicing**: The *slicing* of an element  $e$  according to time instant  $t_0$  is a set of lists of elements  $R_{t_0}(e)$  whose disjunction is implied by  $e$  and whose elements  $c$  have a support included in the two period “before  $t$ ” and “after  $t$ ”:  $\forall P \in R_{t_0}(e), \forall c \in P, S(c) \subseteq ]-\infty; t_0] \vee S(c) \subseteq [t_0; +\infty[$ .

For example, for abstract constraints, if  $t \in [a; b]$ , then:

$$R_{t_0}(\exists \langle a; b \rangle : x) = \{\exists \langle a; t_0 \rangle : x\} \vee \{\exists \langle t_0; b \rangle : x\}.$$

$$R_{t_0}(\forall \langle a; b \rangle : x) = \{\forall \langle a; t_0 \rangle : x \wedge \forall \langle t_0; b \rangle : x\}.$$

As a consequence, if two elements interfere like in the two cases of Fig.5, a rewriting is possible for the upper element into an element that intrinsically interfere with the second one and another one which does not interfere with it. In this case, these rewritten elements are depicted in the two lower boxes and their left parts do not interfere with the lower element of the upper boxes.

3) *Temporal interactions*: Now, while manipulating two or more abstract elements, there may be some interactions. The simplest is the incompatibility between two elements: both cannot be satisfied at the same time. There may be also more subtle interactions that enforce the rewriting of the considered elements.

**Temporal interaction** The *temporal interaction* of two elements  $e$  and  $f$  is (an over-approximation of) the result of the conjunction of  $e$  and  $f$  whose temporal support overlap. It is denoted by  $I(e, f)$ .

In the constraint domain, the *temporal interactions* are complex yet can be computed in a fast way, as presented chapter 3 of [3]. For example, the *temporal interaction* of the two elements:

$$I(\exists[1;3] : f, \forall\langle 2;5 \rangle : t) \triangleq \begin{pmatrix} \exists[1;2] : f \\ \wedge \\ \forall\langle 2;5 \rangle : t \end{pmatrix}.$$

$I(\exists[1;3] : f, \exists[2;5] : f)$  can be defined as:

$$\begin{pmatrix} \exists[2;3] : f \\ \vee \\ \exists[1;2] : f \wedge \forall\langle 2;3 \rangle : t \wedge \exists[3;5] : f \end{pmatrix}.$$

Optionally, a last function called *coherence policy* may be defined. It introduces some restrictions to the elements contained in an abstract domain. Any transfer function has to check if it does not generated ill-formed elements that does not satisfy this policy. For example, in the abstract constraints domain, we may require that two constraints inside one element do not have their temporal support overlapping. We show in Sect. III-E a coherence policy for locally disjunctive domains.

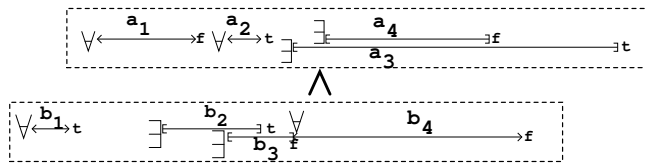
The construction of temporal abstract domain is thus always the same. It is enough to consider a mathematical theory with a continuous variable: *the time*. For each basic element, its temporal support should be computable, so that it is easy to know how far in the time this element can have consequences. We should be able to slice this element in several equivalent elements with narrower temporal support. Finally, the interactions between elements should be easily expressed. The transfer functions should be build *by hand*, but if the mathematical theory behind is well developed, this is most of the time simply an implementation of this theory. This is however enough to build *automatically* a conjunction which is already quite optimized and fast.

### C. Conjunction

A strength of temporal abstract domains is that the abstract conjunction can be performed in a linear time with respect to the sum of the sizes of inputs. Indeed, in order to compute an overapproximation of the abstract conjunction of two elements  $A^\# = (a_1, \dots, a_i, \dots, a_m)$  and  $B^\# = (b_1, \dots, b_i, \dots, b_n)$ , it is enough to cross them from left to right (or conversely) and apply the consequences of the *temporal interactions*  $I(a_i, b_j)$  only to the pairs  $(a_i, b_j)$  of basic elements whose temporal support  $S(a_i)$  and  $S(b_j)$  overlap.

