SIGNAL and SIGALI
User's Manual

Hervé Marchand & Éric Rutten

February 1, 2002

Abstract

This documentation is designed to serve as a user's manual for SIGNAL and SIGALI. It explains how one can use SIGNAL and SIGALI *from scratch*, without any previous knowledge about the architecture of either of them. SIGNAL is the compiler of a synchronous data-flow language of the same name. This language is used for precise specification of real-time reactive discrete event systems. When used with one of its options, the SIGNAL compiler produces a Polynomial Dynamical System(PDS) model of the SIGNAL program in a code appropriate for SIGALI. SIGALI is a model-checking tool based on formal calculus which takes this PDS model as input and offers functionalities for verification of system properties and discrete controller synthesis. The SIGNAL compiler can also produce code in other formats like the DC+ (declarative code) format (which is an equational level encoding of implicit automata) or sequential C code.

1 Modelling a system in SIGNAL

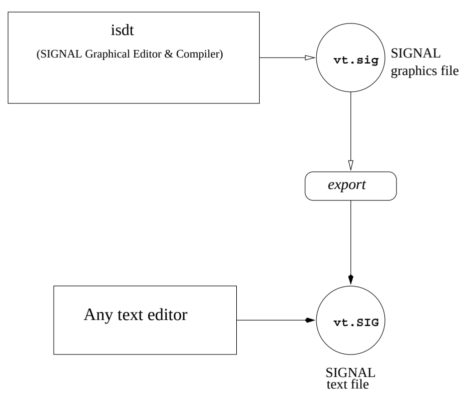

Figure 1: The **specification** stage.

To specify our model, we use the synchronous data flow language SIGNAL [?]. The aim of SIGNAL is to support the design of safety critical applications, especially those involving signal processing and process control. The synchronous approach guarantees the determinism of the specified systems, and supports techniques for the detection of causality cycles and logical incoherences. The design environment features a block-diagram graphical interface [?], a formal verification tool, SIGALI, and a compiler that establishes a hierarchy of inclusion of logical clocks (representing the temporal characteristics of discrete events), checks for the consistency of the inter-dependencies, and automatically generates optimized executable code ready to be embedded in environments for simulation, test, prototyping or the actual system. Further, the model read by SIGALI has to be in **z3z** format which is obtained by compiling the SIGNAL program using the **-z3z** option. Fig. 1 shows the specification stage.

1.1 The SIGNAL language & Specification

For specification of a system, one can use the syntax of the language SIGNAL V4 [3]. The SIGNAL language [?] manipulates *signals* X, which denote unbounded series of typed values, indexed by time. An associated *clock* determines the set of instants at which values are present. The constructs of the language can be used in an equational style to specify the relations between signals, *i.e.*, between their values and between their clocks. Data flow applications are activities executed over a set of instants in time. At each instant, input data is acquired from the execution environment; output values are produced according to the system of equations considered as a network of operations.

The SIGNAL language is defined by a small kernel of operators. The basic language constructs are summarized in Table (1). Each operator has formally defined semantics and is used to obtain a clock equation and the data dependencies of the participating signals. For a more detailed description of the language, its semantic, and applications, the reader is referred to [?].

Language Construct	Signal syntax	Description
stepwise extensions	C := A op B	where op : arithmetic/relational/boolean operator
delay	ZX := X $ n	memorization of the n^{th} past value of X
extraction	C := A when B	C equal to A when B is present and true
priority merging	C := A default B	if A is present C:=A else if B present C:= B else C absent
Process Composition	(|P|Q|)	processes are composed, common names correspond to shared signals
useful extensions		
	when B	the clock of the true instants of B
	event B	the presence instants of B
	A^= B	Clock of A equal with clock of B

Table 1: Basic SIGNAL language constructs

1.1.1 A 1 bit shift-register.

For example, the process m modelled by the following code represents a 1 bit memory:

```
process m = {boolean Minit}
        (? boolean Min ! boolean Mout; )
(|  Z_Mout := Min default Mout)
 |  Mout :=  Z_Mout $ 1 init Minit
 |)/ Z_Mout
```

There is one Boolean input `Min`, one Boolean output `Mout`, and a constant initialization parameter `Minit`. The output is defined as a combination of the input and the value in memory with delay of one clock cycle.

The program is compiled by the SIGNAL compiler which analyses the clocks and checks the constraints, but does not generate executable code because `Mout` is not completely determined by the input. It can be present between two successive occurences of the input, arbitrarily often.

1.1.2 A flipflop.

The 1 bit memory defined above is used in the following process called `flipflop`. It has one Boolean output `B` denoting its two states: `true` and `false`, and one Boolean input `C`. The flipflop changes its state when `C` is `true`(see Fig. 2):

```
process flipflop  =
        (? boolean C ! boolean B; )
(  % One Boolean memory %
 | B := m{false}(NewB)
   % New value is the negation of the current value %
 | NewB := not B when C
   % The memory is synchronized with the input %
 | B ^= C
 |)
where
(declaration of m)
end;
```

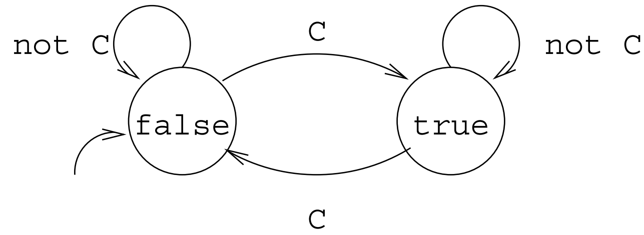

Figure 2: Behavior of `flipflop`.

The program when compiled with the `-c` option, generates C code from which an executable can be obtained in the manner described in the next section. This executable can be simulated on an input file containing the values of the input and produces an output file containing the values of the output.

