Arborescent canonical form of boolean expressions

Tochéou Pascalin Amagbegnon, Loïc Besnard, Paul Le Guernic

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Arborescent Canonical Form of Boolean Expressions

Tochéou AMAGBEGNON

Loïc BESNARD

Paul LE GUERNIC

N° 2290

Juin 1994

PROGRAMME 2

Calcul symbolique,

programmation

et génie logiciel

Rapport de recherche

ISSN 0249-6399

Arborescent Canonical Form of Boolean Expressions

Tochéou AMAGBEGNON
Loïc BESNARD
Paul LE GUERNIC

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet EP-ATR

Rapport de recherche n ° 2290 — Juin 1994 — 63 pages

Abstract: SIGNAL is a synchronous language designed to program real-time systems. Because of its equational style, its compilation requires the *statical* resolution of a system of boolean equations; the variables being *clocks*. This report discusses the arborescent representation of SIGNAL clocks. We introduce a BDD-based data structure called *hierarchy*. Through the factorization of boolean functions, we show that hierarchies are a canonical form of clocks. We also show that this canonical form optimizes the sequential code generated from a SIGNAL program. We finally link hierarchies to the well known ordering problem of BDDs.

Key-words: SIGNAL, BDD, clock calculus, boolean lattice

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Forme Canonique Arborescente des Expressions Booléennes

Résumé : SIGNAL est un langage synchrone pour le temps-réel. Parce qu'il est équationnel, sa compilation requiert que soit résolu *statiquement* un système d'équations booléennes; les variables étant les *horloges*. Ce rapport traite de la représentation arborescente des horloges de SIGNAL. Nous y présentons une structure de données appelée *hiérarchie*. Avec de la factorisation de fonctions booléennes, nous montrons que les hiérarchies sont une forme canonique des horloges. Dans le même temps nous montrons que cette forme canonique optimise le code séquentiel généré à partir d'un programme SIGNAL. Enfin, nous établissons un lien entre les hiérarchies et le problème bien connu de l'ordre dans les BDDs.

Mots-clé : SIGNAL, BDD, calcul d'horloge, treillis booléen

1 Introduction

Real-time systems, and more generally reactive systems [1], constantly interact with their physical environment. Because they must respond to externally generated stimuli, they must be programmed so that they keep pace with the environment. Moreover, in real-time systems safety is very often a big concern. Thus, in such systems, one would wish to prove correctness. Response time and correctness are the major issues to tackle when developing a real-time application.

Traditionally, real-time systems have been programmed with imperative asynchronous languages like:

- C together with some real-time operating system facilities,
- ADA [29] which handles concurrency by rendezvous constructs,
- OCCAM [23, 11] based on Hoare's "Communicating Sequential Processes" formalism [22]

Although they are widely used, they suffer severe drawbacks.

- They lack determinism,
- time must be handled explicitly; the programmer cannot fully abstract from it,
- due to the indeterminism, proofs are highly complex.

There are alternatives to the indeterminism of the previous formalisms.

- The GRAFCET [20] based on Petri nets; it is used on programmable controllers; it lacks clean semantics, hierarchical constructs and proof tools
- Finite state machines (FSM's): there are tools to prove correctness of systems specified with FSM's [16, 17]. The major problem with using FSM's as design language is that a slight modification in the specification may (and generally does) result in a dramatic increase in the number of

states, and composing two FSM's is a rather tedious task. However, due to the power of the tools available (both software and mathematical), FSM's are produced as result of the compilation of higher level languages.

The insufficiencies of the previous formalisms, well discussed in [1, 4], have lead to the design of a new paradigm: *synchronous programming*. The hypotheses of synchronous programming are that:

- calculations and communication actions have zero duration (**instantaneousness**)
- simultaneity exists (**simultaneousness**)

Thus, time is treated from a chronological point of view; succession and simultaneity of intants.

Nevertheless, duration matters are not ignored. They are taken into account specifically for each target architecture: sequential machines, Digital Signal Processors, transputers, ... This allows to fully abstract from implementation details when programming with synchronous languages. Freeing the programmer from explicit management of time puts the burden on the compilers of these languages.

Many synchronous languages are being developed: Esterel [4], Lustre [12] and SIGNAL [26]. The synchronous approach is summarized in [21].

In this report we focus on the language SIGNAL and present part of the techniques used to implement its compiler. The remainder is organized as follows:

- Section 2 introduces the basic elements of the SIGNAL language.
- Compiling a SIGNAL program requires the resolution of a system of boolean equations and efficient code generation requires a "good" representation of the solutions of the equations; section 3 will present the form of the boolean equations we are to solve and an arborescent representation for the resolution.

- Section 4 defines a binary decision diagram-based data structure called *hierarchy* and in section 5, we use hierarchies as an arborescent canonical form of our boolean equations.
- Finally in section 6 we link hierarchies with the well known ordering problem of binary decision diagrams [10].

2 Presentation of the language SIGNAL

SIGNAL [9] is a language designed for real-time programming. It is synchronous, data-flow oriented, declarative and equational. A SIGNAL program is a system of equations on **signals**.

2.1 Signals and clocks

2.1.1 Signals

A signal X is a sequence $(X_t)_{t \in I}$ of typed values (integer, booleans, reals,...). I is a *totally ordered* set of *instants*. We are interested in a discrete time model. So, instants are taken in a denumerable set.

At any given instant t, a signal may be *present* or *absent* depending on whether or not, the instant under consideration belongs to I. Obviously, a signal carries a value only when it is present.

2.1.2 Clocks

The set of instants at which a signal is present is its clock. So, the clock of a signal $(X_t)_{t \in I}$ is its time index I.

Two signals always present at the same instants are said to be *synchronous*: they have the same clock. Thus, the clock of a signal X is the equivalence class of X for the *synchrony relation*; in that sense it is denoted $\widehat{X}$.

Formal semantics of SIGNAL can be found in [25, 3, 2].

Notation: Following [5], the set theoretic operators for clocks, which are sets, are denoted $\wedge$ (intersection), $\vee$ (union) and $\backslash$ (set difference).

2.2 Processes

A statement in SIGNAL is an equation on signals; it is called a process. Here are the different types of processes.

2.2.1 Functional expressions

The operators $(+, -, *, and, \ldots)$ defined on basic data types (booleans, integers, ...) are canonically extended to sequences and consequently to signals. Let f be such an operator of arity n and let $(X_{1t})_{t\in I}, \ldots, (X_{nt})_{t\in I}$ be n sequences with the same time index I. The equation

$$\forall t \in I,\ Y_t = f(X_{1t}, \ldots, X_{nt})$$

is written in SIGNAL

$$Y := f(X_1, \ldots, X_n)$$

The signals involved in that equation are required to have the same time index: they must be synchronous. Thus, the definition of the signal Y implies the following equation on the clocks:

$$\widehat{Y} = \widehat{X_1} = \ldots = \widehat{X_n}$$

Figure 1 shows a timing diagram for such a process.

2.2.2 Reference to past values

We reference past values of a — discrete — signal with the "\$" operator. The SIGNAL process

$$ZX := X\$1$$

is the transcription of the following equation on the sequences $(X_t)_{t\in I}$ and $(ZX_t)_{t\in I}$ defined on the same index I:

$$\forall t \in I,\ ZX_t = X_{t-1}$$

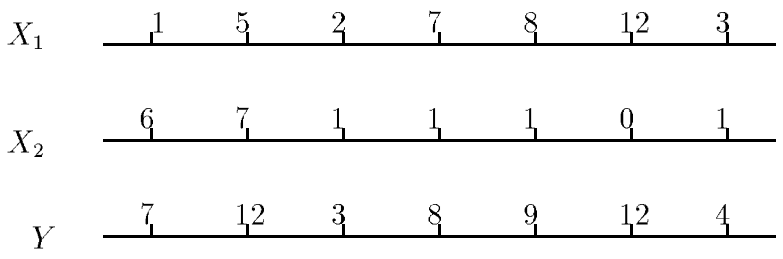


Figure 1: $Y := X_1 + X_2$

Here again, ZX is by definition synchronous with X. The initial value is specified for ZX with the declaration ZX *init* v_0. A timing diagram for this operator is depicted on figure 2.

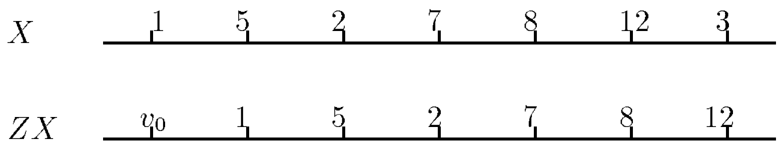


Figure 2: $ZX := X\$1$

2.2.3 Downsampling

Given a signal X and a boolean-valued signal C, the process

$$Y := X \ when \ C$$

defines the signal Y such that:

- $\widehat{Y} = \widehat{X} \wedge [C]$; $[C]$ is the set of instants where C occurs and is *true*.
- $\forall t \in \widehat{Y}$, $Y_t = X_t$. At the instants where Y occurs, it carries the same value as X.

$(Y_t)_{t \in \widehat{Y}}$ can be viewed as a subsequence of $(X_t)_{t \in \widehat{X}}$; $\widehat{Y}$ being a subset of $\widehat{X}$. A timing diagram is depicted on figure 3.

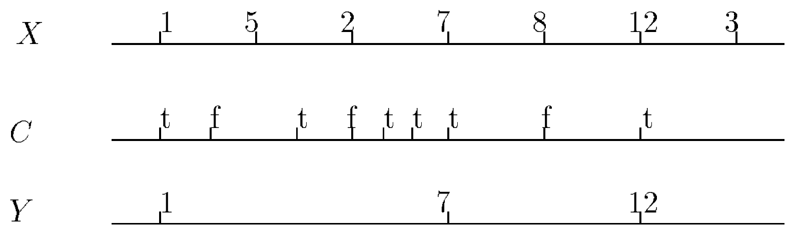


Figure 3: $Y = X\ when\ C$

2.2.4 Deterministic merge

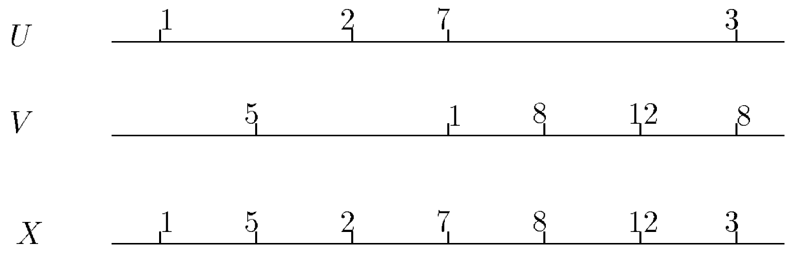


Figure 4: $X = U\ default\ V$

Given two signals U and V, the process

$$X := U\ default\ V$$

defines the signal X such that:

$$\begin{cases} \widehat{X} = \widehat{U} \vee \widehat{V} & \\ \forall t \in \widehat{U} & X_t = U_t \\ \forall t \in \widehat{V} \setminus \widehat{U} & X_t = V_t \end{cases}$$

X_t is well defined for all $t \in \widehat{X}$, since any instant in $\widehat{X}$ belongs either to $\widehat{U}$ or to $\widehat{V} \setminus \widehat{U}$.

Informally, when U is present, X is equal to U; otherwise, X is equal to V. X is absent when neither U nor V is present. A timing diagram is depicted on figure 4.

2.2.5 Composition of processes

The elementary processes we have presented till now may be composed with the commutative and associative operator "|". From an equational point of view, this operator is the union of two systems of equations. For example the system:

$$\forall t \in I, \left\{ \begin{array}{rcl} ZX_t & = & X_{t-1} \\ X_t & = & ZX_t + Y_t \end{array} \right.$$

is the following SIGNAL process:

$$\begin{array}{ll} (\; | & ZX \;:=\; X\$1 \\ \;\;\;\; | & \;\;X \;:=\; ZX + Y \\ \;\;\;\; | \;) & \end{array}$$

2.2.6 Derived operators

For convenience, the full language SIGNAL offers many derived operators; they can be expressed in terms of the kernel operators. Here are some of them:

- the unary *when*: *when* C is an abbreviation for C *when* C
- the operator *event*: *event* X gives the clock of a signal X; in fact, it is an abbreviation for the boolean signal defined by *event* $X \;:=\; (X = X)$; it is synchronous with X and carries the value *true* each time it occurs; thus it can be identified with $\widehat{X}$ the clock of X
- the process *synchro*$\{X_1, \ldots, X_n\}$ specifies the equality of the clocks of the signals $X_1, \ldots, X_n$.