For example, inside the abstract constraint domain, the conjunction of the two following elements involve only considering successively the couples:

$$(a_1, b_2), (a_2, b_2), (a_2, b_3), (a_3, b_4), (a_4, b_4)$$



More details on this linearity is presented in [3].

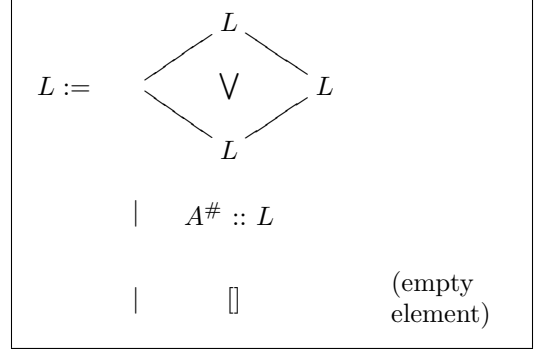


Figure 6: Definition of local disjunctions

## III. LOCAL DISJUNCTION ABSTRACT TRANSFORMER

We now introduce a temporal abstract domain transformer that take a temporal domain  $A^\#$  as argument and returns a new temporal domain  $A^\#_{\vee}$  where local disjunctions are allowed, which makes it more precise while not much more costly. The intuition for this transformation is that during a short time period where it does interfere neither with sub-elements with earlier support neither with sub-elements with later support, a sub-element can actually be a disjunction of sub-elements. This allow to describe more precisely sets of concrete signals while almost not modifying the abstract element from the syntactical point of view. This is furthermore done automatically, whatever the input temporal abstract domain, as soon as the three temporal functions introduced in Sect. II-B are provided.

### A. Local disjunctions

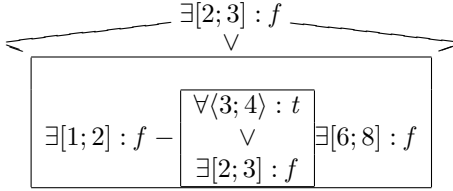
We allow local disjunctions, *i.e.* disjunctions that are local from the temporal point of view. They should only be used when needed, since they let the element grow in complexity. Local disjunctions are formally defined in a recursive way presented in Fig. 6. We thus let  $e :: l_1$  denote the result of the insertion an element from  $A^\#$  in the head position of a local disjunction  $l_1$ . The concatenation of two elements  $l_1$  and  $l_2$  is the recursive version of the  $[\ :: ]$  operator. The resulting element is denoted as  $l_1 @ l_2$ . The definition of *temporal support* can be extended to the elements of the resulting disjunctive domain.

However, this set of local disjunctions is restricted to lists  $l_1$  such that some hypothesis on the relative positions of the basic included elements is satisfied:

- If  $l_1 = \langle \begin{matrix} l_2 \\ \vee \\ l_3 \end{matrix} \rangle l_4$  and the element  $c$  appears in  $l_2$  or in  $l_3$ , and the element  $d$  appears in  $l_4$ , then  $r.b(c) \leq l.b(d)$ , (1)

- If  $l_1 = c :: l_2$ , and the element  $d$  appears in  $l_2$ , then any time in the temporal support of  $c$  has to be before any time in the temporal support of  $d$ , *i.e.*  $r\_b(c) \leq l\_b(d)$  (2).

An example of well-formed disjunctive element for  $A^\#$  being abstract constraints is:



### B. Transfer functions

We now try and define a sound abstract transfer function for the domain resulting of the transformer introducing local disjunctions. If the initial non-disjunctive abstract domain is denoted  $A^\#$ , the resulting locally disjunctive abstract domain is denoted  $A_\vee^\#$ . And if the analyzed program is written in a language containing a primitive  $T^\#$ , the abstract transfer function in  $A_\vee^\#$  for  $T$  can be easily defined from those of  $A^\#$  in the case where they maintain the order on temporal supports of any possible elements in  $A^\#$ :

**Temporal order-preserving operator** A temporal abstract operator  $T^\# \in A^\# \rightarrow A^\#$  is said to preserve temporal order if for all elements  $e, f \in A^\#$ , if  $r\_b(e) < l\_b(f)$  then  $r\_b(T^\#(e)) < l\_b(T^\#(f))$ .