1.1.3 Boolean double memory.

In the process `double_m`, there are two inputs `C1`, `C2` and two outputs `B1`, `B2`. The two outputs encode four states. The inputs and outputs are synchronized meaning that they have the same clock. The process makes use of two Boolean memories:

```
process double_m  =
        (? boolean C1, C2
         ! boolean B1, B2; )

(  % Two Boolean memories %
 | B1 := m{false}(NewB1)
 | B2 := m{false}(NewB2)

   % The new value is the value of the input %
 | NewB1 := C1
 | NewB2 := C2

   % The memory is synchronized with the inputs %
 | B1 ^= B2 ^= C1 ^= C2
 |)
where
(declaration of m)
end;
```

Thus, in the above model, the outputs take the values of the respective inputs with a delay of one clock cycle. So essentially the system memorizes two Boolean values(see Fig. 3).

1.2 Compilation

Compilation into executable C code. For this, the compilation is done with the `-c` option. For example, for the file `D.SIG`, the command is `sig -c D.SIG`. This produces the relevant `.c` and `.h` files which are used to build the executable:

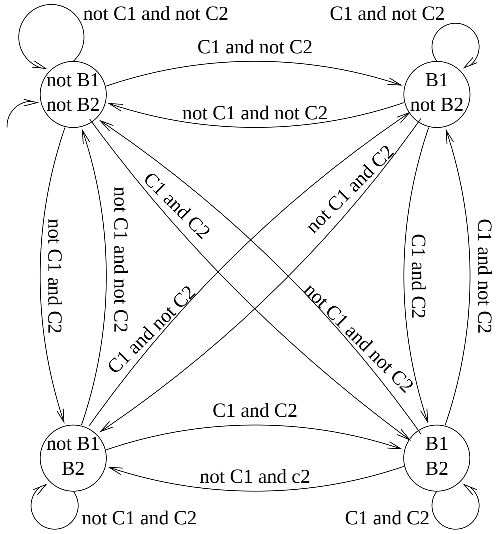

Figure 3: Behavior of `double_m`.

```
prakashp @ yeti >sig -c D.SIG

       SIGNAL/DC+  Compiler version V4.13.10 (Jan 2001) / linux
       INRIA - All rights reserved
             ----------------
 You are entitled to use this software only
 if your organization has signed an agreement with INRIA
             ----------------
  This software uses a Binary Decision Diagram Package -- Copyright (c) 1988, 1989
  Regents of the University of California.  All rights reserved.
             ----------------

   ===> Program analysis
   ===> Reduction to the kernel language
   ===> Graph generation
   ===> Clock calculus (Process: double_m)
    ------------ DC+2bDC        .... BEGIN -------------
       ....BEGIN for node : double_m
       ....DONE....
    ------------ DC+2bDC        .......END -------------
   ===> Graph processing(Process : double_m)
    ------------ bDC+2sbDC+     .... BEGIN -------------
    ------------ bDC+2sbDC+     .......END -------------
   ===> C generation (Process : double_m)
           * Externals Declarations  : double_m_externals.h
           * Types Declarations      : double_m_types.h
           * Main Program            : double_m_main.c
           * Instant Execution       : double_m_body.c
           * Header file (body)      : double_m_body.h
           * Input/Output procedures : double_m_io.c
prakashp @ yeti >
```

Now the C files obtained can be compiled to get the executable:

```
prakashp @ yeti >cc -o double_m double_m_main.c double_m_body.c double_m_io.c
double_m_body.c:
double_m_io.c:
double_m_main.c:
prakashp @ yeti >
```

Fig. 4 summarizes the compilation stage.

In this case the executable is `double_m`. For execution, two files `RC1.dat` and `RC2.dat` are required which contain the Boolean values of the inputs `C1` and `C2` respectively. The execution of `./double_m` produces two files `WB1.dat` and `WB2.dat` which contain the Boolean values of the outputs `B1` and `B2` respectively(see Fig. 5). As an example, the input and output files obtained after a sample simulation are:

```
RC1.dat:  1 0 0 1 1 1 0 1 0 0 0 1
WB1.dat:  0 1 0 0 1 1 1 0 1 0 0 0

RC2.dat:  0 0 0 1 1 0 1 1 1 0 1 0
WB2.dat:  0 0 0 0 1 1 0 1 1 1 0 1
```

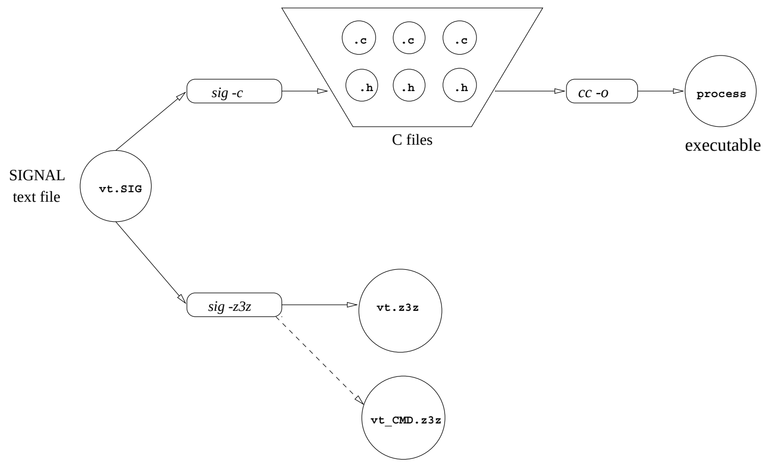

Figure 4: The **compilation** stage.

