Sigali User's manual

Hervé Marchand, Éric Rutten & Michel Le Borgne

March 30, 2004

Abstract

Sigali is a model-checking tool-based which manipulates *Polynomial Dynamical Systems (PDS)* (that can be seen as an implicit representation of an automaton) as intermediate models for discrete event systems. It offers functionalities for verification of reactive systems and discrete controller synthesis. It is developed jointly by Espresso[1] and Vertecs[2].

The techniques used consist in manipulating the system of equations instead of the sets of solution, which avoids the enumeration of the state space. Each set of states is uniquely characterized by a polynomial and the operations on sets can be equivalently performed on the associated polynomials. Therefore, a wide spectre of properties, such as liveness, invariance, reachability and attractivity can be checked. Many algorithms for computing predicates states are also available.

1 The model checker Sigali

1.1 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.

1.1.1 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*).

[1] Espresso Web Site: http://www.irisa.fr/espresso
[2] Vertecs Web Site: http://www.irisa.fr/vertecs

1.1.2 Dynamical systems: Basics

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

$$\left\{ \begin{array}{ccc} Q(X,Y) & = & 0 \\ X' & = & P(X,Y) \\ Q_0(X) & = & 0 \end{array} \right. \tag{3}$$

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.

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

1.2 The SIGALI commands & Operations

1.2.1 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()`:

- `quit();`

Loading the file of a model The `.z3z` (`.lib`) file which contains the 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 a file filename.z3z the command is:

- `read("filename");`

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.

- `chrono(true);` starts the clock. After each subsequent command, the time taken for the computation is displayed.
- `chrono(false);` stops the clock.

1.2.2 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 $a^2b + c^2$.

Variables can be declared 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).

The order of the variables corresponds to the order of declaration. For example, in the previous example, the order is $a < b < c < d$.
The command `indeter();` lists all the indeterminate symbols.

In order to declare variables in a given order, one can use the commands `declare_after`, `declare_first` or `declare_suff`. For example, if the variables $x < y < z$ are already declared, the command

- `declare_after(a,b,c)` will declare the variables according to the following order : $x < y < z < a < b < c$
- $a < b < c < x < y < z$ if you use `declare_first(a,b,c)`).
- `declare_suff([a,b,c])` will declare the variables `a__1,b__1,c__1,` in the following order: $a < a\_\_1 < b < b\_\_1 < c < c\_\_1$.

We can manipulate list of variables as follows: If `L_1` and `L_2` are two lists of variables, then

- `L : union_lvar(L_1,L_2)` is the list of variables which contains the variables of `L_1` and `L_2`.
- `L : inter_lvar(L_1,L_2)` performs the intersection of the two lists.
- `L : comp_lvar(L_1)` is the complementary list according to all the declared variable.
- `L : diff_lvar(L_1,L_2)` is equal to L_1\ L_2
- given a list of variables `L` and a variable `a`, `belong_lvar(a,L);` is *true* whenever `a` $\in$ `L`

1.2.3 Polynomials and equations

Polynomials We can write polynomial expressions, lists of polynomials, etc. All the usual polynomial operations are also available (+, -, *, ...). For example, the polynomial $a^2(-b - b^2)$ is written `a^2*(-b-b^2)`. A symbol can be assigned to a polynomial:

```
P : a^2*(-b-b^2);
```

Note that the variables `a,b` have to be declared first in order to specify this polynomial.
Now, given a polynomial P over the variables L,

- `varof(P)` gives access to the variable set of P
- `nbvar(P)` gives the number of variables of P
- `nb_solution(P,X)` gives the number of solution of the equation $P(X) = 0$, where X is a set of variables that must contains `varof(P)`.

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.):

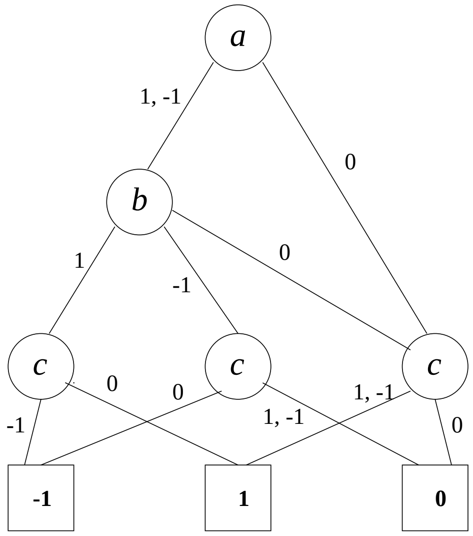

Figure 1: TDD representation of the polynomial $a^2b + c^2$.