This ensures that the temporal support of  $T^\#(e)$  and  $T^\#(f)$  do not overlap. In that case, the temporal transfer function is defined easily.

### Temporal transfer function can be soundly defined

If  $T^\#$  is a temporal order-preserving transfer function sound approximation of  $T$ , then  $T_\vee^\# \in A_\vee^\# \mapsto A_\vee^\#$  is defined recursively:

- $T_\vee^\# \left( \left\langle \begin{array}{c} l_2 \\ \vee \\ l_3 \end{array} \right\rangle l_4 \right) = \left\langle \begin{array}{c} T_\vee^\#(l_2) \\ \vee \\ T_\vee^\#(l_3) \end{array} \right\rangle T_\vee^\#(l_4)$
- $T_\vee^\#(e :: l) = T^\#(e) :: T_\vee^\#(l)$

Then,  $\forall l \in A_\vee^\#, T_\vee(l)$  is a well formed element of  $A_\vee^\#$  and  $T_\vee \circ \gamma \subseteq \gamma \circ T_\vee^\#$ , where  $\gamma$  is the concretization function

### C. Concretization

The concretization inside the new abstract domain  $A_\vee^\#$  is defined recursively as follow:

- $\gamma_\vee^\# \left( \left\langle \begin{array}{c} l_2 \\ \vee \\ l_3 \end{array} \right\rangle l_4 \right) = (\gamma_\vee^\#(l_2) \cup \gamma_\vee^\#(l_3)) \cap \gamma_\vee^\#(l_4)$
- $\gamma_\vee^\#(e :: l) = \gamma^\#(e) \cap \gamma_\vee^\#(l)$

The new locally-disjunctive abstract domains is thus under a few hypotheses on the initial domain quite close to the initial domain, as long as transfer functions are considered. Computing the abstract conjunction is a bit more complex and needs some properties to be proved.

It is mandatory to be able to slice the elements of the locally-disjunctive domain:

**A local disjunction can be sliced** For any time instant  $t$ , for any locally-disjunctive element  $l$ , there exists  $l'_t$ , an equivalent or an overapproximation of  $l$  (*i.e.* having a concretization that include the one of  $l$ ) such that no constraint inside  $l'_t$  intersects  $t$  (*e.g.*  $\forall \langle a; b \rangle : x$  or  $\exists \langle a; b \rangle : x$  with  $a < t < b$  is not acceptable).

Indeed,  $l'_t$  can be obtained by recursively going through  $l$  and splitting any element  $e$  whose temporal support strictly contains  $t$  into  $R_t(e)$ . Furthermore, this operation is also very fast, since it only induces few local changes.

### D. Conjunction

Two local disjunctive elements are said to be *separate* if their supports do not intersect.

### The conjunction of two separate local disjunction

The conjunction of two separate local disjunctions is their simple syntactical concatenation, the one with the earliest *support* being ahead.

This is proved by induction on the local disjunction which has the earliest *support* among the two input arguments.

For non-separate elements, this is more difficult. An example inside the disjunctive abstract constraint domain is proposed in [3]. The following cases may happen:

- Let us assume that both local disjunctions start with elements denoted  $c \in A^\#$  and  $d \in A^\#$ . In this case, the *interaction rules* apply and the result consists in their result, followed by the conjunction of the eventually remaining tails of the elements.

- If one of the local disjunctions starts with a branching:  $\left\langle \begin{array}{c} l_1 \\ \vee \\ l_2 \end{array} \right\rangle l_3$  while the other starts with an element  $c_1$ ,

followed by  $l_4$ , two sub-cases may happen:

- $r\_b(c_1) \leq l\_b(l_1)$  and  $r\_b(c_1) \leq l\_b(l_2)$ . In this case, we perform the conjunction between  $l_4$  and

the result of the conjunction of  $c_1$  and  $\left\langle \begin{array}{c} l_1 \\ \vee \\ l_2 \end{array} \right\rangle$ ,

concatenated with  $l_3$ .

- $l_1$  or  $l_2$  both end earlier than  $c_1$ . In that case, we use the separability introduced by theorem III-C and separate so that it is no longer the case.