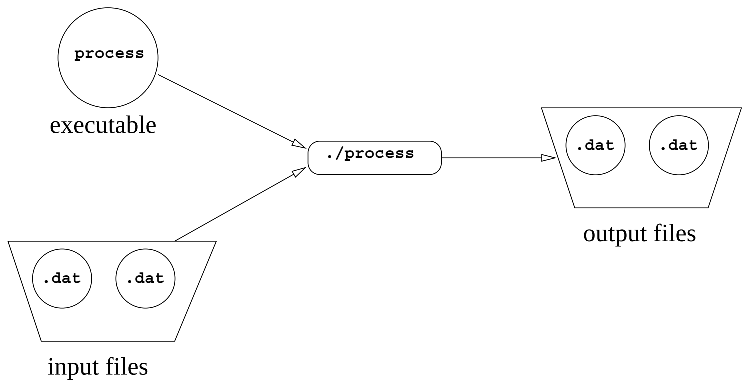

Figure 5: Simulation by an executable and input-output files.

Thus, the output values are same as their corresponding input values delayed by one clock cycle, which is the expected result.

Compilation into the z3z format. In order to analyze the system using SIGALI, one has to compute a polynomial dynamical system. For this,the specification of the system written in SIGNAL is compiled with the `-z3z` option. Suppose the file in which the process `double_m` is specified is `D.SIG`. When `D.SIG` is compiled by the command `sig -z3z D.SIG`, a file called `double_m.z3z` is obtained as output:

```
prakashp @ yeti >sig -z3z D.SIG

       SIGNAL/DC+  Compiler version V4.13.10 (Jan 2001) / linux
       INRIA - All rights reserved
             -----------------
 You are entitled to use this software only
 if your organization has signed an agreement with INRIA
             -----------------
  This software uses a Binary Decision Diagram Package -- Copyright (c) 1988, 1989
  Regents of the University of California.  All rights reserved.
             -----------------

  ===> Program analysis
  ===> Reduction to the kernel language
  ===> Graph generation
  ===> Clock calculus (Process: double_m)
     # Equations over Z/3Z generation: double_m.z3z
prakashp @ yeti >
```

we will come back to this point in the next section.

2 The model checker SIGALI

The SIGNAL environment also contains a verification and controller synthesis tool-box, named SIGALI. This tool allows to prove the correctness of the dynamical behavior of the system. The equational nature of the SIGNAL language leads naturally to the use of a method based on polynomial dynamical equation systems (PDS) over $\mathbb{Z}/_{3\mathbb{Z}}$ (*i.e.*, integers modulo 3: {-1,0,1}) as a formal model of program behavior.

2.1 Basic facts about SIGALI

The theory of Polynomial Dynamical Systems uses classical tools in algebraic geometry, such as ideals, varieties and comorphisms [?]. The techniques consist in manipulating the system of equations instead of the sets of solutions, which avoids enumerating the state space.

2.1.1 The mathematical framework : an Overview

Let $Z = \{Z_1, Z_2, ..., Z_p\}$ be a set of p variables and $\mathbb{Z}/_{3\mathbb{Z}}[Z]$ be the ring of polynomials with variables Z. Thus $\mathbb{Z}/_{3\mathbb{Z}}[Z]$ is the set of all polynomials of p variables. Given an element of $\mathbb{Z}/_{3\mathbb{Z}}[Z]$, $P(Z_1, Z_2, \ldots, Z_p)$ (shortly $P(Z)$), we associate its set of solutions $Sol(P) \subseteq (\mathbb{Z}/_{3\mathbb{Z}})^m$:

$$Sol(P) \stackrel{\text{def}}{=} \{(z_1, ..., z_k) \in (\mathbb{Z}/_{3\mathbb{Z}})^k | P(z_1, ..., z_k) = 0\} \tag{1}$$

It is worthwhile noting that in $\mathbb{Z}/_{3\mathbb{Z}}[Z]$, $Z_1^p - Z_1, ..., Z_k^p - Z_k$ evaluate to zero. Then for any $P(Z) \in \mathbb{Z}/_{3\mathbb{Z}}[Z]$, one has $Sol(P) = Sol(P + (Z_i^p - Z_i))$. We then introduce the quotient ring of polynomial functions $A[Z] = \mathbb{Z}/_{3\mathbb{Z}}[Z]/_{<Z^p - Z>}$, where all polynomials $Z_i^p - Z_i$ are identified to zero, written for short $Z^p - Z = 0$. $A[Z]$ can be regarded as the set of polynomial functions with coefficients in $\mathbb{Z}/_{3\mathbb{Z}}$ for which the degree in each variable is lower than 2. [?] showed how to define a representative of $Sol(P)$ called the *canonical generator*. Our techniques will rely on the following: For all polynomials $P_1, P_2, P \in \mathbb{Z}/_{3\mathbb{Z}}[Z]$

- $Sol(P_1) \subseteq Sol(P_2)$ whenever $(1 - P_1^2) * P_2 \equiv 0$. (*inclusion*)
- $Sol(P1) \cap Sol(P_2) = Sol(P_1 \oplus P_2)$ (*intersection*), where

$$P_1 \oplus P_2 \stackrel{\text{def}}{=} (P_1^2 + P_2^2)^2 \tag{2}$$

- $Sol(P_1) \cup Sol(P_2) = Sol(P_1 * P_2)$ (*union*) and $(\mathbb{Z}/_{3\mathbb{Z}})^m \setminus Sol(P) = Sol(1 - P^2)$ (*complementary*).

2.1.2 Dynamical systems: Basics

A dynamical system can be mathematically modelled as a system of polynomial equations over $\mathbb{Z}/3\mathrm{Z}$ Z (the Galois field of integers modulo 3) of the form:

$$\begin{cases} Q(X,Y) & = & 0 \\ X' & = & P(X,Y) \\ Q_0(X) & = & 0 \end{cases} \tag{3}$$

where,

- X is the set of n **state** variables, represented by a vector in $(\mathbb{Z}/3\mathbb{Z})^n$;
- Y is the set of m **event** variables, represented by a vector in $(\mathbb{Z}/3\mathbb{Z})^m$;
- $Q(X,Y) = 0$ is the **constraint** equation;
- $X' = P(X,Y)$ is the **evolution** equation. It can be considered as a vectorial function from $(\mathbb{Z}/3\mathbb{Z})^{n+m}$ to $(\mathbb{Z}/3\mathbb{Z})^n$; and,
- $Q_0(X) = 0$ is the **initialization** equation.