3 Compiling a SIGNAL program

The important point of SIGNAL compilation [26, 2] is to ensure the respect of the synchronization of the computations expressed by the language. It is necessary, on the one hand, to calculate the relations among the different clocks and, on the other hand, to analyze the dependencies between the calculi of the program. To achieve this, we caracterize a SIGNAL program [26] by an abstract representation which combines an equational control modeling (presented in sub-section 3.2) with a dependence graph (presented in sub-section 3.3). Over this abstract representation, two complementary studies are perfomed:

- the study of static properties, in other words the set of invariant temporal properties ; they allow to characterize the set of states in which the automaton associated to a SIGNAL program can evolve, *independently of initial values*, and the set of the transitions between these states.
- the study of dynamical properties which take initial state and trajectories into account, through the boolean signals defined by the delay operator($).

3.1 Purposes and motivations

In this report, we focus on the study of static properties. It is a preliminary study for code generation and study of dynamical properties.

We have, in the previous section, expressed the temporal relations in terms of availability/unavailability of signals, and we have noted the particular role of boolean expressions when they are used to define a second operand of the **when** operator. Such an expression introduces two new clocks (downsampling) depending on the possible values ("true","false") of that expression when it is defined. One can observe that the relevant information to reason about clocks is concentrated in three values: present-true, present-false, absent. To deal with this information, synchronizations can be modelled:

- in the commutative field $\mathbb{Z}/3\mathbb{Z}$ by the following coding ; to a boolean signal **C** is associated a variable C which takes its values in the set $\{0, 1, -1\}$ which means respectively the absence of the signal, the presence of the signal with the "true" value and, lastly, the presence of the signal with the "false" value. Since the presence/absence of the boolean signal **C**, whatever its value is, is obtained by C^2, which takes its values in $\{0, 1\}$, the presence/absence of a non boolean signal **X** is also denoted by X^2. This encoding is part of a denotational semantics thoroughly presented in [2]. A theory [8] and a tool named SIGALI have been built on this concept to analyze and verify dynamical properties of SIGNAL programs.

- in the boolean lattice associated with a set of variables by laying down constraints on some boolean variables. Here we present this method. We will focus on the techniques actually implemented in the compiler to partially solve the system of boolean equations resulting from a SIGNAL program ; the partial synthesis of explicit expressions of control is expressed by a collection of trees whose roots can be arguments of non solved constraints and internal nodes are explicit expressions.

3.2 System of boolean equations

It appears clearly from the previous section that an equation on clocks lies under each signal process.

- For a functional expression $Y := f(X_1, \ldots, X_n)$ the requirement on the clocks is $\widehat{Y} = \widehat{X_1} = \ldots = \widehat{X_n}$.
- For a delay process $ZX := X\$1$, we have $\widehat{ZX} = \widehat{X}$.
- For a deterministic merge $X := U \; default \; V$ the clocks are related by $\widehat{X} = \widehat{U} \vee \widehat{V}$.
- For a downsampling $Y := X \; when \; C$, we have $\widehat{Y} = \widehat{X} \wedge [C]$ where $[C]$ denote the clock of the signal *when* C.

The boolean signal C used as the second operand of the *when* operator introduces supplementary equations. Indeed, consider the clocks denoted $[C]$ and $[\neg C]$ and defined as:

$$\begin{array}{rcccl} [C] & = & when\ C & = & the\ set\ of\ instants\ at\ which\ C \\ & & & & carries\ the\ value\ true \\ [\neg C] & = & when(not\ C) & = & the\ set\ of\ instants\ at\ which\ C \\ & & & & carries\ the\ value\ false \end{array}$$

The pair $([C], [\neg C])$ defines a *partition* of the clock $\widehat{C}$. Thus, we have the additional information

$$\left\{ \begin{array}{rcl} [C] \vee [\neg C] & = & \widehat{C} \\ [C] \wedge [\neg C] & = & \mathbb{O} \;\; \text{the null clock, the empty set of instants} \end{array} \right.$$

All these equations are recapitulated on table 1

signal process	clock calculus	additional equations
Y:= f(X$_1$,...,X$_n$)	$\widehat{Y} = \widehat{X_1} = \ldots = \widehat{X_n}$	
ZX:= X$1	$\widehat{ZX} = \widehat{X}$	
X:= U default V	$\widehat{X} = \widehat{U} \vee \widehat{V}$	
Y:= X when C	$\widehat{Y} = \widehat{X} \wedge [C]$	$[C] \vee [\neg C] = \widehat{C}$
		$[C] \wedge [\neg C] = \mathbb{O}$

Table 1: From Signal operators to boolean equations

Free conditions: Due to some distributivity properties of the operators *when*, *default*, *and* (on boolean-valued signals) and *or*, the expressions *when C* and *when*(*not C*) can be rewritten as intersection and union of clocks. If a clock *when C* cannot be rewritten, the boolean C is said to be a *free condition*. The rewriting rules are not enumerated here; they can be found in [5].

Example:

If A and B are two synchronous boolean signals, the boolean signal $C = A\ and\ B$ is not a free condition for, the clocks *when C* and *when* (*not C*) can be rewritten as:

$$\begin{array}{rcl} when\ C & = & (when\ A)\ \wedge\ (when\ B) \\ when(not\ C) & = & (when(not\ A))\ \vee\ (when(not\ B)) \end{array}$$

The boolean signal $C := (X = 0)$ where X is an integer signal is a free condition for no boolean rule apply to this expression.

In the partitions $([C], [\neg C])$ we will consider in the sequel, C is supposed to be free; the conditions that are not free are rewritten.

3.3 Conditional Dependency Graph

A data dependency is associated with each process. A process is compiled into a graph representing the dependency between signals. The excerpt of graph

$$X \xrightarrow{h} Y$$

means that at each instant of the clock h, Y's value depends on X's value. Such a graph is constructed from the elementary processes as follows:

- $X := f(X_1, \ldots, X_n)$ is associated to the graph

$$\forall i = 1 \ldots n,\ X_i \xrightarrow{\widehat{X}} X$$

- $X := U\ when\ C$ is associated to the graph

$$U \xrightarrow{\widehat{X}} X \text{ where } \widehat{X} = \widehat{U} \cap [C]$$

- $X := U\ default\ V$ is associated to the graph

$$U \xrightarrow{\widehat{U}} X \xleftarrow{\widehat{V} \setminus \widehat{U}} V$$

- For each signal X there is the dependency

$$\widehat{X} \longrightarrow X$$

  which means that its clock must be evaluated first to check if the expression of X is to be computed.

Sequential code generation from that graph follows a very simple scheme. For example a sequential code for the process $X := U\ default\ V$ is:

```
if present(X) then
        if present(U) then
                $X := U$
        else
                $X := V$
        endif
endif
```

Note: We write *if present*(X) to test if the signal X is present; in other words, to test if the instant under consideration is an element of $\widehat{X}$.

Full details on the code generation process can be found in [27]

3.4 Resolution of the system of equations: objectives

From the conditional dependency graph and the code generation scheme, we can figure out what the needs are in terms of resolution.

At any given instant, before the value of a signal X is computed, a test is made on the presence/absence of X; that is the presence/absence of its clock $\widehat{X}$. So there is a need for a resolution method that will allow to efficiently check the presence of a clock. The choice made in the SIGNAL compiler is a *triangularization*; that is, a transformation of the system of equations into a set of equalities of the form $h_i = h_{i1}{<}op{>}h_{i2}$ such that the clock-to-clock dependency graph — the edges $h_{i_1} \longrightarrow h_i \longleftarrow h_{i_2}$ — be an acyclic graph.

3.5 Strategy of resolution

A triangular system of equations is progressively constructed from the original system (see table 1). An equation of the form $h = h_1{<}op{>}h_2$ is oriented (we note $h := h_1{<}op{>}h_2$) in order to consider the clock formula $h_1{<}op{>}h_2$ as the definition of the clock variable h. During this process, an orientation of some equations may not be trivially possible. There are 3 reasons for that.

1. The equation under consideration is of the form $h = h_1{<}op{>}h_2$ but there is already a definition of the variable h. In this case, a rewriting can be performed to verify that the formula $h_1{<}op{>}h_2$ is equivalent to the previous definition of h.

2. It is an equation of the form $h = h_1{<}op{>}h_2$ but an orientation would induce a cycle in the clock-to-clock dependency graph. In this case, an attempt can be made to rewrite the formula $h_1{<}op{>}h_2$ and break the cycle.

3. It is an equation of the form $h_1{<}op{>}h_2 = k_1{<}op{>}k_2$: none of the parts are variables. In this case, an attempt can be made to prove the equivalence of the formulas with rewriting techniques.

At the end of this process the program is said to be temporally incorrect if there are some non-proved equalities, or if there are some equations whose orientation induces a cycle. Hence there is a need of an accurate automatic rewriting system to avoid the rejection of programs that may have been accepted if the rewriting was carried out manually.

In [5] an implementation of this strategy is proposed; it is based on an arborescent representation of the equations and a sum-of-product normal form. As it will be shown in the following paragraph, the arborescent representation exhibits the definitions and the equivalence of two formulas is proved by checking the equality of their normal forms.

3.6 Hierarchical representation of the equations

To meet the requirements presented above, an arborescent organization of the formulas has been defined in [5]. It speeds up the rewritings and it captures the triangularity of the system of equations. We give here the main ideas.

3.6.1 Partition tree

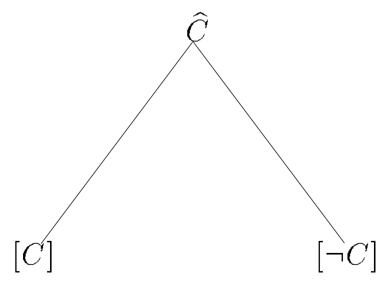


Figure 5: Basic partition tree

Let C be a free condition and $\widehat{C}$ its clock. According to the properties described in the previous sections, the pair $([C], [\neg C])$ is a partition of $\widehat{C}$ and satisfies the equations:

$$\left\{ \begin{array}{lcl} [C] \vee [\neg C] & = & \widehat{C} \\ [C] \wedge [\neg C] & = & \mathbb{O} \text{ (the null clock, the empty set of instants)} \end{array} \right. \quad (1)$$

Such a partition is represented by the tree on figure 5.

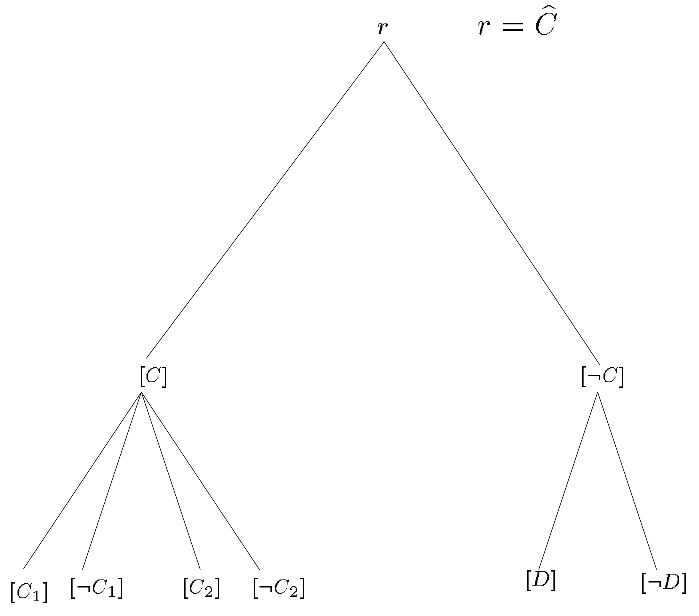


Figure 6: A hierarchical partitioning

Since any clock can be partitioned by a free condition, this basic tree can grow as its nodes are partitioned. For example on figure 6, $([C_1], [\neg C_1])$ is a partition of the clock $[C]$. The root may be an arbitrary formula but the internal nodes are partitions.

3.6.2 Forest of clocks

As any clock formula can be partitioned, the formulas originating from a SIGNAL program can be grouped into partition trees; this set of trees is called a *forest of clocks.* Within this forest, some trees may be one-node trees; these are clocks that have not been partitioned in the original SIGNAL program.