- The easiest case is when there exist a time instant  $t$  that separates  $l_1 = l_{1,head}@l_{1,tail}$  and  $l_2 =$

$l_{2,\text{head}}@l_{2,\text{tail}}$  i.e.  $\text{support}(l_{1,\text{head}}) \leq t \leq \text{support}(l_{1,\text{tail}})$  and  $\text{support}(l_{2,\text{head}}) \leq t \leq \text{support}(l_{2,\text{tail}})$  In that case, the result of the conjunction is defined recursively:

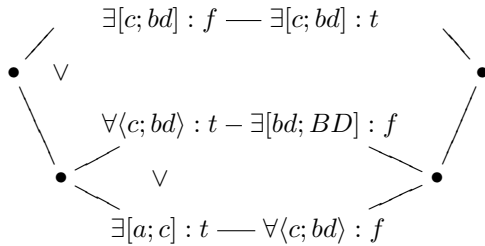
$$\begin{aligned} & (l_{1,\text{head}}@l_{1,\text{tail}}) \wedge (l_{2,\text{head}}@l_{2,\text{tail}}) \\ \Leftrightarrow & (l_{1,\text{head}} \wedge l_{2,\text{head}}) @ (l_{1,\text{tail}} \wedge l_{2,\text{tail}}) \end{aligned}$$

As a result, performing an optimized conjunction over two local disjunctions of elements, which can be exponential in the size of the arguments, if they have many overlapping elements in the disjunctive branches, can be defined independently of the underlying abstract domain and is enough in practice, at least for tests in this domain. If it is not fast enough, a specific conjunction should then be defined. In the case of the Abstract Constraints introduced in Sect. 2.1, the resulting locally-disjunctive abstract constraints domain is almost as fast as the initial domain.

### E. Reducing overlapping

Once local disjunction are defined, the overlapping of existential constraints can always be reduced by partitioning to disjunctions of non-overlapping constraints and of constraints with exactly the same interval. This is useful since it ensures that along a left to right crossing path there are never two constraints that overlap, except if both are existential constraints and have same support. There are therefore never interactions between constraints along a left to right crossing path, since temporal support are disjoint.

Reducing overlapping can be done since any signal satisfying two overlapping constraints  $c_1 = \exists[a; b] : t$  and  $c_2 = \exists[c; d] : f$ , with  $a \leq c$ , either takes both Boolean value on the overlapping interval or only  $t$  or only  $f$ . As a result,  $c_1 :: c_2$  can be rewritten to the following, with  $bd$  denoting  $\min(b, d)$  and  $BD$  denoting  $\max(b, d)$ :



Although this reduction may seem technical only, it is a key to having many of the operations in temporal abstract domain be performed in a *linear time*. Indeed it ensures that in any branch of a locally disjunctive element, the sub-element do not have their temporal support that overlap (except if done on purpose).

## IV. ANOTHER TEMPORAL ABSTRACT DOMAIN, THE CHANGES COUNTING DOMAIN

We now introduce the Changes Counting domain in order to demonstrate the reduced-product between temporal abstract domains. The Changes Counting domain was designed in order to deal automatically with reasoning on stability and the variability of systems.

**Local Changes counting domain:** The Local Changes counting Domain is the set of elements  $(\leq k, a \blacktriangleright\blacktriangleleft b)$  and  $(\geq k, a \blacktriangleright\blacktriangleleft b)$ , for  $a, b \in \mathbb{R}^+$  and  $k \in \mathbb{N}$ .

The meaning of an element  $(\leq k, a \blacktriangleright\blacktriangleleft b)$  (respectively  $(\geq k, a \blacktriangleright\blacktriangleleft b)$ ) is that behaviors in the concrete semantics do not change their value more (respectively less) than  $k$  times during time interval  $[a; b]$ . The concretization, and transfer functions for a lustre-like synchronous languages inside this abstract domain are presented in detail in [3].  $(= k, a \blacktriangleright\blacktriangleleft b)$  is just short for  $(\leq k, a \blacktriangleright\blacktriangleleft b) \wedge (\geq k, a \blacktriangleright\blacktriangleleft b)$ . The temporal support of any abstract element  $(\leq k, a \blacktriangleright\blacktriangleleft b)$  or  $(\geq k, a \blacktriangleright\blacktriangleleft b)$  is  $[a; b]$ . In the changes counting domain, if  $t \in [a; b]$ , the slicing is defined as in this example:

$$\begin{aligned} R_t((\leq 3, a \blacktriangleright\blacktriangleleft b)) = & \{ (= 3, a \blacktriangleright\blacktriangleleft t) \wedge (= 0, t \blacktriangleright\blacktriangleleft b) \} \\ & \vee \{ (= 2, a \blacktriangleright\blacktriangleleft t) \wedge (\leq 1, t \blacktriangleright\blacktriangleleft b) \} \\ & \vee \{ (= 1, a \blacktriangleright\blacktriangleleft t) \wedge (\leq 2, t \blacktriangleright\blacktriangleleft b) \} \\ & \vee \{ (= 0, a \blacktriangleright\blacktriangleleft t) \wedge (\leq 3, t \blacktriangleright\blacktriangleleft b) \} \end{aligned}$$

In the changes counting domain, the *temporal interactions* are computed according to similar rules as for Abstract Constraints, which is described in chapter 4 of [3]. We will recall them when needed. For example, the *temporal interaction* of the two elements  $a = (\leq k, u \blacktriangleright\blacktriangleleft v)$  and  $b = (\leq l, w \blacktriangleright\blacktriangleleft z)$  with  $u \leq w$ ,  $z \leq v$  and  $k \leq l$ , is  $a \cap^\# b \triangleq a$ .

## V. REDUCED PRODUCT

### A. Ad hoc optimizations between abstract constraints and changes counting domains

The abstract time used by temporal abstract domains can be used as a basis for a communication language between the domains. It thus enables the definition of a precise reduced-product. In particular, if two elements from different temporal abstract domains have temporal support that do not overlap, it is impossible to gain any more information about them.

If they overlap, they might be reduced, i.e. the elements from both domains may be rewritten in a different way which is more efficient for proving their property. But this reduction is not guaranteed. The temporal aspect thus allows to restrict the number of cases where a reduction may be applied, which is of great help.

In Fig. 7 and 8, four cases are considered, following a simple arithmetic for temporal support. The lower right case is the more complex. As a consequence of having no