In order to prove its dynamical properties, every SIGNAL process is translated into a system of polynomial equations over $\mathbb{Z}/3\mathbb{Z} = \{-1, 0, 1\}$ having the above form. The principle is to encode the 3 possible values of a Boolean signal by:

$$\begin{cases} present \wedge true & \iff & +1 \\ present \wedge false & \iff & -1 \\ absent & \iff & 0 \end{cases}$$

For the non-boolean signals, we only code the fact that the signal is *present* or *absent*: ($present \rightarrow 1$ and $absent \rightarrow 0$). Note that the square of *present* is 1, whatever its value, when it is present. Hence, for a signal `X`, its clock can be coded by x^2. It follows that two synchronous signals `X` and `Y` satisfy the constraint equation: $x^2 = y^2$. This fact is used extensively in the following. **Primitive operators.** Each of the primitive processes of SIGNAL can be encoded in a polynomial equation. For example `C := A when B`, which means "*if* $b = 1$ *then* $c = a$ *else* $c = 0$" can be rewritten in $c = a(-b - b^2)$: the solutions of this are the set of behaviors of the primitive process `when`. The delay `$`, which is a dynamic operator deserves some extra explanations. It requires memorizing the past value of the signal into a *state variable*. Translating `B` := `A $1`, requires the introduction of two auxiliary equations: (1) $x' = a + (1 - a^2)x$, where x' denotes the next value of state variable x, expresses the dynamics of the system. (2) $b = a^2x$ delivers the value of the delayed signal according to the memorization in state variable x. Table 2 shows how all the primitive operators are translated into polynomial equations. For the non boolean expressions, we just translate the synchronization between the signals. By composing the equations representing the

boolean instructions	
`B := not A`	$b = -a$
`C := A and B`	$c = ab(ab - a - b - 1)$ $a^2 = b^2$
`C := A or B`	$c = ab(1 - a - b - ab)$ $a^2 = b^2$
`C := A default B`	$c = a + (1 - a^2)b$
`C := A when B`	$c = a(-b - b^2)$
`B := A $1 (init` b_0`)`	$x' = a + (1 - a^2)x$ $b = a^2x$ $x_0 = \mathsf{b}_0$
non-boolean instructions	
`B :=` $f(A_1, \ldots, A_n)$	$b^2 = a_1^2 = \cdots = a_n^2$
`C := A default B`	$c^2 = a^2 + b^2 - a^2b^2$
`C := A when B`	$c^2 = a^2(-b - b^2)$
`B := A $1 (init` b_0`)`	$c^2 = a^2$

Table 2: Translation of the primitive operators.

primitive processes, any SIGNAL specification can be translated into a set of equations called polynomial dynamical system (PDS) as the one described in (3).

We now explain how one can use the model-checker SIGALI, in order to analyze the obtain polynomial dynamical system.

2.2 The SIGALI commands & Operations

2.3 General Commands

Starting and exiting The SIGALI environment can be started by the `sigali` command. A prompt `Sigali :` appears. To quit, one can use the SIGALI command `quit()`:

```
Sigali : quit();
-----------------------------------------------------------------
```

`^D` works also fine.

Loading the file of a model The `.z3z` file which contains the PDS model of the system (or any other SIGALI files, can be loaded by using the `load` or the `read` command. For example, in case of `double_m` the command is:

```
Sigali : load("double_m.z3z");
-----------------------------------------------------------------
```

Trace By the `trace` command it is possible to save in a file all the commands executed and results obtained in the `Sigali` environment:

- `trace("filename");` opens the file for trace.
- `fintrace();` closes the current trace file.

All commands executed(and the corresponding responses) in between are saved in the trace file.

Execution time SIGALI allows the measurement of the time taken for each computation. The command `chrono(true);` starts the clock. After each subsequent command, the time taken for the computation is displayed. The command `chrono(false);` stops the clock.

2.3.1 Symbols and declarations

A symbol or an identifier can be assigned to an expression in the following format:
`symbol : <expression>;`
For example:
`p : a^2 * b + c^2;`
assigns the identifier p to the expression $a^2b + c^2$.

Indeterminate symbols can be delared by the command: `declare` or `ldeclare`. For example:
`declare(a,b,c,d);` takes one or more parameters.
`ldeclare([a,b,c,d]);` takes only one parameter (as a list).
Once a symbol is declared, its not possible to modify its value. The command `indeter();` lists all the indeterminate symbols.

2.3.2 Polynomials and equations

A polynomial is an expression. An equation is of the form `p1=p2` where `p1` and `p2` are two polynomial expressions. SIGALI can also manipulate lists of polynomials and equations. For example, `[a + b, a, b, 0, 1];` is a list of 5 polynomials and `[a ^2 = b ^2, c = a and b];` is a list of 2 equations. Of course a symbol can be assigned to a list as well. For example:
`list : [a + b, a, b, 0, 1];`
`equations : [a ^2 = b ^2, c = a and b];`
The command `eval` evaluates a polynomial:
`eval(p,[a,b,c],[0,1,-1]);`
evaluates the polynomial `p` after substituting `0`, `1` and `-1` for `a`, `b` and `c` respectively. Of course these variables must occur in `p`.