3.6.3 Fusion of clock trees

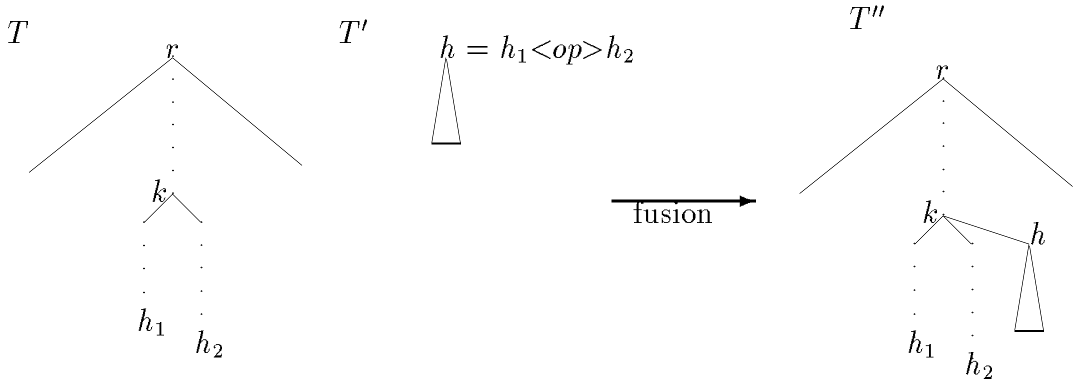


Figure 7: Fusion of trees

In the forest of clocks, let T and T' be 2 trees with roots r and h such that:

- h be defined by a formula $h_1{<}op{>}h_2$
- h_1 and h_2 belong to the tree T

that is the operand of the root h are in the tree T. We carry out a *fusion* of T' into T by inserting h into the tree T as depicted on figure 7. On figure 7 we point out a particular node k of the tree T. k is the *branching* of the nodes h_1 and h_2; it is the first common ancestor of the 2 nodes. T' is now a subtree of the merge tree T''. The fusion of 2 partition trees yields a more general tree we call *clock tree.*

The main idea of the insertion of the formula h_1<*op*>h_2 is that it is inserted under the branching of its operands, at the "right hand side"; we give more formal definitions further. This insertion procedure has two interesting features:

- it preserves the triangularity of the system of equations
- it optimizes the code generated by nesting *if-then-else* control structures.

3.6.3.1 Triangularity preservation

During a depth first search (dfs) of the tree T'' "from left to right", the nodes h_1 and h_2 are visited before the node $h = h_1$<*op*>h_2; it means that the ordering that makes the system be triangular is embodied in the tree.

3.6.3.2 Code optimization

A partition tree can be viewed as the representation of an inclusion relation. As a matter of fact, the first partition equation $[C] \vee [\neg C] = \widehat{C}$ implies that $[C] \subseteq \widehat{C}$ and $[\neg C] \subseteq \widehat{C}$. Hence, in a partition tree, a node is included in its parent. And more generally, a node is included in its ancestors.

On figure 7 the clock $h = h_1$<*op*>h_2 is included in the clock k. As a matter of fact, k being an ancestor of both h_1 and h_2, we have $h_1 \subseteq k$ and $h_2 \subseteq k$. Consequently all of the 3 formulas $h_1 \vee h_2$, $h_1 \wedge h_2$ and $h_1 \setminus h_2$ — denoted h_1<*op*>h_2 — are included in k. That is, the clock tree resulting from a fusion of 2 trees represents an inclusion relation.

The nesting of *if-then-else* structures for code optimization is based on the remark that, if h and k are 2 clocks such that $h \subseteq k$, then for an instant t, the following implication holds:

$$t \notin k \Longrightarrow t \notin h$$

In other words, if the test $t \in k$ fails, there is no need to test if $t \in h$. Thus, code generation can take advantage of the inclusion relation between clocks.

As an example, this code

```
if present(k) then
    do-something-k
    if present(h) then
        do-something-h
    endif
endif
```

is more efficient than this one

```
if present(k) then
    do-something-k
endif
if present(h) then
    do-something-h
endif
```

3.6.4 Arborescent resolution

We give here in three steps the algorithm of resolution.

1. Take a tree T' in the forest and attempt to rewrite its root h in a way that will make the operands of h belong to the same tree T. If this succeeds, the root formula of T' can be inserted into T as described in 3.6.3 without disturbing the triangularity of T.

2. Realize the fusion of T and T' to yield a tree T'' as described above.

   After the fusion of T and T' , a formula which had one operand in T and the other one in T' now has its 2 operands in the tree T''. So, the fusion of T and T' may lead to more fusions; that is the purpose of step 3.

3. Do step 1 and step 2 till the rewriting rules of step 1 no longer apply.

Step 1 is implemented using a notion of *p-depth* resolution thoroughly presented in [5]. To put it roughly, the user of the compiler can set an integer parameter p; this parameter is the maximum depth of the syntactic trees of the formulas manipulated during the rewriting. Setting a limit to the formulas, solves the duration and termination problems commonly encountered in rewriting systems.

During step 2, a formula is given a sum-of-products normal form before it is inserted in the tree. A normal form may fail to equate two equivalent formulas, leading to the rejection of a correct SIGNAL program. So there is a need of an canonical form to make the resolution more accurate.

The major purpose of this report is to show a canonical form that yields an optimal code as well.

Before the presentation of the arborescent canonical form, we intuively link our modelling with $\mathbb{Z}/3\mathbb{Z}$ which is the advocated modelling presented in [3].

3.6.5 Intuitive relation with $\mathbb{Z}/3\mathbb{Z}$

We have indicated (section 3.1) that the coding of temporal relations of SIGNAL could be modelled in $\mathbb{Z}/3\mathbb{Z}$. We link here intuitively this representation with the representation presented in this report.

Let E be a clock expression built on the boolean signals $C_1, ..., C_n$ of a clock tree; such an expression can be factorized in the following manner:

$$(E \setminus \widehat{C_1}) \vee \widehat{C_1} \wedge (([C_1] \wedge E_1) \vee ([\neg C_1] \wedge E_2))$$

where the expressions $E_1, E_2, E \setminus \widehat{C_1}$ do not involve C_1. In this factorization, the three possible states of a boolean signal clearly appear. On the one hand, one can in this expression substitute the clock $\widehat{C_1}$ by its definition. This definition, by definition of a clock tree, is built on conditions that are not in the sub-tree of root $\widehat{C_1}$. On the other hand, one can eliminate the set difference operator $(\setminus)$ using the following classical rules of boolean lattice :

$$E \setminus E = \mathbb{O}$$

$$E \setminus (H \wedge G) = (E \setminus H) \vee (E \setminus G)$$
$$E \setminus (H \vee G) = (E \setminus H) \wedge (E \setminus G)$$

and the rule induced by the downsampling

$$E \setminus [C] = (E \wedge [\neg C]) \vee E \setminus \widehat{C}$$

This method, when applied to the others conditions $C_2, ...C_n$, allows to express any clock expression with the operators $\vee$, $\wedge$ and downsamplings by condition $([C], [\neg C])$. Since the absence of a signal can be expressed in terms of presence of other signals, only two values (presence with the value "true", presence with the value "false") are used to express any clock expression.

3.6.6 Arborescent canonical form

3.6.6.1 Insertion = Factorization

We show informally that the insertion of a formula into a tree is a factorization.

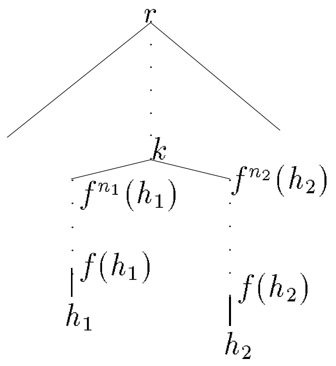


Figure 8: Factorization

Let us start with a partition tree (see figure 8). The parent of a node h_i is denoted $f(h_i)$. Since a node is included in its parent, $(h_i \subseteq f(h_i))$, we have

$$h_i = h_i \wedge f(h_i)$$

Recursively applying that equality to h_1 and h_2 up to their branching k yields

$$\begin{array}{l} h_1 = \overbrace{h_1 \wedge f(h_1) \wedge f^2(h_1) \ldots \wedge f^{n_1}(h_1)}^{F_1} \wedge k \\ h_2 = \underbrace{h_2 \wedge f(h_2) \wedge f^2(h_2) \ldots \wedge f^{n_2}(h_2)}_{F_2} \wedge k \end{array}$$

A clock h defined by a formula $h_1{<}op{>}h_2$ can be written:

$$h = h_1{<}op{>}h_2 = F \wedge k$$

where $F = F_1{<}op{>}F_2$, for, the operator $\wedge$ distributes over all of the operators $\wedge$, $\vee$ and $\backslash$.

Because of the way in which the fusion of trees is carried out, all the clock variables involved in the term F are descendants of k. This is a kind of "locality property" of the factor F.

As it will be shown in the following example, the factorization may depend on the rewriting of the expression of h.

3.6.6.2 Example

Consider the tree on figure 6 and consider the formulas

$$\begin{array}{l} h_1 = [C_1] \vee [D] \\ h_2 = [C_2] \vee [\neg D] \end{array}$$

Following the idea that a formula is inserted under the branching of its operands, the insertion of h_1 and h_2 yields the tree depicted on figure 9.

Now consider the formula $h = h_1 \wedge h_2$. The same argument would trivially place h as a child of r (see figure 10). But h can be rewritten into another expression.

Indeed, developing h yields:

$$\begin{array}{rl} h = & h_1 \wedge h_2 \\ = & ([C_1] \vee [D]) \wedge ([C_2] \vee [\neg D]) \\ = & ([C_1] \wedge [C_2]) \;\vee\; ([C_1] \wedge [\neg D]) \;\vee\; ([C_2] \wedge [D]) \;\vee\; ([D] \wedge [\neg D]) \end{array}$$

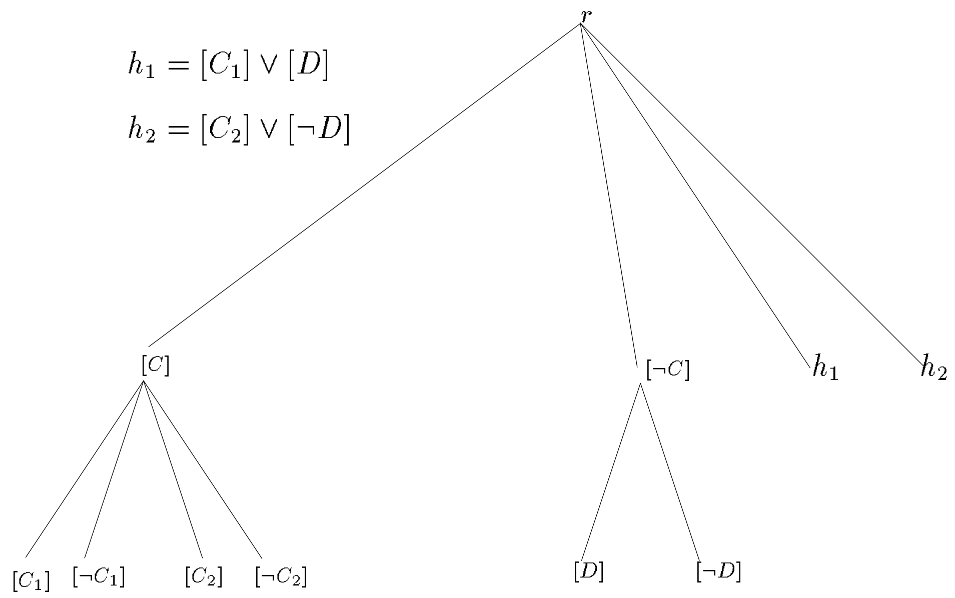


Figure 9: insertion of formulas

$[D]$ and $[\neg D]$ being a partition of $[\neg C]$, we have $[D] \wedge [\neg D] = \mathbb{O}$. Thus

$$h = ([C_1] \wedge [C_2]) \ \vee \ ([C_1] \wedge [\neg D]) \ \vee \ ([C_2] \wedge [D])$$

Using the inclusion relations $\left\{ \begin{array}{rl} [C_1] = & [C_1] \wedge [C] \\ [C_2] = & [C_2] \wedge [C] \\ [D] = & [D] \wedge [\neg C] \end{array} \right.$ yields:

$$h = ([C_1] \wedge [C_2]) \ \vee \ ([C_1] \wedge [C] \wedge [\neg D] \wedge [\neg C]) \ \vee \ ([C_2] \wedge [C] \wedge [D] \wedge [\neg C])$$

Finally, $[C]$ and $[\neg C]$ being a partition of r, we have $[C] \wedge [\neg C] = \mathbb{O}$, which implies that:

$$h = [C_1] \wedge [C_2]$$

The branching of $[C_1]$ and $[C_2]$ being $[C]$, h can be placed under $[C]$ (see figure 10).