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

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#`.
the command `size(P);` gives the number of nodes of the TDD that encodes the polynomial `P`.

Lists The list of polynomials, equations, etc. are written as follows: `[a+b,c+d]` is a list of polynomials and `[a+b=x,a*d^2=b^2]` is an equation system. Of course, a symbol can be assigned to a list or an equation system as well. For example:

- `list :  [a + b, a, b, 0, 1];`
- `equations :  [a ^2 = b ^2, c = a and b];`

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:

- `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`.

  Note that we can use the command `init_lconst` to declare the list of constants, e.g.

  `lconst : init_lconst(3,0);` is a list of 3 elements that are equal to 0, i.e. `lconst=[0,0,0]`.

- `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 the same function as their counterparts for each polynomial of the list.

1.2.4 (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 is `gen(lpoly);` 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]);`

If $P_1 = a+b-c$ and $P_2 = a^2-1$, then the previous command will compute the polynomial $P = P_1 \oplus P_2 = (P_1^2 + P_2^2)^2$, which entails that the solution of P will be the solution that are common to P_1 and P_2.

- `equal(p1,p2)` compares two polynomials `p1` and `p2` and test whether they are equal or not.

Complementation. Let `g` be a polynomial and V its set of solutions, 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:

- `subset(g1,g2);`

is `True` if and only if $V_1 \subseteq V_2$.

1.2.5 Existential/universal variable elimination

Given a polynomial P over the variables X, Y, then we define the Existential/universal variable elimination as follows

- `P' : exist(Y,P)` is a polynomial such that $Sol(P') = \{x | \exists y, P(x, y) = 0\}$
- `P' : forall(X,P)` is a polynomial such that $Sol(P') = \{x | \forall y, P(x, y) = 0\}$

1.2.6 Automatic reordering

The `set_reorder`(1) (exists also with the parameter 2) performs an automatic variable reordering using heuristics. This is very useful to decrease the size of the TDD. `set_reorder(0)` stops the automatic reordering.

1.3 Systems and Processes

SIGALI distinguishes between two categories of dynamical systems: *systems* and *processes*. *Systems* are general dynamical systems in which null transitions (basically self loops) are taken into account even when all the signals are absent, whereas in a *process*, null transitions are excluded i.e. No transition can take place in the absence all the signals. Dynamical systems can be automatically derived from either SIGNAL programs or Matou programs thus allowing to allowing the modeling of reactive systems by means of Mode Automata.
From a Signal/Matou programs, a file is automatically generated. it contains the following data:

- *events* is a list of variables encoding the event variables
- *states* is a list of variables which encodes the states variables
- *controllables* is a subset of `events` and corresponds to the controllable event variables (See section 3 for more details).
- *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).
- *constraints* is also a list of polynomial encoding the constraints part of the polynomial dynamical system (i.e. $Q(X, Y) = 0$).

If one want to construct from these sets a process (respectively a system), the following command has to be used.

- `syst : processus(events,states,evolutions,initialisations,constraints,controllables);`

Conversely, if `syst` is a dynamical system, as described by (3), constructed by the command `system` or `process`, then the 6 components of `syst` can be accessed by:

- `event_var(syst);` : returns the event variable set of a system, i.e. the vector Y
- `state_var(syst);` : returns the state variable set of a system, i.e. the vector X
- `evolution(syst);` : returns the vector of polynomials encoding the evolution equations, i.e. $[P_1(X, Y), \cdots, P_n(X, Y)]$
- `initial(syst);` : returns the polynomial encoding the initial states of the systems
- `constraint(syst);`: returns the constrains polynomial $Q(X, Y)$
- `controllable_var(syst);` : returns the controllable variable set of a system, i.e. the vector U with $U \subseteq Y$ (See section 3 for more details).

1.3.1 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.
- `all_succ(syst, g);` is the canonical generator of the set of states, such that *all* successors belong to E.

- `adm_events(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:

- `adm_states(syst, g);`
  is the canonical generator of the set of states compatible with at least one of the events in F.