If `p` is a polynomial, `lp1` and `lp2` are two lists of polynomials,`lvar1` and `lvar2` are two lists of variables, and `lconst` is a list of constants(with values 0, 1 or -1), then:
`rename(p, lvar1, lvar2);`
replaces in `p`, the i^{th} variable of `lvar1` by the i^{th} variable of `lvar2`.
`subst(p, lvar1, lp1);`
replaces in `p`, the i^{th} variable of `lvar1` by the i^{th} polynomial of `lp1`.
In case of the functions:
`l_eval(lp1, lvar1, lconst);`
`l_rename(lp1, lvar1, lvar2);`
`l_subst(lp1, lvar1, lp2);`
the first argument is a list of polynomials instead of one polynomial and they perform they same function as their counterparts for each polynomial of the list.
The command `equal` compares two polynomials:

```
Sigali: ldeclare([a, b]);
-----------------------------------------------------------------

Sigali: equal(a,b);
False
-----------------------------------------------------------------

Sigali: equal(a when b, a * (- b - b^2));
True
-----------------------------------------------------------------
```

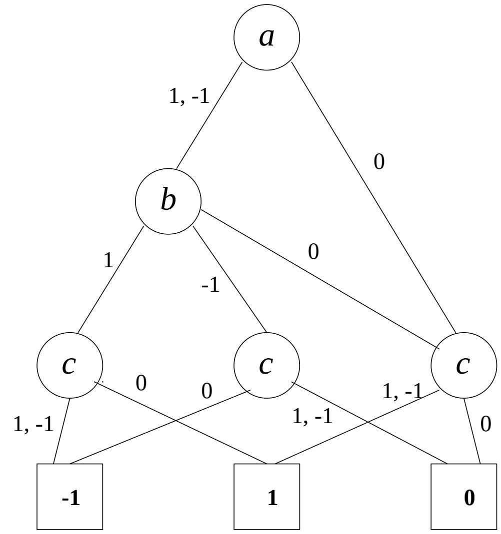

Figure 6: TDD representation of the polynomial $a^2b + c^2$.

2.3.3 Representation of polynomials

A variable or polynomial can only take values belonging to $\mathcal{F}_3 = \{-1, 0, 1\}$. In SIGALI, a polynomial is represented by means of a *Ternary Decision Diagram*(TDD) which is an extension of a *Binary Decision Diagram*(BDD). In a TDD, each non-leaf node represents a variable and each leaf node is a value of the polynomial. An arbitrary ordering of the variables must be done to facilitate the assignment of a node to a variable. Further, each non-leaf node has 3 edges emanating from it, labelled by the 3 possible values: {(-1 or 2), 0, 1} that the corresponding variable may take. So, each path from the root to a leaf assigns a unique sequence of values to the variables and the value of the leaf gives the value of the polynomial for that particular assignment. For example, if p is the polynomial $a^2b + c^2$, and the ordering is $a < b < c$, then p is represented by SIGALI as follows (The TDD representation of p is shown in Fig. 6.):

```
Sigali :  p : a^2 * b + c^2;
-----------------------------------------------------------
p
-----------------------------------------------------------
Sigali : p;
-----------------------------------------------------------
a=0
   #0#
   c=0
      %0
   c=1
      %1
   c=2
      %1
   c
a=1
   #1#
   b=0
      subformula 0
   b=1
      c=0
         %1
      c=1
         %2
      c=2
         %2
      c
   b=2
      c=0
         %2
      c=1
         %0
      c=2
         %0
      c
   b
a=2
   subformula 1
a
-----------------------------------------------------------
```

Note: The value 2 is equivalent to -1.

In order to avoid repetitions in listing, portions occuring more than once are labelled as `#n#` (n = 0, 1, 2, ...). These repetitions tend to occur when two or more edges enter a non-leaf node in the TDD. While reading the TDD, the label `subformula n`, wherever it occurs, is to be replaced by the portion labelled `#n#`.

2.3.4 (System of) Polynomials manipulation

The canonical generator of a polynomial system given by a list of polynomials can be computed by the function `gen`. The command: `gen(lpoly);`s where `lpoly` is a list of polynomials. For example:
`gen([a + b - c, a^2 - 1]);` gives the canonical generator of the polynomial system given by the two polynomials `a + b - c` and `a^2 - 1`. The previous command can also be given as:
`gen([a + b = c, a^2 = 1]);`

Complementation. Let `g` be a polynomial and V its set of solution, then the generator of the complement of V is obtained by: `complementary(g);`

Intersection. Let `p1` and `p2` be two polynomials and $V1$ and $V2$ be the corresponding set of solutions, then: `intersection(p1,p2);` is the canonical generator of $V_1 \cap V_2$. The number of arguments can be greater than 2. For example one can write `intersection(p1,p2,p3,p4);`

Union. Let `p1` and `p2` be two polynomials and $V1$ and $V2$ be the corresponding set of solutions, then: `union(p1,p2);` is the canonical generator of $V_1 \cup V_2$. As in case of `intersection`, the number of arguments can be greater than 2.

Tests of inclusion Let `p1` and `p2` be two polynomials and $V1$ and $V2$ be the corresponding set of solutions, then: `inclus(g1,g2);` is `True` if and only if $V_1 \subseteq V_2$. For example:

```
> declare(a);

> g1 : gen([a^2 = 1]);
g1

> g2 : gen([a = 1]);
g2

> inclus(g2,g1);
True

> inclus(g1,g2);
False
```

An Example:

```
> declare(a, b);


> list : [a = 1, b = -1];
list
> poly : gen([a = 1, b = -1]);
poly
> A : gen([a = 1]);
A
> B : gen([b = -1]);
B
> AnB : intersection(A, B);
AnB
> AuB : union(A, B);
AuB
> equal(poly, AnB);
True
> equal(poly, AuB);
False
> equal(complementary(AuB), intersection(complementary(A), complementary(B)));
True
> equal(AuB, A + B - AnB);
True
------------------------------------------------------------------
```

2.4 Systems and Processes

SIGALI distinguishes between two categories of dynamical systems: *systems* and *processes*.