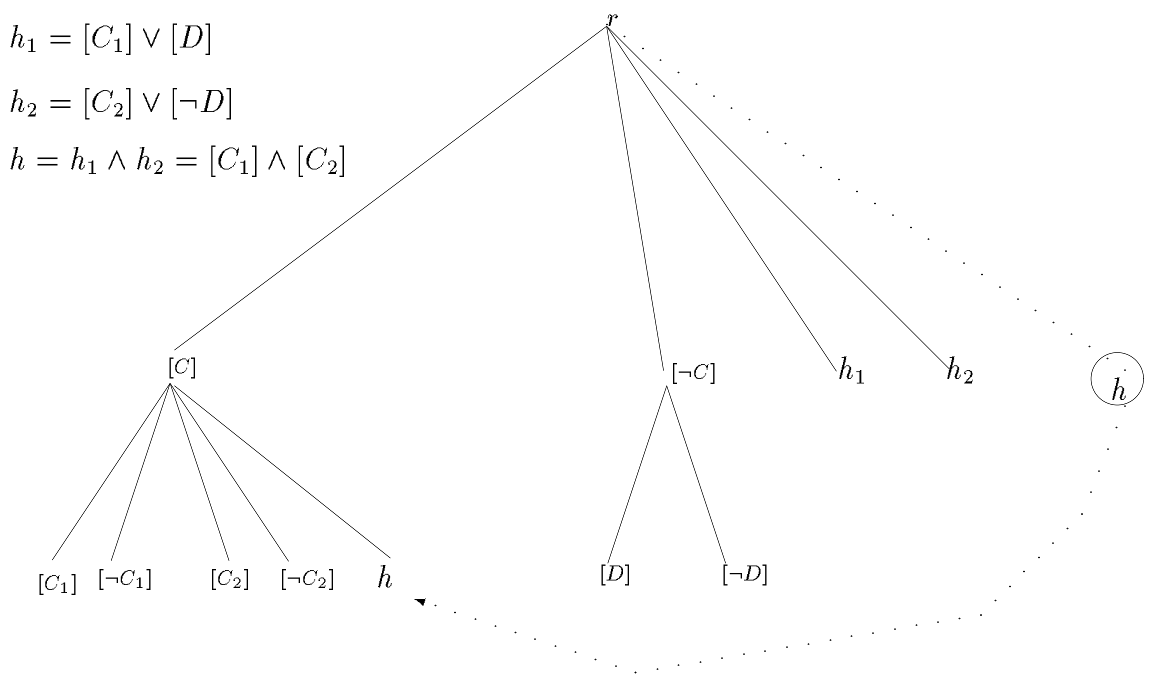


Figure 10: Best factorization

As the code generation is based on *if-then-else* nesting, the insertion of h under $[C]$ yields a much more efficient code.

3.6.6.3 Outline of the remainder

It appears from the previous example that a canonical form of the clocks must also seek to place formulas as deeper as possible. The canonical form we propose in this report, satisfies that requirement.

The next section introduces formally a data structure called *hierarchy*. A hierarchy is a tree which may seem at first sight very artificial. But it is endowed with features that mimic trees of clocks. Afterwards, we show that a node in a hierarchy can be associated in a one-to-one manner with a clock formula.

4 Hierarchies

A hierarchy is a tree whose nodes are decorated with boolean functions. Before giving a formal definition, let us introduce a canonical form of boolean functions and a formal definition of a tree.

4.1 Boolean functions

4.1.1 Definitions and notations

Let $B = \{0, 1\}$, and $(B, \cdot, +)$ be the boolean algebra. In some boolean terms the "$\cdot$" operator may be omitted. Let $V = \{x_1, \ldots, x_n\}$ be a set of $n > 0$ boolean variables.
A *boolean function* of the variables $x_1, \ldots, x_n$ is a map

$$\begin{array}{rccl} \psi & : & B^n & \longrightarrow & B \\ & & (x_1, \ldots, x_n) & \longmapsto & \psi(x_1, \ldots, x_n) \end{array}$$

We denote $\mathbf{1}$ the boolean function that maps any vector on the constant 1. Similarly, $\mathbf{0}$ denotes the boolean function that maps any vector on the constant 0.

We call *valuation*, a partial map that associates a value in B to a symbol in V.

For example $(x_1 = 0, x_3 = 1)$ is a valuation that involves only the variables x_1 and x_3.

When the valuation involves the whole set V, it is represented by the vector — $\in B^n$ — of the values. Throughout this report, when we reference a valuation, we will either enumerate the variables involved or use a vector notation ($\vec{v}$); the context will then precise the variables involved.

Let A be a subset of V and $\vec{v}$ be a valuation of the variables in A. We denote $\psi_{|\vec{v}}$ the *"restriction of ψ to $\vec{v}$"*; it is the function ψ in which the variables in A are replaced by their values.

Example: $\psi_{|x_i=0}(x_1,\ldots,x_n) = \psi(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n)$.

Note that $\psi_{|x_i=0}$ does not depend x_i any longer.

4.1.2 Shannon's expansion theorem

Let ψ be a boolean function of the variables $x_1,\ldots,x_n$. Shannon's expansion theorem around the variable x_i is given by

$$\psi = (\overline{x_i} \cdot \psi_{|x_i=0}) + (x_i \cdot \psi_{|x_i=1})$$

As mentioned above $\psi_{|x_i=0}$ and $\psi_{|x_i=1}$ do not depend on x_i any longer.

4.1.3 Support set

Given a boolean function ψ of the variables $x_1,\ldots,x_n$, we define a set called *support* as follows:

$$Supp(\psi) = \{x_i \text{ such that } \psi_{|x_i=0} \neq \psi_{|x_i=1}\}$$

$Supp(\psi)$ is the set of variables on which ψ depends.

We will use this immediate consequence of Shannon's theorem very frequently.

$$x_i \notin Supp(\psi) \overset{\text{def}}{\Longleftrightarrow} \psi_{|x_i=0} = \psi_{|x_i=1} \Longleftrightarrow \psi = \psi_{|x_i=0} = \psi_{|x_i=1} \qquad (2)$$

To put it another way, "ψ does not depend on x_i" if and only if all the three functions $\psi_{|x_i=0}$, $\psi_{|x_i=1}$ and ψ are equal.

4.1.4 Construction of BDDs

The Binary Decision Diagrams (BDD) technique has been presented by Bryant [10] as an efficient (both in time and space) data structure for the representation of boolean functions. It has been widely used in the field of hardware verification: verification of combinational circuits [10, 7] and verification of sequential circuits [30, 14]. A variant called TDD for Ternary Decision Diagrams has been introduced in [17, 8] as a canonical form of $\mathcal{Z}/3\mathcal{Z}$ polynomials.

Consider a boolean function ψ of the variables $x_1, \ldots, x_n$. Assume there is a total order $\leq$ on the variables e.g $x_1 \leq x_2 \leq \ldots \leq x_n$. A BDD of the function ψ with respect to the ordering $\leq$ is a directed acyclic graph that represents ψ. The construction of a BDD is based on applications of Shannon's expansion theorem successively around the variable x_1, then x_2 thru x_n. First, Shannon's theorem is applied to ψ around x_1 yielding two functions $\psi_{|x_1=0}$ and $\psi_{|x_1=1}$. Then it is applied to $\psi_{|x_1=0}$ and $\psi_{|x_1=1}$ around the variable x_2,...

The precise algorithms to construct BDDs are not relevant for what follows. In this report, BDDs are interesting mostly because of their canonicity property summarized in the following theorem.

Theorem 1 *Let ψ be a boolean function of the variables $x_1, \ldots, x_n$. Given a total order $\leq$ on the variables e.g $x_1 \leq x_2 \leq \ldots \leq x_n$, ψ has a unique* BDD *representation.*

A proof can be found in [10].

This allows us throughout this paper, to identify a boolean function with its BDD and to apply to BDDs, the same operators as to boolean functions.

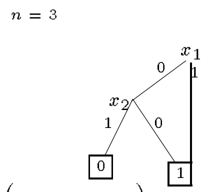
4.1.5 Example

Figure 11 shows some BDDs and one interesting feature. For any function ψ, the variables in $Supp(\psi)$ are the only ones that appear in its BDD.

4.2 Tree: formal definition

A tree is a pair (V, f) where

- V is a non-empty set



$\psi(x_1, x_2, x_3) = x_1 + \overline{x_2}$
$Supp(\psi) = \{x_1, x_2\}$

$\boxed{0}$
$\psi(x_1, x_2, x_3) = 0$
$Supp(\psi) = \emptyset$

$\boxed{1}$
$\psi(x_1, x_2, x_3) = 1$
$Supp(\psi) = \emptyset$

Figure 11: Support sets of some BDDs

- $f : V \longmapsto V$ is a map called *filiation function.* f satisfies

$$\exists r \in V, \forall x \in V, \exists n \in I\!N, r = f^n(x) \ and \ f(r) = r$$

It can be easily shown that such an element r is unique. r is the *root.*

A tree may sometimes be denoted (V, f, r) to point out its root.

Terminology : Given a tree (V, f, r) , a node $x \in V$, we define here some terms and objects we will frequently use.

- $f(x)$ is the *parent* of the node x
- $Ch(x) = \{y \in V / f(y) = x\} \setminus \{x\}$ is the set of the children of x; x has been subtracted from $Ch(x)$ so that the root r do not belong to $Ch(r)$;
- 2 distinct elements of a set $Ch(x)$ are called *siblings*
- $Anc(x) = \{f^n(x), n \in I\!N\} \setminus \{x\}$ is the set of the ancestors of x,
- $Des(x) = \{y \in V / \exists n \in I\!N, f^n(y) = x\} \setminus \{x\}$ the set of the descendants of x,
- $d(x) = Min\{n \in I\!N / f^n(x) = r\}$ is the depth of x.

4.3 Total order on a tree

An arborescent structure naturally defines a partial order $\preceq$ on the nodes; it is the reflexive and transitive closure of the relation $\mathcal{R}$:

$$x \mathcal{R} y \Longleftrightarrow x = f(y)$$

$\preceq$ is a partial order because two siblings are incomparable. So, if we choose an arbitrary total order $\mathcal{R}_x$ of the children of each node x, a total order $\leq$ can be put on the tree by a preorder traversal, i.e a depth first search — *dfs* — in which a node is visited before its children. Throughout this report, each time a total order of a tree is needed, it is defined by a family of total orders on the children of each node.

When a tree is drawn, $\mathcal{R}_x$ could be the enumeration of the children of x from left to right. As an example, a preorder traversal of the tree on figure 10 would yield the order:

$$r, [C], [C_1], [\neg C_1], [C_2], [\neg C_2], h, [\neg C], [D], [\neg D], h_1, h_2$$

The key features of the total order $\leq$ are the following properties :

$$\begin{cases} \forall x \in V, f(x) \leq x \\ \forall x \in V, \forall x_1, x_2 \in Ch(x),\ x_1 \neq x_2, x_1 \leq x_2 \Longrightarrow \forall y \in Des(x_1), y \leq x_2 \end{cases} \tag{3}$$

These properties are depicted on figure 12.
The first one comes from the fact that, a node x is always visited after its parent $f(x)$.

In the second one, x_1 and x_2 are distinct siblings, with x as their parent, such that $x_1 \leq x_2$; on $Ch(x)$, $\leq$ coincides with $\mathcal{R}_x$. Thus, $x_1 \mathcal{R}_x x_2$; and consequently x_1's descendants are visited before x_2.

4.4 Hierarchy: definition

A hierarchy is a tuple $(V, f, r, \leq, \mathcal{B})$ where:

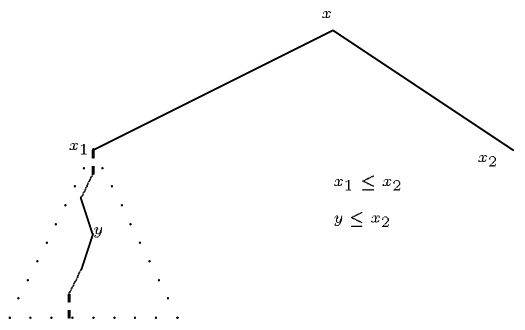


Figure 12: Order on the nodes

- V is a finite non-empty set; the elements of V are considered as boolean variables
- (V, f, r) is a tree
- $\leq$ is a total order on the tree and satisfies the dfs properties described above
- $\mathcal{B}$ is map that associates to each element of V, a boolean function on the variables of V; the total order $\leq$ on V allows to consider these boolean functions as BDDs. $\mathcal{B}$ satisfies:
  - $\mathcal{B}(r) = \mathbf{1}$
  - the following properties we call *locality criterion*

$$\begin{cases} \forall y \in \mathit{Supp}(\mathcal{B}(x)), y \leq x \\ \mathit{Supp}(\mathcal{B}(x)) \subseteq \mathit{Des}(f(x)) \end{cases} \tag{4}$$

To put it another way, $\mathit{Supp}(\mathcal{B}(x)) \subseteq \mathit{Des}(f(x))$ means that the node x is decorated with a BDD whose variables are descendants of $f(x)$, the parent of x. And, $\forall y \in \mathit{Supp}(\mathcal{B}(x)), y \leq x$ means that during a dfs, the node x is visited after the variables involved in $\mathcal{B}(x)$

Intuitive relation with clocks
We require $\mathcal{B}$ to satisfy the properties (4) in order to mimic the features of clock trees. The properties of $\mathcal{B}$ capture two features of clock trees:

- The insertion of a clock formula in a tree of clocks (see 3.6.3) was done in such a way that a dfs visit a formula after its operands; we called it *triangularity preservation*
- In section 3.6.6.1, we showed that a clock formula h inserted under a clock k could be factorized $h = F \wedge k$ where F was a term whose variables were descendants of k. The requirement $Supp(\mathcal{B}(x)) \subseteq Des(f(x))$ is designed to be a formalization of the requirement on F.