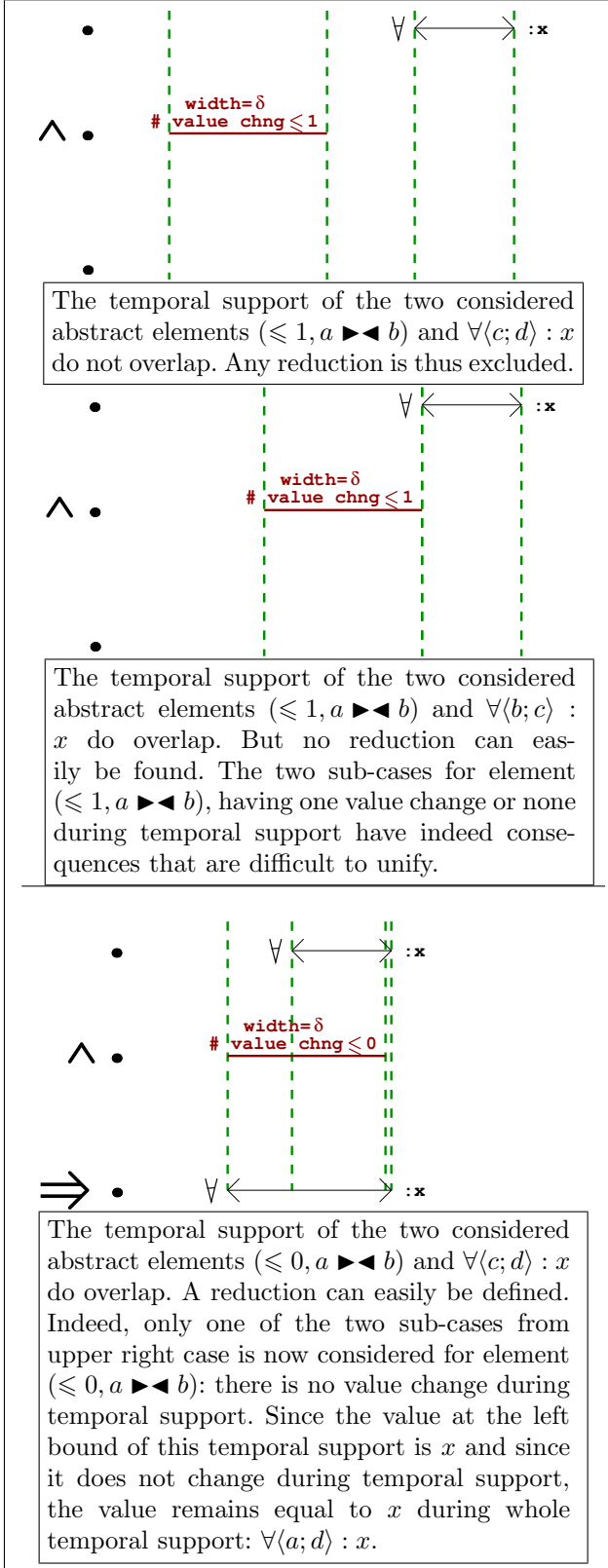


Figure 7: Three cases for the reduction of two elements of different temporal abstract domains

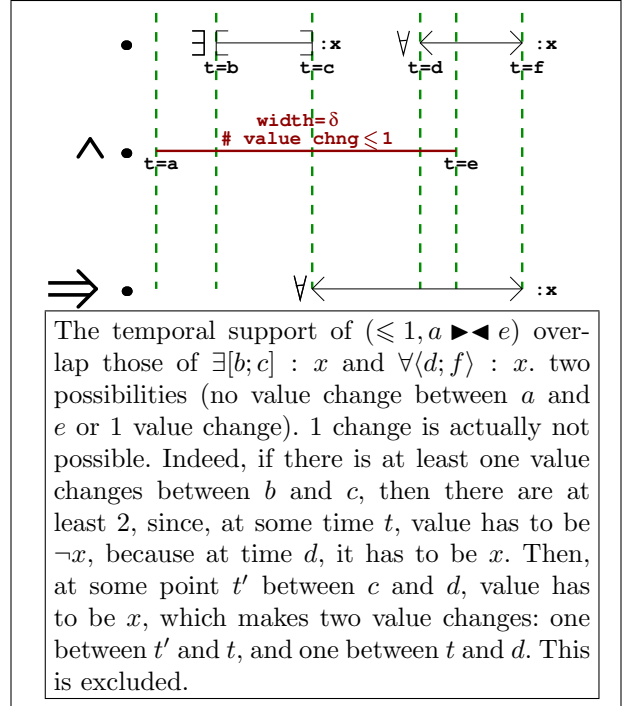


Figure 8: Last case for the reduction of two elements of different temporal abstract domains

no value change between  $a$  and  $e$  and since the value at time  $d$  is  $x$  and since it does not change, the value has to remain equal to  $x$  during whole time interval, which can be translated into  $\forall \langle c; d \rangle : x$ . This constraint fuses with constraint  $\forall \langle d; f \rangle : x$  into  $\forall \langle c; f \rangle : x$ .

The above reasoning being always the same, it may be interesting not only to automatize them, but even to automatize their discovering, whatever the two temporal abstract domain that we try to reduce.

### B. Automatic reduced-product by partitioning

The temporal considerations can also be the key to an automatic reduction by partitioning. Indeed, introducing a disjunctively complete case study inside the temporal support can allow a precise reduction. Furthermore, the cost of the partitioning which is commonly seen as its biggest disadvantage is here bounded by considering only partitioning inside one temporal abstract domain in the temporal support of an element of another abstract domain. If the two elements have their temporal supports overlapping, there might be an interesting reduction.