2.4.1 Systems

Systems are general dynamical systemss in which null transitions(basically self loops) are taken into account even when all the signals are absent. For example in case of the process `double_m`, a *system* can be constructed as follows:

```
sys_double_m : systeme(conditions,etats,evolutions,initialisations,contraintes,controlables);

-------------------------------------------------------------------------
sys_double_m
-------------------------------------------------------------------------
```

`conditions` is a list of variables encoding the event variables, whereas *etats* is a list of variables which encodes the states variables. *controlables* is a subset of `conditions` and corresponds to the controllable event variables. *evolutions* is a list of polynomials (one for each state variables) which corresponds to the evolution of each state variables. *initialisations* is a list of polynomials (the solutions of this polynomial systems correspond to the initial states of the system). *contraintes* is also a list of polynomial encoding the constraints part of the polynomial dyanmical system (i.e. $Q(X, Y) = 0$).

2.4.2 Processes

In a *process*, null transitions are excluded i.e. no transition can take place in the absence all the signals. All dynamical systems originating from SIGNAL programs fall under this category. In case of the process `double_m`, a *process* can be constructed as follows:

```
proc_double_m :
processus(conditions,etats,evolutions,initialisations,contraintes,controlable);

-------------------------------------------------------------------------
proc_double_m
-------------------------------------------------------------------------
```

2.4.3 Access to the components

If `syst` is a dynamical system constructed by the command `systeme` or `processus`, then the 6 components of `syst` can be accessed by:

```
event_var(syst);
state_var(syst);
evolution(syst);
initial(syst);
constraint(syst);
controllable_var(syst);
```

2.4.4 Some special sets

If `g` is the canonical generator of a set of states E, then: `pred(syst, g);` is the canonical generator of the set of predecessors of E. Similarly, `all_succ(syst, g);` is the canonical generator of the set of states *all* of whose successors belong to E. `evnt_adm(syst, g);`is the canonical generator of the set of events admissible in E. If `g` is the canonical generator of a set of events F, then: `etats_adm(syst, g);` is the canonical generator of the set of states compatible with at least one of the events in F.

3 Verification of systems using SIGALI

SIGALI provides certain functionalities for the verification of the properties of a dynamical system.

3.1 Loading of the necessary libraries

The following files must be loaded:

```
load("Creat_SDP.z3z");

-------------------------------------------------------------------------

-------------------------------------------------------------------------
load("Verif_Determ.bib");

-------------------------------------------------------------------------

-------------------------------------------------------------------------
```

A more convenient way is to make a file called `Bibli.z3z` containing the `read` commands for the above files and then to load `Bibli.z3z` at the `Sigali` prompt.

3.2 Liveness

3.2.1 Rudiments

Definition: *A dynamical system is* **alive** *iff* $\forall x, y$ *such that* $Q(x,y) = 0$, $\exists y'$ *such that* $Q(P(x,y), y') = 0$.

In other words, a system is **alive** iff it contains no sink states.
If `syst` is a *system* or a *process*, then:
`vivace(syst);`
is `True` if and only if `syst` is **alive**.
In case of the *process* `proc_double_m` for example:

```
vivace(proc_double_m);

----------------------------------------------------------------------------
True
----------------------------------------------------------------------------
```

3.2.2 An example of the difference between *system* and *process*

A flipflop with constraint on the input. One may use the 1 bit memory `m` to define the process `flipflop_c`:

```
process flipflop_c  =
        (? event E
         ! boolean B; )

(  % One Boolean memory %
 | B := m{false}(NewB)
   % New value : negation of the current value %
 | NewB := not B
   % Memory no more frequent than input %
 | E ^+ B ^= E
   % Input admissible when the memory value is True %
 | E ^= when B
 |)
where

(declaration of m)
end;
```

The clock constraint specifies that an input is accepted only when the value of the memory is `true`. It also specifies that the memory value is present only when the input is present. These together try to impose constraints on the external event from within the system. This prevents the generation of executable code in this case.

Also, once the memory value becomes `false`, it remains `false` since no further input is accepted. Thus the *process* is blocked and it is not **alive**. On the other hand, in case of the *system*, null transitions can still take place from the `false` state to itself and so the *system* is **alive**.

Relative liveness. The evaluation of liveness of the two representations(*system* and *process*) by SIGALI yields:

```
load("flipflop_c.z3z");

----------------------------------------------------------------------------

----------------------------------------------------------------------------
sys_flipflop_c : systeme(conditions,etats,evolutions,initialisations,contraintes);

----------------------------------------------------------------------------
sys_flipflop_c
----------------------------------------------------------------------------
proc_flipflop_c : processus(conditions,etats,evolutions,initialisations,contraintes);

----------------------------------------------------------------------------
proc_flipflop_c
----------------------------------------------------------------------------
vivace(proc_flipflop_c);

----------------------------------------------------------------------------
False
----------------------------------------------------------------------------
vivace(sys_flipflop_c);

----------------------------------------------------------------------------
True
----------------------------------------------------------------------------
```

As expected, the *system* is **alive** but the *process* is not.

3.3 Safety Properties

3.3.1 Invariance

Definition: *A set of states E is* **invariant** *for a dynamical system iff for every state x in E and every event y admissible in x, the successor state $x' = P(x, y)$ is also in E.*
If `syst` is a dynamical system and `g` is the canonical generator polynomial of a set of states E,

`invariant(syst, g);`

is `True` if and only if E is **invariant** for `syst`.