4.5 Sub-hierarchy

The locality criterion on $\mathcal{B}$ allows to view each sub-tree of a hierarchy as a hierarchy.

Consider a node v in a hierarchy $(V, f, r, \leq, \mathcal{B})$. The sub-tree with root v — the set of nodes $\{v\} \cup Desc(v)$ — can be very simply endowed with a hierarchy structure as follows:

- the filiation function f_v :

$$f_v(x) = \begin{cases} v \; if \; x = v \\ f(x) \; if \; x \in Desc(v) \end{cases}$$

- the total order $\leq_v$ is inherited from $\leq$
- the map $\mathcal{B}_v$ is defined by :

$$\mathcal{B}_v(x) = \begin{cases} \mathbf{1} \; if \; x = v \\ \mathcal{B}(x) \; if \; x \in Desc(v) \end{cases}$$

4.6 Fusion of hierarchies

In the previous paragraph, we showed that in a hierarchy, each sub-tree can be viewed as a hierarchy. Conversely, consider two hierarchies $(V_1, f_1, r_1, \leq_1, \mathcal{B}_1)$ and $(V_2, f_2, r_2, \leq_2, \mathcal{B}_2)$ such that $V_1 \cap V_2 = \emptyset$; that is, they are made up with disjoint sets of boolean variables. Let $v \in V_1$ and consider the tree graphically defined on figure 13.

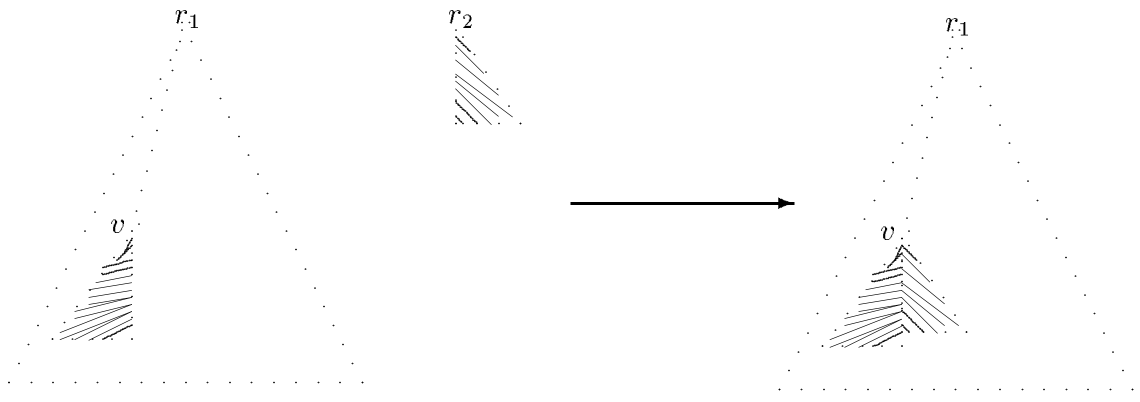


Figure 13: fusion of hierarchies

Informally, r_2 is dropped out; the set of v's children is augmented with the children of r_2; consequently, the sub-tree of v is augmented with r_2's descendants.

More formally, we have a new hierarchy $(V, f, r, \leq, \mathcal{B})$ defined by:

- $V = (V_1 \cup V_2) \setminus \{r_2\}$
- the filiation function

$$f(x) = \begin{cases} f_1(x) \ if \ x \in V_1 \\ f_2(x) \ if \ x \in V_2 \ and \ f_2(x) \neq r_2 \\ v \ if \ x \in V_2 \ and \ f_2(x) = r_2 \end{cases}$$

- the total order $\leq$ is obtained from a dfs when the new set $Ch(v)$ of v's children is ordered in such a way that the original children of v be less than the children of r_2. That is, if x_1 is the greatest child of v with respect to $\leq_1$, and if x_2 is the least child of r_2 with respect to $\leq_2$, we choose the ordering $x_1 \leq x_2$. By transitivity, it yields a total order on the new set of children $Ch(v)$ and a dfs yields a total order on the whole new tree.

- the map $\mathcal{B}$ is defined on V as follows:

$$\mathcal{B}(x) = \begin{cases} \mathcal{B}_1(x) \ if \ x \in V_1 \\ \mathcal{B}_2(x) \ if \ x \in V_2 \end{cases}$$

We say that V *is the fusion of* V_2 *into* V_1 *through the node* v *of* V_1.

Intuitive relation with clocks
The fusion of hierarchies formalizes and precises the fusion of clock trees described in 3.6.3. If in the hierarchy $(V_1, f_1, r_1, \leq_1, \mathcal{B}_1)$, the node v has no children, a picture of the fusion will be the figure 7.

4.7 Upward expansion in a hierarchy

Given a hierarchy $(V, f, r, \leq, \mathcal{B})$, we define a map $\mathcal{L}$ that associates a BDD $\mathcal{L}(x)$ to a node $x \in V$ as follows:

$$\mathcal{L}(x) = \mathcal{B}(x) \cdot \mathcal{B}(f(x)) \cdot \mathcal{B}(f^2(x)) \ldots \mathcal{B}(r) = \prod_{i=0}^{d(x)} \mathcal{B}(f^i(x))$$

In short,

$$\begin{cases} \mathcal{L}(x) = \mathcal{B}(x)\mathcal{L}(f(x)) \\ \mathcal{L}(r) = \mathbf{1} \end{cases}$$

Intuitive relation to clocks
In lines to come, we will show a link between the expression

$$\mathcal{L}(x) = \mathcal{B}(x) \cdot \mathcal{B}(f(x)) \cdot \mathcal{B}(f^2(x)) \ldots \mathcal{B}(r)$$

and the expression

$$h_1 = h_1 \wedge f(h_1) \wedge f^2(h_1) \ldots \wedge f^{n_1}(h_1) \wedge k$$

we introduced in 3.6.6.1.

4.8 Canonical factorization

Given a hierarchy $(V, f, r, \leq, \mathcal{B})$ and 2 nodes x_1 and x_2. Let the BDD $G = \mathcal{L}(x_1)\text{<}op\text{>}\mathcal{L}(x_2)$ denote one of the three terms $\mathcal{L}(x_1) \cdot \mathcal{L}(x_2)$, $\mathcal{L}(x_1) + \mathcal{L}(x_2)$, $\mathcal{L}(x_1) \cdot \overline{\mathcal{L}(x_2)}$.

We are going to show that, when $G \neq \mathbf{0}$ and $G \neq \mathbf{1}$, there exists a unique node $p \in V$ and a unique BDD G' which satisfy

1.
$$\left\{ \begin{array}{l} G = \mathcal{L}(p) \cdot G' \\ Supp(G') \neq \emptyset \\ Supp(G') \subseteq Des(p) \end{array} \right. \tag{5}$$

2. p has a maximal depth; that is, the depth of any node $p' \in V$ which satisfies (5) is less than the depth of p — $d(p') \leq d(p)$.

Intuitive relation with clocks
Recall that in section 3.6.6.1, for a clock formula $h = h_1\text{<}op\text{>}h_2$, we trivially had a factorization $h = F \wedge k$ and the operands of the term F were descendants of k.
With the proper links between clocks and hierarchies, this proposition asserts the existence of the "best factorization", in the sense that the parent will have the greatest possible depth.

In the following lines, we prove that statement in three steps:

- the existence of p and G' is shown; its quite straightforward
- we define very simply $Pot(G)$, a — totally ordered — subset of V, and we show that its greatest element is a convenient p: hence the uniqueness of p
- we finally show the uniqueness the BDD G' by giving its expression.

4.8.1 Intermediate results

Let G denote one of the 3 BDDs $\mathcal{L}(x_1) \cdot \mathcal{L}(x_2)$, $\mathcal{L}(x_1) + \mathcal{L}(x_2)$, $\mathcal{L}(x_1) \cdot \overline{\mathcal{L}(x_2)}$. We will sometimes use the notation $\mathcal{L}(x_1)\text{<}op\text{>}\mathcal{L}(x_2)$.

Suppose $G \neq \mathbf{1}$ and $G \neq \mathbf{0}$.

Consider the set $Pot(G)$ of the nodes that can "potentially" satisfy the proposition. That is,

$$\begin{array}{rcll} Pot(G) & = & \{p \in V \;/\; \exists \text{ } a \text{ BDD } G' \text{ such that} & \mathcal{L}(p) \cdot G' = G \\ & & & Supp(G') \neq \emptyset \\ & & & Supp(G') \subseteq Des(p)\} \end{array}$$

The aim is to prove that there is in $Pot(G)$ a unique element which has the greatest depth and that the associated G' is also unique.

Here is a property of BDDs we will use very frequently.

Lemma 1 *Let B and B' be 2 BDDs; let A be one of the 3 BDDs $B+B'$, $B \cdot B'$ or $B \cdot \overline{B'}$. Then*

$$Supp(A) \subseteq Supp(B) \cup Supp(B')$$

This lemma is quite trivial; it means that when 2 boolean functions B and B' are composed with a boolean operator <*op*>, the variables involved in the resulting function A, can be found in B or B'. The proof is straightforward.
$\square$

Here are some features of the set $Pot(G)$.

Proposition 1 *The root r is in $Pot(G)$.*

Proof:

We know from the definition of $\mathcal{L}$ that $\mathcal{L}(r) = \mathbf{1}$ and that

$$\mathcal{L}(x_1) = \mathcal{B}(x_1) \cdot \mathcal{L}(f(x_1))$$

As $r = f^{d(x_1)}(x_1)$, $d(x_1)$ being the depth of x_1, we have:

$$\mathcal{L}(x_1) = \mathcal{L}(r) \cdot \prod_{i=0}^{d(x_1)-1} \mathcal{B}(f^i(x_1))$$

Similarly

$$\mathcal{L}(x_2) = \mathcal{L}(r) \cdot \prod_{i=0}^{d(x_2)-1} \mathcal{B}(f^i(x_2))$$

Hence

$$\begin{aligned} G &= \mathcal{L}(x_1) \texttt{<}op\texttt{>} \mathcal{L}(x_2) \\ &= \mathcal{L}(r) \cdot \underbrace{\left(\prod_{i=0}^{d(x_1)-1} \mathcal{B}(f^i(x_1)) \texttt{<}op\texttt{>} \prod_{i=0}^{d(x_2)-1} \mathcal{B}(f^i(x_2)) \right)}_{G'} \\ &= \mathcal{L}(r) \cdot G' \end{aligned}$$

- $Supp(G') \neq \emptyset$ for, if $Supp(G') = \emptyset$, we would have $G' = \mathbf{0}$ or $G' = \mathbf{1}$; which means that $G = \mathbf{0}$ or $G = \mathbf{1}$: these cases have been excluded by hypothesis.
- $Supp(G') \subseteq Des(r)$ for:

  From lemma 1 we can write

  $$Supp(G') \subseteq \bigcup_{i=0}^{d(x_1)-1} Supp(\mathcal{B}(f^i(x_1))) \cup \bigcup_{i=0}^{d(x_2)-1} Supp(\mathcal{B}(f^i(x_2)))$$

  From the properties (4) of $\mathcal{B}$ — $\forall x \in V, Supp(\mathcal{B}(x)) \subseteq Des(f(x))$ — we can write

  $$Supp(G') \subseteq \bigcup_{i=0}^{d(x_1)-1} Des(f^{i+1}(x_1)) \cup \bigcup_{i=0}^{d(x_2)-1} Des(f^{i+1}(x_2))$$

A simple property of trees — $Des(f^i(x)) \subseteq Des(f^{i+1}(x))$ — allows to write

$$Supp(G') \subseteq Des(f^{d(x_1)}(x_1)) \cup Des(f^{d(x_2)}(x_2))$$

And finally from the definition of d the depth in a tree — $r = f^{d(x_1)}(x_1) = f^{d(x_2)}(x_2)$ — we write

$$Supp(G') \subseteq Des(r) \cup Des(r) = Des(r)$$

Conclusion: $r \in Pot(G)$.
□

Lemma 2 *Let p be an element of $Pot(G)$ and G' be the* BDD *specified in the definition of the set $Pot(G)$ i.e G' is such that*

$$\begin{cases} \mathcal{L}(p) \cdot G' = G \\ Supp(G') \neq \emptyset \\ Supp(G') \subseteq Des(p) \end{cases}$$

Then $\forall x \in Supp(\mathcal{L}(p)), x \leq p$

Proof:

From the definition of $\mathcal{L}$,

$$\mathcal{L}(p) = \prod_{i=0}^{d(p)} \mathcal{B}(f^i(p))$$

which implies — see lemma 1 — that

$$Supp(\mathcal{L}(p)) \subseteq \bigcup_{i=0}^{d(p)} Supp(\mathcal{B}(f^i(p)))$$

From (3) the properties of $\leq$ we have $f^i(p) \leq p$
and from (4) the properties of $\mathcal{B}$ we have

$$\forall x \in Supp(\mathcal{B}(f^i(p))), x \leq f^i(p)$$

consequently $x \leq p$ for all $x \in Supp(\mathcal{B}(f^i(p)))$

Since any $x \in Supp(\mathcal{L}(p))$ belongs to some $Supp(\mathcal{B}(f^i(p)))$ — lemma 1 — we have

$$\forall x \in Supp(\mathcal{L}(p)), x \leq p$$

□

Lemma 3 *Let p be an element of Pot(G) and G' be the* BDD *specified in the definition of the set Pot(G).*

$$\forall x \in Supp(G'), x > p$$

$>$ *is an abbreviation for not* $\leq$

Proof

As specified in the definition of $Pot(G)$, $Supp(G') \subseteq Des(k)$.
And from the definition

$$Des(p) = \{x \in V / \exists n \in I\!N, f^n(x) = p\} \setminus \{p\}$$

we deduce that

$$\forall x \in Des(p), p \leq x \text{ and } p \neq x$$

which means that

$$\forall x \in Des(p), x > p$$

Since $Supp(G') \subseteq Des(p)$ we have $\forall x \in Supp(G'), x > p$.