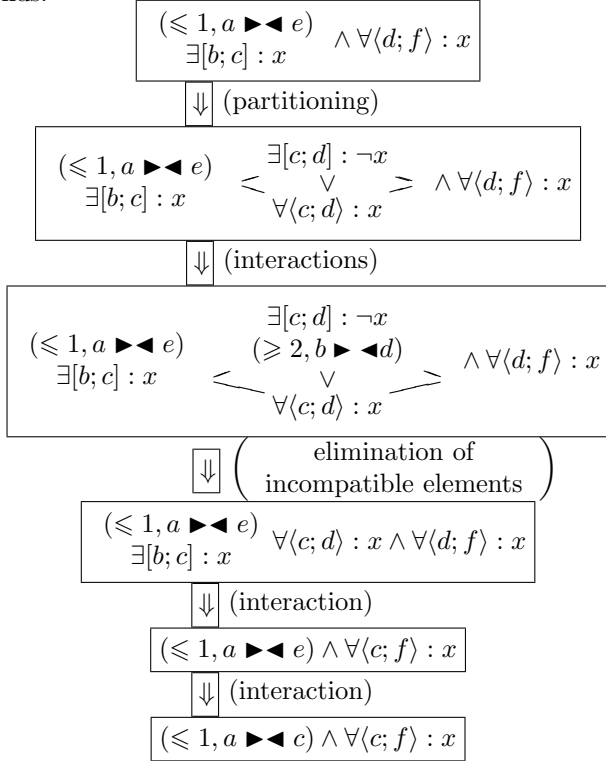
For example, a disjunctive partition over a temporal support  $[a; b]$  for an abstract element  $\forall \langle b; c \rangle : x$  is:  $\exists [a; b] : x \vee \forall \langle a; b \rangle : \neg x$ .

A disjunctive partition over a temporal support  $[a; b]$  for an abstract element  $(\leq 1, a \blacktriangleright \blacktriangleleft c)$  is:  $(= 0, a \blacktriangleright \blacktriangleleft b) \vee (= 1, a \blacktriangleright \blacktriangleleft b)$ .

This enables the discovering in an almost automatic way of many reductions, including the lower right case of Fig. 7 considered above. Indeed, the only missing part for an automatic reasoning to complete is for it to know that, if  $b \leq c$ , then:

$$\exists[a; b] : x \wedge \exists[c; d] : \neg x \Rightarrow (\geq 1, a \blacktriangleright \blacktriangleleft d).$$

The automatic reasoning, for  $a < b < c < d < e < f$  is thus:



## VI. IMPLEMENTATION

A prototype of static analyzer has been developed implementing the two temporal abstract domains presented in Sect. 2 and 4 as well as the domains introduced in [3].

The prototype of analyzer was able to prove temporal properties of several systems including an interesting one made of redundant units with a voting unit deciding between them. The specification was proved automatically although a case study seems out of reach.

Furthermore, when a property does not hold, looking at the abstract set that could not be reduced to the abstract empty set led to discover easily erroneous traces in altered implementations.

No hypothesis was given on the inputs of the studied system but a specification was given for an output. Looking at the system, the only thing that was obvious was that it was without hypothesis on inputs, the system does not satisfy the specification.

We started several automatic analyzes with several hypothesis for a constant  $k$  to be the input stability.

- with an input stability of  $k_0$  milliseconds, the analyzer could prove the specification

- with an input stability of  $2/3 \times k_0$  milliseconds, the analyzer could not prove the specification but inside the abstract result computed, it was very easy (and that could have been made automatically) to find a counter-example to the specification.
- with an input stability between  $2/3 \times k_0$  and  $k_0$  milliseconds, the analyzer could not prove the specification while the abstract result does not show an obvious counter-example to the specification. It is therefore unknown if the specification then holds.

This result is very interesting since it demonstrates the necessity of stabilizing input signals. But the analyzer also gives a minimal value for this stabilization. In order for analyzer to get closer to the optimal stabilization *i.e.* suggest a smaller time interval for which we have no information, domains should be improved or new more precise abstract domain should be added.

## Related works

Considering the time as a continuous notion is not the usual choice of the designers of embedded systems. It may however be of great interest for the certification of these systems. The duration calculus, introduced in [16] is a logic whose classical model is based on continuous-time, modeled as real numbers, as in our work shares, but duration calculus is defined as a logic and not as an abstract domain, and consequently has no operators for the primitives of programming languages (the transfer functions). It thus cannot propagate by itself any information it proved for some part of the code to other parts. It is however used for model checking in [8]. UP-PAAL2K, introduced in [14] allows the design of networks of communicating timed-automata but also integrates a model-checker dedicated to temporal specifications and also can have a continuous-time semantics.

To my knowledge, the only other works where the time is not abstracted as a discrete notion during the static analysis are the one of O. Bouissou and M. Martel in [6] as well as the one by S. Thompson and A. Mycroft, who proposed in [15] several abstractions to study asynchronous circuits and the changes counting domain we introduce has similarities with their work. The abstract time is reduced to a temporal relation between events.