For example, in case of the process `double_m`, one can specify a property `pr_eq : [etat_1 = etat_2];`. The **invariance** of this property can then be tested by the command:

`invariant(pf, gen(pr_eq));`,

where `pf` is the *process* constructed by the command `processus`.

3.3.2 Greatest invariant subset

Given a set of states E, there exists a set F which is the greatest invariant subset of E. If `syst` is a dynamical system and `g` is the canonical generator of E,then:

`pg_invariant(syst, g);`

gives the canonical generator of F. Abbreviation: `pgi(syst, g);`

3.3.3 Invariance under control

Definition: *A set of states E is* **control-invariant** *for a dynamical system iff for every state x in E, there exists an event y such that $Q(x, y) = 0$ and the successor state $x' = P(x, y)$ is also in E.*
If `syst` is a dynamical system and `g` is the canonical generator polynomial of a set of states E,

`c_invariant(syst, g);`

is `True` if and only if E is **control-invariant** for `syst`.

3.3.4 Greatest control-invariant subset

Given a set of states E, there exists a set F' which is the greatest control-invariant subset of E. If `syst` is a dynamical system and `g` is the canonical generator of E,then:

`pg_c_invariant(syst, g);`

gives the canonical generator of F'. Abbreviation: `pgci(syst, g);`

3.4 Reachability Properties

3.4.1 Reachability

Definition: *A set of states E is* **reachable** *iff for every state $x \in E$ there exists a trajectory starting from the initial states that reaches x.*
If `syst` is a dynamical system and `g` is the canonical generator polynomial of a set of states E,

`accessible(syst, g);`

is `True` if and only if E is **reachable** from the initial states of `syst`.

Note that `Reachable(syst, g);` works also fine

3.4.2 Attractivity

Definition: *A set of states F is* **attractive** *for a set of states E iff every trajectory initialized on E reaches F.* If `syst` is a dynamical system and `g` is the canonical generator polynomial of a set of states E,

`Attractivity(syst, g);`

is `True` if and only if E is **Attractive** from the initial states of `syst`.

Note: To avoid confusion between *states* and *properties*, it is essential to keep in mind that when a property is defined, SIGALI computes the set of states where the property holds. So for every property, there always corresponds a unique set of states. This set is empty if the property does not hold at any state of the system.

3.5 A demonstration

In order to demonstrate how one can use the SIGALI commands given in this section, as well as interpret SIGALI's response to these commands, a small example is given below. It is for the process **double_m** which is defined and explained in Section 2.1. It will be a good exercise to check for oneself the results produced by SIGALI in order to get a clear picture of the issues involved.

```
> trace("double_m.log");
> load("double_m.z3z");
> pf : processus(conditions, etats, evolutions, initialisations, contraintes);
pf
> varetat(pf);
[etat_1, etat_2]
> varevent(pf);
[C1, C2]
> pr_eq : [etat_1 = etat_2];
pr_eq
> pr_sync : [etat_1^2 = etat_2^2];
pr_sync
> pr_val : [etat_1 = -1, etat_2 = 1];
pr_val
> pr_par : [etat_2 = -1];
pr_par
> pr_eq;
------------------------
etat_1=0
   etat_2=0
      %0
   etat_2=1
      %1
   etat_2=2
      %1
   etat_2
etat_1=1
   etat_2=0
      %1
   etat_2=1
      %0
   etat_2=2
      %1
   etat_2
etat_1=2
   etat_2=0
      %1
   etat_2=1
      %1
   etat_2=2
      %0
   etat_2
etat_1
------------------------



> pr_sync;
------------------------
etat_1=0
   etat_2=0
      %0
   etat_2=1
      %1
   etat_2=2
      %1
   etat_2
etat_1=1
   #0#
   etat_2=0
      %1
   etat_2=1
      %0
   etat_2=2
      %0
   etat_2
etat_1=2
   subformula 0
etat_1
------------------------



> pr_val;
------------------------
etat_1=0
   %1
etat_1=1
   %1
etat_1=2
   %0
etat_1
------------------------
etat_2=0
   %1
etat_2=1
   %0
etat_2=2
   %1
etat_2
------------------------



> pr_par;
```

```
------------------------
etat_2=0
   %1
etat_2=1
   %1
etat_2=2
   %0
etat_2
------------------------



> invariant(pf, gen(pr_eq));
False


> invariant(pf, gen(pr_sync));
True


> invariant(pf, gen(pr_val));
False


> invariant(pf, gen(pr_par));
False


> accessible(pf, gen(pr_eq));
True


> accessible(pf, gen(pr_sync));
True


> accessible(pf, gen(pr_val));
True


> accessible(pf, gen(pr_par));
True


> c_invariant(pf, gen(pr_eq));
True


> c_invariant(pf, gen(pr_sync));
True


> c_invariant(pf, gen(pr_val));
True


> c_invariant(pf, gen(pr_par));
True


> pg_invariant(pf, gen(pr_eq));
%1



> pgi(pf, gen(pr_eq));
%1



> pgci(pf, gen(pr_eq));
etat_1=0
   etat_2=0
      %0
   etat_2=1
      %1
   etat_2=2
      %1
   etat_2
etat_1=1
   etat_2=0
      %1
   etat_2=1
      %0
   etat_2=2
      %1
   etat_2
etat_1=2
   etat_2=0
      %1
   etat_2=1
      %1
   etat_2=2
      %0
   etat_2
etat_1



> pgi(pf, gen(pr_sync));
etat_1=0
   etat_2=0
      %0
   etat_2=1
      %1
   etat_2=2
```

```
      %1
   etat_2
etat_1=1
   #0#
   etat_2=0
      %1
   etat_2=1
      %0
   etat_2=2
      %0
   etat_2
etat_1=2
   subformula 0
etat_1


> pgci(pf, gen(pr_sync));
etat_1=0
   etat_2=0
      %0
   etat_2=1
      %1
   etat_2=2
      %1
   etat_2
etat_1=1
   #0#
   etat_2=0
      %1
   etat_2=1
      %0
   etat_2=2
      %0
   etat_2
etat_1=2
   subformula 0
etat_1


> pgi(pf, gen(pr_val));
%1


> pgci(pf, gen(pr_val));
etat_1=0
   %1
etat_1=1
   %1
etat_1=2
   etat_2=0
      %1
   etat_2=1
      %0
   etat_2=2
      %1
   etat_2
etat_1


> pgi(pf, gen(pr_par));
%1


> pgci(pf, gen(pr_par));
etat_2=0
   %1
etat_2=1
   %1
etat_2=2
   %0
etat_2


> pg_c_invariant(pf, gen(pr_par));
etat_2=0
   %1
etat_2=1
   %1
etat_2=2
   %0
etat_2


> quit();
```