□.

Lemma 4 *Let p be an element of Pot(G) and G' be the* BDD *specified in the definition of the set Pot(G).*

$$Supp(\mathcal{L}(p)) \cap Supp(G') = \emptyset$$

Proof:

It is straightforward from the 2 previous lemmas.

$\square$.

Lemma 5 *Let p be an element of $Pot(G)$ and G' be the* BDD *specified in the definition of the set $Pot(G)$. Then*

$$Supp(G) = Supp(\mathcal{L}(p)) \cup Supp(G')$$

Proof:

- The inclusion $Supp(G) \subseteq Supp(\mathcal{L}(p)) \cup Supp(G')$ results from lemma 1.
- $Supp(G') \subseteq Supp(G)$?

  Let x be an element of $Supp(G')$ i.e $G'_{|x=0} \neq G'_{|x=1}$

  $\mathcal{L}(p) \neq \mathbf{0}$ for $G \neq \mathbf{0}$. Thus, there is a valuation $\vec{v}$ of the variables in $Supp(\mathcal{L}(p))$, such that $\mathcal{L}(p)_{|\vec{v}} = 1$

  Since $Supp(G') \cap Supp(\mathcal{L}(p)) = \emptyset$, we have $x \notin Supp(\mathcal{L}(p))$. Which means

  $$\mathcal{L}(p)_{|x=0} = \mathcal{L}(p)_{|x=1} = \mathcal{L}(p)$$

  Since $Supp(G') \cap Supp(\mathcal{L}(p)) = \emptyset$, $\vec{v}$ doesn't involve the variables in $Supp(G')$ and consequently

  $$G'_{|\vec{v}} = G'$$

  Now, suppose $x \notin Supp(G)$ i.e $G_{|x=0} = G_{|x=1}$. We would have:

  $$\begin{aligned} G_{|x=0,\vec{v}} &= G_{|x=0,\vec{v}} \\ \mathcal{L}(p)_{|x=0,\vec{v}} \wedge G'_{|x=0,\vec{v}} &= \mathcal{L}(p)_{|x=1,\vec{v}} \wedge G'_{|x=1,\vec{v}} \\ \mathcal{L}(p)_{|\vec{v}} \wedge G'_{|x=0} &= \mathcal{L}(p)_{|\vec{v}} \wedge G'_{|x=1} \\ 1 \wedge G'_{|x=0} &= 1 \wedge G'_{|x=1} \\ G'_{|x=0} &= G'_{|x=1} \end{aligned}$$

Which contradicts the hypothesis that $x \in Supp(G')$.

Then $x \in Supp(G)$ i.e $Supp(G') \subset Supp(G)$

- A similar argument proves that $Supp(\mathcal{L}(p)) \subset Supp(G)$.

From the previous points, we deduce $Supp(G) = Supp(G') \cup Supp(\mathcal{L}(p))$.
□

Proposition 2 *Let x_n be the greatest element of the set $Supp(G)$ i.e $x_n = Max(Supp(G))$. Then, any element of $Pot(G)$ is an ancestor of x_n i.e*

$$Pot(G) \subseteq Anc(x_n)$$

This theorem is very interesting because the set $Anc(x_n)$ is quite easy to compute.

Proof:

Let p be an element of $Pot(G)$, G' be the BDD specified in the definition of the set $Pot(G)$ and x_n be the greatest element in $Supp(G)$.

We know from lemma 5 that

$$Supp(G) = Supp(\mathcal{L}(p)) \cup Supp(G')$$

and from lemmas 2 and 3 that

$$\forall x \in Supp(\mathcal{L}(p)), \forall y \in Supp(G'), x \leq p < y$$

Consequently, the greatest element of $Supp(G)$ is in $Supp(G')$ i.e

$$x_n \in Supp(G')$$

Now, from the definition of Pot, $Supp(G') \subseteq Des(p)$. Then $x_n \in Des(p)$ and $p \in Anc(x_n)$.

□

Proposition 3 *There exists a unique element $p \in Pot(G)$ such that*

$$\forall q \in Pot(G), d(q) \leq_{I\!N} d(p)$$

$\leq_{I\!N}$ denotes the usual total order on the natural integers.

Proof:

The existence is trivial: $Pot(G)$ is a finite, totally ordered and non-empty set.

The uniqueness is due to the fact that the elements of $Anc(x_n)$ are ordered by their depths. As a matter of fact, let p_1 and p_2 be 2 elements of $Pot(G)$ such that

$$d(p_1) = d(p_2) = \max_{q \in Pot(G)} d(q)$$

Properties of trees allow to write:

$$\begin{aligned} p_1 &= f^{d(x_n)-d(p_1)}(x_n) \quad \text{for } p_1 \in Anc(x_n) \\ &= f^{d(x_n)-d(p_2)}(x_n) \quad \text{for } d(p_1) = d(p_2) \\ &= p_2 \quad \text{for } p_2 \in Anc(x_n) \end{aligned}$$

□

Proposition 4 *Let p be the element of $Pot(G)$ with the greatest depth. The* BDD *G' specified by the definition of $Pot(G)$ is unique.*

Proof:

Let $p \in Pot(G)$ be the element which has the greatest depth. There exists a BDD G' such that

$$\begin{cases} \mathcal{L}(p) \cdot G' = G \\ Supp(G') \neq \emptyset \\ Supp(G') \subseteq Des(p) \end{cases}$$

To show the uniqueness of G', we are going to give its expression.

Let $\vec{v}$ be a valuation of the variables in $Supp(\mathcal{L}(p))$ such that $\mathcal{L}(p)_{|\vec{v}} = 1$. We have

$$\begin{array}{rcl} G & = & \mathcal{L}(p) \cdot G' \\ G_{|\vec{v}} & = & \mathcal{L}(p)_{|\vec{v}} \cdot G'_{|\vec{v}} \\ & = & G'_{|\vec{v}} \\ & = & G' \ \ \textit{for } \vec{v} \textit{ doesn't involve the variables of } G' \end{array}$$

Hence the uniqueness of G'.

□

4.8.2 Canonical factorization theorem

Theorem 2 .
Given a hierarchy $(V, f, r, \leq, \mathcal{B})$ and 2 nodes x_1 and x_2. Let the BDD *$G = \mathcal{L}(x_1) \underline{<op>} \mathcal{L}(x_2)$ denote one of the three terms $\mathcal{L}(x_1) \cdot \mathcal{L}(x_2)$, $\mathcal{L}(x_1) + \mathcal{L}(x_2)$, $\mathcal{L}(x_1) \cdot \overline{\mathcal{L}(x_2)}$.*

There exists a unique node $p \in V$ and a unique BDD *G' which satisfy*

1.
$$\left\{ \begin{array}{l} G = \mathcal{L}(p) \cdot G' \\ Supp(G') \neq \emptyset \\ Supp(G') \subseteq Des(p) \end{array} \right. \qquad (6)$$

2. *p has a maximal depth; that is, the depth of any node $p' \in V$ which satisfies (6) is less than the depth of p — $d(p') \leq d(p)$.*

Proof:

The proof is given by propositions 3 and 4.

□

4.9 Recapitulation

In this section

- we defined a structure $(V, f, r, \leq, \mathcal{B})$ called *hierarchy*
- we defined a fusion operation which inserts a hierarchy H' as a sub-hierarchy of another hierarchy H
- we defined some boolean functions $\mathcal{L}(x)$ which have nice factorization properties.

In the following section, we define a hierarchy structure on clocks. Starting from partition trees, we perform the fusion operation. And as the hierarchy structure is preserved by the fusion, the process can be iterated.

5 From clocks to hierarchies

Consider a SIGNAL program and the forest of partition trees resulting from it. In this section we show that:

- with the adequate decorations, a partition tree can be viewed as a hierarchy
- the insertion of a formula in a partition tree can be carried out canonically through our factorization
- the fusion of partition trees described informally in 3.6.3 can be viewed as a fusion of hierarchies; since the fusion of 2 hierarchies yields a hierarchy, the fusion process can be iterated. Hence hierarchies are a formal framework for the construction of SIGNAL clock trees.

5.1 Clock variables and boolean variables

To make the distinction between clocks, considered as sets, and clocks viewed as booleans, we use a set of boolean variables and define a one-to-one map φ that associates a boolean variable $\varphi(h)$ to a clock variable h. Through the map φ a partition tree is transformed into a tree of boolean variables, hence preparing it for the definition of a hierarchy.

5.2 Ordering a partition tree

Let the pair $([C], [\neg C])$ be a partition of a clock $\widehat{C}$. The children $\varphi([C])$ and $\varphi([\neg C])$ of the node $\varphi(\widehat{C})$ are given the ordering $\varphi([C]) \leq \varphi([\neg C])$; we will justify this choice in lines to come.

If $([C], [\neg C])$ and $([C'], [\neg C'])$ are 2 distinct partitions of the same clock $\widehat{C}$ — that is, the signals C and C' have the same clock — , the boolean variables $\varphi([C])$, $\varphi([\neg C])$, $\varphi([C'])$ and $\varphi([\neg C])$ are children of $\varphi(\widehat{C})$. As said above, we have the ordering

$$\varphi([C]) \leq \varphi([\neg C]) \qquad \varphi([C']) \leq \varphi([\neg C'])$$

To make the order total, we *arbitrarily* choose

$$\varphi([\neg C]) \leq \varphi([C'])$$

Once the children of each node are totally ordered, a dfs yields a total order on the tree.

5.3 Mapping clocks on BDDs

We define here the map $\mathcal{B}$ that associate a BDD to each node — boolean variable — of a totally ordered tree.

In a partition tree, the root r may be defined by an arbitrary formula, but the internal nodes are partitions and appear in pairs $([C], [\neg C])$. On the tree (V, f), image — through φ — of a partition tree, we define $\mathcal{B}$ as follows:

- the root: $\mathcal{B}(\varphi(r)) = \mathbf{1}$
- $[C]:\ \mathcal{B}(\varphi([C])) = \varphi([C])$ the positive literal
- $[\neg C]:\ \mathcal{B}(\varphi([\neg C])) = \overline{\varphi([C])}$ the negative literal of $\varphi([C])$.

Note that the sets of boolean variables involved in $\mathcal{B}(\varphi([C]))$ and $\mathcal{B}(\varphi([\neg C]))$ — their support sets — are both equal to $\{\varphi([C])\}$. The ordering $\varphi([C]) \leq \varphi([\neg C])$ is justified by the fact that it allows $\mathcal{B}(\varphi([\neg C]))$ to satisfy the locality criterion (4) that we recall here:

$$\left\{\begin{array}{l} \forall y \in Supp(\mathcal{B}(x)), y \leq x \\ Supp(\mathcal{B}(x)) \subseteq Des(f(x)) \end{array}\right.$$

The ordering is especially needed for the first equation.

With the definitions of $\leq$ and $\mathcal{B}$, the image (V, f) of a partition tree is a hierarchy.