The difficulties of continuous-time behaviors can sometime be simulated in discrete frameworks like **Lustre** using the primitives **when** and **current**, as presented in [7], [13], [12], [11], but we will rather express them in a continuous time semantics that will get easily and precisely abstracted. [1] proposes a protocol that ensures the robustness of the system to clock imprecisions. In parallel, we tried to present abstract domains that can automatically perform similar reasoning on the code produced following guidelines presented in these articles, even if this code has been modified later.

## VII. CONCLUSION AND FUTURE WORKS

Continuous-time abstract analysis enables thus precise and inexpensive abstractions. These abstractions need new *temporal* abstract domains that have yet to be defined. It allows the reuse of theories about continuous numbers that are very well known.

Not only the definitions of some parts of a temporal domain (abstract transfer functions, abstract conjunction, ...) can be automatized, but also refinements of the whole domains through the use of abstract transformers, as the one introducing local disjunctions. Even some reductions between different domains can be partially defined automatically, precisely in the case where both abstract domains are temporal.

The domains we defined are non-relational. They describe what happens at *one* point of the system and can propagate it at another point, then only focusing on this other point. In the future, it would be of great help to have new relational temporal domains. Some early experiments in our prototype with a domain based on the directed topology introduced in [10], [9] show that this can be done in our framework.

*Acknowledgments:* I would like to thank professor He Jifeng and faculty members of SEI lab at ECNU University (Shanghai, China). Part of the research for this article was done during a post-doc in their team, where I was very warmly welcomed.

## REFERENCES

- [1] A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. *LNCS, Proceedings of the Second International Conference on Embedded Software*, p. : 252 - 265, 2002.
- [2] J. Bertrane. Static analysis by abstract interpretation of the quasi-synchronous composition of synchronous programs. *VMCAI*, 2005.
- [3] J. Bertrane. *Analyse statique de systèmes synchrones communicants à horloges imparfaites par interprétation abstraite à temps-continu*. PhD thesis, École Polytechnique, 2008.
- [4] J. Bertrane. Proving the properties of communicating imperfectly-clocked synchronous systems. *13th International Static Analysis Symposium (SAS'06)*, Seoul, South Korea, 29-31 august, 2006.
- [5] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace (I@A 2010)*, number AIAA-2010-3385, pages 1–38. AIAA (American Institute of Aeronautics and Astronautics), Apr. 2010. <http://www.di.ens.fr/~mine/publi/bertrane-al-aiaa10.pdf>.
- [6] O. Bouissou and M. Martel. Abstract interpretation of the physical inputs of embedded programs. In *9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 08)*, San Francisco, 2008.
- [7] P. Caspi, C. Mazuet, R. Salem, and D. Weber. Formal design of distributed control systems with lustre. In *(Safecomp'99)*, volume 1698 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.
- [8] Z. Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, December 1991.
- [9] E. Goubault. Some geometric perspectives in concurrency theory. *Homology, Homotopy and Applications*, 5:95–136, 2003.
- [10] E. Goubault and E. Haucourt. A practical application of geometric semantics to static analysis of concurrent programs. In M. Abadi and L. de Alfaro, editors, *CONCUR 2005*, volume 3653 of *LNCS*, pages 503–517, 2005.
- [11] N. Halbwachs and S. Baghdadi. Synchronous modelling of asynchronous systems. In Springer Verlag, editor, *EMSOFT'02*, volume 2491 of *LNCS*, 2002.
- [12] N. Halbwachs and L. Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *6th International Conference on Application of Concurrency to System Design (ACSD'06)*, Turku, Finland, 2006.
- [13] E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, and D. Lesens. Virtual execution of aadl models via a translation into synchronous programs. In *Proc. of the 7th International Conference on Embedded Software (EMSOFT 2007)*, Salzburg, Austria, 2007.
- [14] P. Pettersson and K. G. Larsen. Uppaal2k. *Bulletin of the European Association for Theoretical Computer Science*, volume 70, pages 40-44, 2000.
- [15] S. Thompson and A. Mycroft. Abstract interpretation of asynchronous circuits. *SAS, Verona, Italy*, August 2004.
- [16] Ch. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. Springer, 2004.