3.6 Expression of system properties in SIGNAL+

It was already seen how system properties can be declared in SIGALI by means of symbols, identifiers and indeterminates. The subsequent sections will explain how to verify these properties and synthesize the controller using SIGALI. Using an extension of the SIGNAL language, called SIGNAL+, it is also possible to express the properties to be checked, as well as the control objectives to be synthesized, in the SIGNAL program. The syntax is:

```
(| SIGALI(Objective(B_?(PROP))) |)
```

The keyword `SIGALI` means that the subexpression has to be evaluated by SIGALI. The function `B_?` will encode the "*value*" of the boolean *PROP* defined in the SIGNAL program, that we want to analyse (it can be either `B_True` or `B_False`, which means that we are interested in analyzing the set of states where the boolean `PROP` is *true* (resp. *false* The function `Objective` can be a verification objective: it can be `Always`, `C_Invariant`, `Reachable`, `Attractivity`, etc, or a control objective to be synthesized. we will come back to this point in the next section.

References

[1] L. Besnard, P. Bournai, T. Gautier, N. Halbwachs, S. Nadjm-Tehrani, and A. Ressouche. Design of a multi-formalism application and distribution in a data-flow context: an example. In *Proceedings of The 12th International Symposium on Languages for Intensional Programming, ISLIP' 99*, NCSR Demokritos, Athens, Greece, 28-30 June 1999.

[2] B. Dutertre, M. Le Borgne, and H. Marchand. SIGALI *: un système de calcul formel pour la vérification de programmes* SIGNAL *– Manuel d'utilisation*. Note technique, non publiée, Décembre 1998.

[3] T. Gautier, P. Le Guernic, and F. Dupont. SIGNAL V4 *: manuel de référence*. Publication Interne No 832, IRISA, June 1994.

[4] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. Publication Interne No 459, IRISA, 1989.

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

[6] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the SIGNAL environment. *Discrete Event Dynamical System: Theory and Applications*, October 2000.

[7] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. A design environment for discrete-event controllers based on the SIGNAL language. In *1998 IEEE International Conference On Systems, Man, And Cybernetics*, pages 770-775, San Diego, California, October 1998.

[8] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real time applications with SIGNAL. Publication Interne No 582, IRISA, April 1991.

[9] P. Bournai and P. Le Guernic. *Un environnement graphique pour le langage* SIGNAL. Publication Interne No 741, IRISA, September 1993.

[10] B. Dutertre. *Spécification et preuve de systèmes dynamiques : Application à* SIGNAL. PhD Thesis, Université de Rennes, December 1992.

[11] B. Dutertre and M. Le Borgne. Control of polynomial dynamic systems: an example. Publication Interne No 798, IRISA, January 1994.

[12] T. P. Amagbegnon, P. Le Guernic, H. Marchand, and E. Rutten. The Signal data flow methodology applied to a production cell. Publication Interne No 917, IRISA, March 1995.

[13] H. Marchand and M. Le Borgne. Partial order control and optimal control of discrete event systems modelled as polynomial dynamical systems over Galois fields. Publication Interne No 1125, IRISA, October 1997.

[14] M. Le Borgne, H. Marchand, E. Rutten, and M. Samaan. Formal verification of SIGNAL programs: Application to a power transformer station controller. In *Proceedings of AMAST'96*, Vol 1101 of *Lecture Notes in Computer Science*, pages 271-285, Springer-Verlag, Munich, Germany, July 1996.

[15] H. Marchand and M. Le Borgne. On the optimal control of polynomial dynamical systems over $\mathbb{Z}/p\mathbb{Z}$. In 4^{th} *International Workshop on Discrete Event Systems*, pages 385-390, Cagliari, Italy, August 1998.

[16] H. Marchand and M. Le Borgne. Partial order control of discrete event systems modelled as polynomial dynamical systems. In *1998 IEEE International Conference On Control Applications*, Trieste, Italy, September 1998.

[17] M. Le Borgne, B. Dutertre, A. Benveniste, and P. Le Guernic. Dynamical systems over Galois fields. In *Proc. ECC93*, Groningen, 1993.

[18] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems*, 77(1):pages 81-98, January 1989.

[19] C. Le Maire. *Le langage* SIGNAL *: un exemple en segmentation automatique de la parole continue*. INRIA Research Report 1217, Rennes, France, 1990.

[20] E. Rutten and P. Le Guernic. Sequencing Data Flow Tasks in SIGNAL. INRIA Research Report 2120, November 1993. Also in *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, Orlando, Florida, June 1994.

[21] E. Rutten and F. Martinez. SIGNALGTI, implementing task preemption and time interval in the synchronous data-flow language SIGNAL. In 7^{th} *Euromicro Workshop on Real Time Systems*, Odense(Denmark), June 1995.

[22] F. Maraninchi, Y. Rémond, and E. Rutten. Effective programming language support for discrete-continuous mode-switching control systems. *(submitted)*