5.4 Bijection between clock formulas and boolean functions

Given a partition tree with root r, consider two clocks h_1 and h_2 of the tree. Consider the three formulas $h_1 \vee h_2$, $h_1 \wedge h_2$ and $h_1 \setminus h_2$.

With the upward expansion map $\mathcal{L}$ introduced in 4.7, we define the following correspondence:

	clock formulas	boolean functions
1	r the root	$\mathcal{L}(\varphi(r)) = \mathbf{1}$
2	$\mathbb{O}$ the null clock	$\mathcal{L}(\varphi(\mathbb{O})) = \mathbf{0}$
3	$h = h_1 \vee h_2$	$\mathcal{L}(\varphi(h_1)) + \mathcal{L}(\varphi(h_2))$
4	$h = h_1 \wedge h_2$	$\mathcal{L}(\varphi(h_1)) \cdot \mathcal{L}(\varphi(h_2))$
5	$h = h_1 \setminus h_2$ which is $h_1 \wedge (r \setminus h_2)$	$\mathcal{L}(\varphi(h_1)) \cdot \overline{\mathcal{L}(\varphi(h_2))}$

In a more mathematical framework we can show that $\mathcal{L}$ defines a bijective correspondence between clock formulas and the boolean functions that it associated to them; hence, $\mathcal{L}(\varphi(h))$ is truly a representation of the clock h. This is basically due to results in lattice theory. Precisions and details about the material presented here can be found in [15]. We give informally some justifications.

- The set of clocks — which are sets themselves — that make up a partition tree with root r can be viewed as a subset of a finite boolean lattice L.

- its supremum is r, the root of the tree
- its infimum is $\mathbb{O}$, the empty set of instants
- the upper bound operation is the union of 2 clocks: $h_1 \vee h_2$
- the lower bound operation is the intersection : $h_1 \wedge h_2$
- the complement of a clock h is $r \setminus h$

- Similarly, for a finite set of boolean variables, the set of the boolean functions that can be constructed with these variables is a finite boolean lattice.
  - its supremum is the constant boolean function $\mathbf{1}$
  - its infimum is the constant function $\mathbf{0}$
  - the upper bound operation is the *or* — denoted $+$ — of 2 functions
  - the lower bound operation is the *and* — denoted $\cdot$
  - the complementation is denoted $\_$

- $\mathcal{L}$ and φ define a map $\mathcal{L} \circ \varphi$ that preserves the suprema, the infima, the lattices operations (see the table above). This map is injective due to the properties of lattice homomorphisms. Hence $\mathcal{L} \circ \varphi$ is an isomorphism from L to $\mathcal{L} \circ \varphi(L)$.

  Since φ is defined to be a mere variable renaming, the interesting isomorphism is actually $\mathcal{L}$.

The aim of all this is to use $\mathcal{L}(\varphi(h_1))$ and $\mathcal{L}(\varphi(h_2))$ as the representations of h_1 and h_2 in order to perform the computation of h_1<*op*>h_2 — one of $h_1 \vee h_2$, $h_1 \wedge h_2$, $h_1 \setminus h_2$.

The image $\mathcal{L}(\varphi(h))$ of a clock variable h defined by some formula h_1<*op*>h_2 is called its *arborescent canonical form.*

Important remark

Let $([C], [\neg C])$ be a partition of a clock $\widehat{C}$. We defined $\mathcal{B}(\varphi([\neg C])) = \overline{\varphi([C])}$. We give here some justifications of that definition.

From the definition of $\mathcal{L}$, we have

$$\begin{aligned}
\mathcal{L}(\varphi([C])) &= \mathcal{L}(\varphi(\widehat{C})) \cdot \mathcal{B}(\varphi([C])) \\
&= \mathcal{L}(\varphi(\widehat{C})) \cdot \varphi([C])
\end{aligned}$$

Similarly,

$$\begin{aligned}
\mathcal{L}(\varphi([\neg C])) &= \mathcal{L}(\varphi(\widehat{C})) \cdot \mathcal{B}(\varphi([\neg C])) \\
&= \mathcal{L}(\varphi(\widehat{C})) \cdot \overline{\varphi([C])} \\
&= \mathcal{L}(\varphi(\widehat{C})) \cdot \overline{\mathcal{L}(\varphi([C]))} \text{ after some manipulations}
\end{aligned}$$

On the other hand,$[C]$, $[\neg C]$ and $\widehat{C}$ are related by the partition equations we recall here:

$$\left\{ \begin{array}{lcl} [C] \vee [\neg C] &=& \widehat{C} \\ [C] \wedge [\neg C] &=& \mathbb{O} \end{array} \right.$$

which means $[\neg C] = \widehat{C} \setminus [C]$.

Conclusion: $\mathcal{B}(\varphi([\neg C]))$ has been defined so that $\mathcal{L}(\varphi([\neg C]))$ be consistent with the general case $h_1 \setminus h_2$.

5.5 Canonical insertion

Consider a partition tree T with root r, and a partition tree T' with root r' such that the operands of r' be in the tree T; that is, r' is defined by some h_1<*op*>h_2 where both h_1 and h_2 belong to T. As described earlier in 3.6.3 a fusion of clock trees can be done. In this section, we transform it into a more formal fusion hierarchies that will be optimal.

The image $V = \varphi(T)$ is endowed with a hierarchy structure; that hierarchy is denoted $(V, f, r, \leq, \mathcal{B})$. Idem for $\varphi(T')$.

Consider the BDD $G = \mathcal{L}(\varphi(h_1)) {<}op{>} \mathcal{L}(\varphi(h_2))$ associated with $h_1 {<}op{>} h_2$. We showed that when G is not a constant boolean function it is factorized into

$$G = \mathcal{L}(p) \cdot G'$$

where

- p is a boolean variable in the hierarchy $\varphi(T)$; p is equal to $\varphi(k)$ for some — unique — clock k in T
- G' is a BDD and satisfies the locality criterion

In the following lines, we define a fusion of the hierarchies $\varphi(T)$ and $\varphi(T')$ in order to capture the fusion of clock trees described in 3.6.3.

4 cases are considered.

1. $G = \mathbf{0}$: we eliminate this case.

   As $\mathcal{L}$ is a one-to-one map, this case means that the formula $h_1 {<}op{>} h_2$ is equivalent to the null clock $\mathbb{0}$. Consequently — a clock is included in its parent — all the clocks in the tree T' are equivalent to $\mathbb{0}$. This may actually induce the simplification of many terms during the whole clock calculus process. But it does not result in the fusion of $\varphi(T)$ and $\varphi(T')$.

2. $G = \mathbf{1}$

   This means that $h_1 {<}op{>} h_2$ is equivalent to the root r, since $\mathcal{L}(\varphi(r)) = \mathbf{1}$ too. In this case we build the fusion of $\varphi(T')$ into $\varphi(T)$ through the node r.

3. More generally, if there exists a node $\varphi(h)$ in the hierarchy $\varphi(T)$ such that $\mathcal{L}(\varphi(h)) = G$, we perform the fusion of $\varphi(T')$ into $\varphi(T)$ through the node $\varphi(h)$.

4. If there exists no such node, we pick a new boolean variable $\varphi(l)$ — which belongs neither to $\varphi(T)$ nor to $\varphi(T')$ — and we augment the hierarchy $\varphi(T)$ whith $\varphi(l)$ in order to fall back into the previous case. Here is how we proceed.

   The hypotheses of the factorization — $G \neq \mathbf{0}$ and $G \neq \mathbf{1}$ — are satisfied; G can be written:
   $$G = \mathcal{L}(p) \cdot G'$$
   where p is a boolean variable — some $\varphi(k)$ — and G' is a BDD.

   We decorate $\varphi(l)$ with the BDD G' and insert it as the last child of $\varphi(k)$. In other terms, the hierarchy $\varphi(T)$ is augmented with the node $\varphi(l)$ and, we update the filiation function f, the total order $\leq$ and the map $\mathcal{B}$ in order to preserve the hierarchy structure for $\varphi(T)$.

   - The filiation: $f(\varphi(l)) = \varphi(k)$. $f(\varphi(l))$ is well defined, for $p = \varphi(k)$ is unique;
   - The ordering: $\varphi(l)$ is greater than all the other children of $\varphi(k)$ (see figure 14); a dfs on the augmented tree yields the new total order. Note that with this ordering $\varphi(l)$ is greater than all the descendants of $\varphi(k)$ (see (3) the properties of the total order).

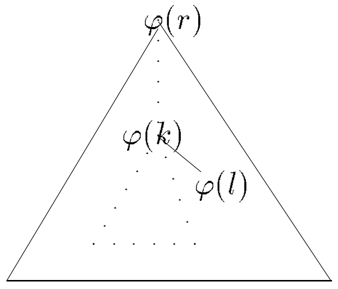


Figure 14: insertion of a new node

- the map $\mathcal{B}$: we define $\mathcal{B}(\varphi(l)) = G'$; $\mathcal{B}(\varphi(l))$ is well defined, for G' is unique; moreover, $\varphi(l)$ is given an ordering such that $\mathcal{B}(\varphi(l))$ satisfy the locality criterion (4).

With the decoration of $\varphi(l)$ we have

$$\begin{aligned} \mathcal{L}(\varphi(l)) &= \mathcal{B}(\varphi(l)) \cdot \mathcal{L}(f(\varphi(l))) \quad \text{see the definition of } \mathcal{L} \text{ in 4.7} \\ &= G' \cdot \mathcal{L}(\varphi(k)) \\ &= G \end{aligned}$$

We are back in the case 3; we perform the fusion of $\varphi(T')$ into $\varphi(T)$ through the node $\varphi(l)$ which has been created and inserted into $\varphi(T)$ for that purpose.

5.6 Iterating the fusion process

In the previous paragraphs we showed that a partition tree T can be decorated and transformed into a hierarchy $\varphi(T)$. Afterwards, we showed that the fusion of a partition tree T' into a partition tree T can be done by the fusion of the hierarchy $\varphi(T')$ into the hierarchy $\varphi(T)$. Since the tree which results from the fusion of 2 hierarchies is also a hierarchy, the fusion process can be iterated. Hence, hierarchies are a formal framework and a generalization of partition trees.

5.7 Experimental results

5.7.1 Description of the experimentation

In this section we compare 4 representations of a system of boolean equations.

1. *Tree and Normal Form* (*T&NF*): the system of equations is represented by a tree of clocks; each clock is given a sum-of-products normal form. This method is presented in [5].

2. *Tree and* BDD (*T&BDD*): a tree structure together with a BDD canonical form as presented earlier in this report.

3. BDD *characteristic function*: the whole system of equations is represented by a single BDD; a system of equations over n boolean variables can be viewed as a subset of $\{0, 1\}^n$. Hence it can be given a representation in the form of a characteristic function [17, 30].

4. BDD *characteristic function after T&BDD*: the original system of equations is transformed by a *T&BDD* into a tree (which is still a system of equations); then a BDD characteristic function is constructed.

These 4 representations are compared in terms of the time and space it takes to built them for some sample SIGNAL processes. The time is the classical unix *user-time*. For BDD-based methods, the space is given by the number of BDD nodes. And for *T&NF* the space is given in terms of number of formulas.

The sample SIGNAL programs are:

- STOPWATCH: this program models a watch with chronometer and alarm capabilities plus a simulation context;
- WATCH: it is a sub-process of STOPWATCH; its watch part;
- SUPERVISOR: it is a part of STOPWATCH; it abstracts the control part of the physical environment and dispatches external events to the watch, the alarm and the chronometer sub-processes,
- ALARM: the alarm function;
- CHRONO: the chronometer function;
- PACE MAKER: it is a part of STOPWATCH; it permits to change the pace of events in order to perform various simulations;
- ROBOT: it is part of the controller of an active-vision robot; whereas the previous processes contain much control, this one models a computation-intensive estimation of some 3-D parameters.

The measures are conducted on a SUN4/Sparc10 with 64MB main memory. Manipulations of BDDs use a UC Berkeley BDD package.

For the experimentation we set a 200MB virtual memory limit and a 40mn cpu time limit.

5.7.2 Comparisons

	T&BDD	BDD characteristic function	BDD charac. func. after *T&BDD*
STOPWATCH 1318 variables	61893	unable-cpu	unable-cpu
WATCH 785 variables	34753	unable-cpu	unable-cpu
ALARM 465 variables	3428	unable-mem	unable-cpu
CHRONO 282 variables	1548	unable-mem	422975
SUPERVISOR 202 variables	425	unable-cpu	226472
PACE MAKER 96 variables	50	53610	582
ROBOT 99 variables	36	unable-cpu	415

Comparisons in terms of BDD-nodes

unable-cpu denotes a computation that was unable to terminate within the time limit.
unable-mem denotes a computation that was unable to terminate within the memory limit

	T&NF	*T&BDD*	BDD characteristic function	BDD charac. func. after *T&BDD*
STOPWATCH	69.22s	27.07s	unable-cpu	unable-cpu
WATCH	24.20s	14.67s	unable-cpu	unable-cpu
ALARM	2.60s	2.19s	unable-mem	unable-cpu
CHRONO	0.55s	0.92s	unable-mem	409.09s
SUPERVISOR	0.20s	0.45s	unable-cpu	146.32s
PACE MAKER	0.07s	0.10s	160.50s	0.36s
ROBOT	0.20s	0.27s	unable-cpu	0.31s

Comparisons in terms of cpu-time

As programs get larger, *T&BDD* tend to be faster than *T&NF*. There are basically 2 reasons for that:

- during a *T&NF* process, as described in [5], the insertion of a formula into a hierarchy requires the insertion of many other sub-formulas;
- as it will be shown in the next section, during a *T&BDD* process, the properties of our ordering yields small-sized BDDs.

Most of the measures that involve a characteristic function were unable to compute within the resource limits. It appears clearly that characteristic functions are impractical.

The following table gives the informations about the *T&NF* representation; in this table, the *number of initial formulas* is the number of formulas that are built for the creation of the system of equations; during this creation, some elementary reductions such that $h \vee h = h, h \wedge h = h, \widehat{c} \wedge [c] = [c]$ are applied. So, at the end of the creation step we obtain (second column of the table) the *number of formulas before solving*. We give after that in the third column the *number of built formulas* for solving the system of equations and lastly the *number of useful formulas*: a formula is useful if it defines a clock of signals of the program or it is a clock of the conditional dependency graph.

	number of initial formulas	number of formulas before solving	number of all built formulas	number of useful formulas
STOPWATCH	1876	1318	5295	526
WATCH	1134	785	2943	309
ALARM	710	465	1117	187
CHRONO	434	282	674	117
SUPERVISOR	368	202	477	91
PACE MAKER	146	96	171	36
ROBOT	245	99	257	30

Informations about the *T&NF* representation

6 Hierarchies and the ordering problem in BDDs

The BDD technique is an efficient representation of boolean functions. A BDD is defined with respect to a total order on the boolean variables (see 4.1). As we will see it on the following example, the size of the BDD that represents a boolean function depends on the ordering of the variables.

6.1 Example

Consider the boolean function:

$$(x_1, x_2, x_3, x_4) \longmapsto x_1x_2 \ + \ x_3x_4$$

We denote $BDD(x_1x_2 \ + \ x_3x_4, <x_1, x_2, x_3, x_4>)$ its BDD with respect to the ordering $x_1 \leq x_2 \leq x_3 \leq x_4$.

Similarly, we denote $BDD(x_1x_2 \ + \ x_3x_4, <x_1, x_3, x_2, x_4>)$ its BDD with respect to the ordering $x_1 \leq x_3 \leq x_2 \leq x_4$.

These BDDs are drawn on figures 15 and 16.

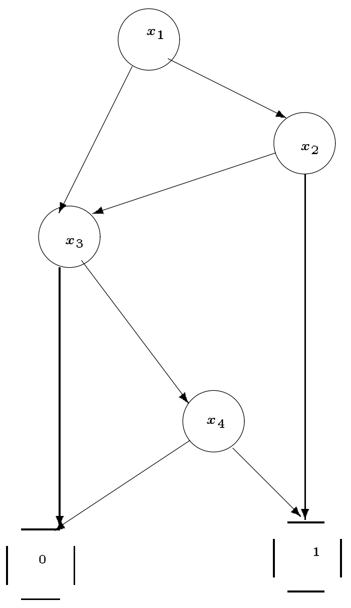


Figure 15: $BDD(x_1x_2 \; + \; x_3x_4, <x_1, x_2, x_3, x_4>)$

6.2 Finding the best ordering

Unfortunately, finding the best ordering for an arbitrary boolean function is an NP-hard problem [13]. There is an algorithm to find the best ordering [18] and its complexity is $O(n^2 3^n)$ is n is the number of variables.

Finding the best ordering being very complex, many heuristics have been designed.

6.3 Overview of the heuristics

Most of the heuristics that yield a "good" ordering for BDD minimization fall into 2 classes:

- the methods based on a traversal of the boolean term viewed as a logic circuit [28, 19]

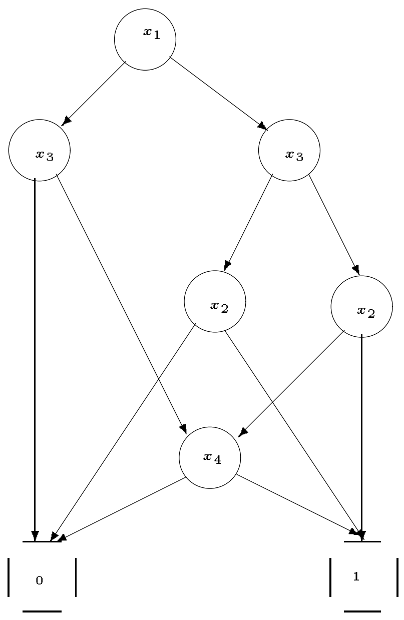


Figure 16: $BDD(x_1x_2 + x_3x_4, <x_1, x_3, x_2, x_4>)$

- the methods that construct first a BDD with an arbitrary ordering, then gradually improve it by permuting variables [24]

Going into further details with the known heuristics is not the purpose of this report. Though there is a theorem that relates, in some cases, the ordering problem to a factorization. Hence the link with SIGNAL hierarchies.

6.4 Disjoint support decomposition

The material presented here is borrowed from [28, 6]

If a boolean function F can be expressed by $SF_1{<}op{>}SF_2$ where $<op>$ is a boolean operator, and, SF_1 and SF_2 are boolean functions with disjoint input support — $Supp(SF_1) \cap Supp(SF_2) = \emptyset$ — then F is said to have a disjoint support decomposition.

Example

$$F = \underbrace{x_1 \cdot x_2}_{SF_1} + \underbrace{x_3 \cdot x_4}_{SF_2}$$

has a disjoint support decomposition.

$$SF_1 = \underbrace{x_1}_{SF_{1_1}} \cdot \underbrace{x_2}_{SF_{1_2}}$$

has a disjoint support decomposition as well.

Proposition 5 *If a boolean function accepts a disjoint support decomposition with respect to an operator <op> — $F = SF_1$<op>SF_2 — then in a minimized sized* BDD, *the variables of SF_1 precede of follow those of SF_2; but, they may not be interleaved.*

6.5 Link with hierarchies

There is a strong similarity between that proposition and SIGNAL hierarchies because our factorization is actually a disjoint support decomposition.

Recall the canonical factorization we presented in 4.8: given two nodes x_1 and x_2 of a hierarchy, let G denote one of the three BDDs $\mathcal{L}(x_1)\cdot\mathcal{L}(x_2)$, $\mathcal{L}(x_1)+\mathcal{L}(x_2)$, $\mathcal{L}(x_1)\cdot\overline{\mathcal{L}(x_2)}$.
We showed that for $G \neq \mathbf{0}$ and $G \neq \mathbf{1}$:

$$\left\{\begin{array}{l} G = \mathcal{L}(p) \cdot G' \\ Supp(G') \neq \emptyset \\ Supp(G') \subseteq Des(p) \end{array}\right.$$

In lemma 4 we showed that $Supp(G') \cap Supp(\mathcal{L}(p)) = \emptyset$ which means that we have a disjoint support decomposition.

Moreover, lemmas 3 and 2 showed that the ordering on the nodes is such that

$$\forall x \in Supp(\mathcal{L}(p)),\ \forall y \in Supp(G'),\ x < y$$

which is clearly the ordering suggested by the proposition.

7 Conclusion

In this report we have presented the synchronous language SIGNAL designed for real-time programming. The compilation of SIGNAL programs requires the resolution of a system of boolean equations; the variables being *clocks*. We have presented the *clock calculus*: the set of techniques used to solve the system of equations.

The clock calculus is based on an arborescent representation of clocks. Following that basis we have introduced as a formal framework a BDD-based data structure called *hierarchy*. We have shown through boolean function factorization that, under some hypotheses, hierarchies are a *canonical form* of SIGNAL clocks.

Finally, we have shown a link between hierarchies and the ordering problem in BDDs. The ordering given in the definition of a hierarchy yields "small-sized" BDDs. Future works could investigate the converse: take a boolean function and try to find a hierarchical organization of its variables in order to yield a "good" BDD. Then, hierarchies would be a framework for the design of ordering heuristics.

Due to the nice properties of their total order, BDD-based hierarchies have been successfully implemented in the SIGNAL compiler. The problems induced by the integration of BDDs in the existing compiler, will be discussed in another report.

Contents

References

[1] A. Benveniste and G. Berry. Special section on another look at real-time programming. *Proceedings of the IEEE*, 79(9):1268–1336, September 1991.

[2] A. Benveniste, B. Le Goff, and P. Le Guernic. *Hybrid Dynamical Systems theory and the language* SIGNAL. Research Report 838, INRIA, Rocquencourt, April 1988.

[3] A. Benveniste and P. Le Guernic. *A denotational theory of synchronous communicating systems.* Research Report 685, INRIA, Rocquencourt, June 1987.

[4] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 87–152, 1992.

[5] L. Besnard. *Compilation de* SIGNAL*: horloges, dépendances, environnement.* PhD thesis, Université de Rennes 1, France, Septembre 1992.

[6] T. Besson, H. Bouzouzou, I. Floricica, G. Saucier, and R. Roane. Input order for roobdds based on kernel analysis. In *Euro Asic 93*, pages 266–272, 1993.

[7] J. Billon and J. Madre. Original concepts of PRIAM, an industrial tool for efficient formal verification of combinational circuits. In G. Milne, editor, *Proceedings of the fusion of hardware design and verification*, pages 487–501, North Holland, 1988.

[8] M. L. Borgne. *Systḿes dynamiques polynomiaux sur des corps finis.* PhD thesis, Université de Rennes 1, 1993.

[9] P. Bournai, B. Chéron, T. Gautier, B. Houssais, and P. Le Guernic. SIGNAL*manual.* Technical Report 745, IRISA, July 1993.

[10] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on computers*, C-35(8):677–691, August 1986.

[11] A. Burns. *Programming in* OCCAM 2. Addison-Wesley, 1988.

[12] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, pages 178–188, Munich, 1987.

[13] O. Coudert. *SIAM: Une Boîte à Outils Pour la Preuve Formelle de Syst'emes Séquentiels.* PhD thesis, Ecole Nationale Supérieure des Télécommunications, France, 1991.

[14] O. Coudert, C. Berthet, and J. Madre. Verification of synchronous sequential machines based on symbolic execution. In *LNCS 407*, pages 365–373, 1989.

[15] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge Mathematical Textbooks, 1991.

[16] R. de Simone and D. VERGAMINI. *Aboard AUTO.* Technical Report, INRIA - Sophia Antipolis, October 1989.

[17] B. Dutertre. *Spécification et preuve de systèmes dynamiques.* PhD thesis, Université de Rennes 1, France, Décembre 1992.

[18] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for bdds. In *24th ACM/IEEE Design Automation Conference*, pages 348–356, 1987.

[19] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *ICCAD 88*, pages 2–5, 1988.

[20] G.R.E.P.A. *Le GRAFCET, de nouveaux concepts.* Cepadues Editions, 1985.

[21] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer, 1993.

[22] C. A. R. Hoare. Communicating Sequential Processes. *Communications of ACM*, 21(8):666–677, August 1978.

[23] INMOS. OCCAM*2 Reference Manual*. Prentice Hall, 1987.

[24] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based of exchanges of variables. In *ICCAD 91*, pages 472–475, 1991.

[25] B. Le Goff. *Inférence de contrôle hiérarchique : application au temps-réel.* PhD thesis, Université de Rennes 1, France, 1989.

[26] P. Le Guernic and A. Benveniste. *Real-time synchronous, data-flow programming: The language* SIGNAL *and its mathematical semantics.* Research report 620, INRIA, Rocquencourt, France, June 1986.

[27] O. Maffeïs, B. Chéron, and P. Le Guernic. *Transformations du Graphe des programmes* SIGNAL. Research report 1574, INRIA France, Rennes, January 1992.

[28] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vicentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *ICCAD*, pages 6–9, 1988.

[29] M. W. Rogers. *Ada, Language, Compilers and Bibliography.* Cambridge University Press, 1984.

[30] H. J. Touati, H. Savoy, R. Brayton, B. Lin, and A. Sangiovanni-Vicentelli. Implicit state enumeration of finite state machines using bdd's. In *IEEE conference on Computer-Aided Design*, pages 130–133, 1990.

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399