1.3.2 Implicit System

Starting from a system modeled as an PDS S as described in Equation System (3), for some particular analysis, it is important to have access to the implicit corresponding implicit PDS of the form

$$\begin{cases} R(X, Y, X') = 0 \\ Q_o(X_o) = 0 \end{cases} \tag{4}$$

The SIGALI function that gives access to this new system is `implicit_sys`(syst). The result is an I-PDS (for implicit PDS). From a structure point of view, it is a 5-tuple (X, X', Y, R, Q_0) and the functions that gives access to the components of this I-PDS are respectively `state_var_I`(), `state_var_next_I`(), `event_var_I`(), `trans_rel_I`(), `initial_I`(), `"controllable_var_I`().
By loading the library `Orbite.lib`, you have access to the two following commands:

- `P : Orbite(S_Imp);` returns the set of reachable states of the implicit system S_Imp
- `S_1: Pruned((S_Imp,Orbite);` is an implicit Dynamical system, where all the states are reachable.

1.4 Fix-point Computation & Function definition

Fix point computation can also be performed. For example, given :

$$\begin{cases} p_0 & = & 0 \\ p_{i+1} & = & p^2 + 1 \end{cases}$$

the corresponding expression in SIGALI is

```
loop x=x^2+1 init 0;
```

Of course, such sequences do not always converge. This is not checked by the system.

New function construction. Starting from the existing functions, it is also possible to define new functions. The syntax is the following:

```
def f(x,y,z) :
        with
            intern_1 =   sigali expression,
            intern_2 =   sigali expression
        do
SIGALI body of the function   ;
```

For example, assume we want to compute the set of reachable states from Sol(P) in one step of an implicit system S_I. The corresponding function `succ(P)` is the following:

```
def succ(P)  :
  with
     X = state_var_I(S_I),
     X_Next = state_var_Next_I(S_I),
     Y = event_var_I(Y),
     Rel = trans_rel_I(S_I)
  do
   rename(exist(X, intersection(P,exist(Y, Rel)),X_Next,X));
```

1.5 Cost functions

SIGALI also offers the possibility to manipulate integers. Let $X = (x_1, \ldots, X_n)$ be declared variables of the system. Then, a cost function is a map from $(\mathbb{Z}/_{3\mathbb{Z}})^n$ to $\mathbb{N}$, which associates to each $x = (x_1, \cdots, x_n)$ of $(\mathbb{Z}/_{3\mathbb{Z}})^n$ some integer k. When $f(x)$ is not defined then we assume that $f(x) = \infty$. To encode these functions, we make the use of the ADD (Arithmetic decision diagrams). The ADD are similar to the TDD expect that we attach integers to the leaves of the ADD. For example, let $X < Y$ be two variables in $\mathbb{Z}/_{3\mathbb{Z}}$ and f a cost function such that

X	0	0	0	1	1	1	-1	-1	-1
Y	0	1	-1	0	1	-1	0	1	- 1
f	3	2	5	6	3	4	3	8	3

Then the ADD that represents the function f is given by the graph of Figure 2 :

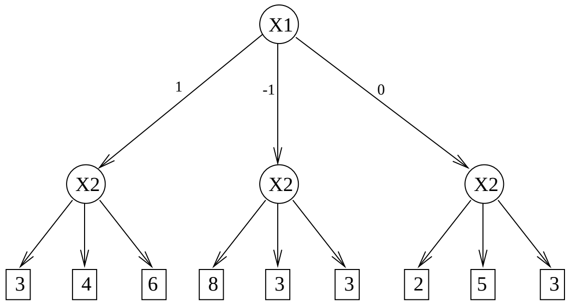

Figure 2: Exemple d'un ADD

In order to build and manipulate cost functions, the following SIGALI operations are available.

- `a_const(n)` build the constant function equal to the integer n
- `a_var(X,n1,n2,n3)`: given a declared variable `X` and 3 integers, `f: a_var(X,n1,n2,n3)` build the cost function such that
$$\begin{cases} X &= 1 &\Leftrightarrow& c_i(X) &= a \\ X &= -1 &\Leftrightarrow& c_i(X) &= b \\ X &= 0 &\Leftrightarrow& c_i(X) &= c \end{cases}$$
- Given two cost functions f_1 and f_2, one can perform the sum and the product of f_1 and f_2 by simply using the classical operators $+$ and $*$.
- `a_min(f1,f2)` is such that $\forall x \in \mathbb{Z}/_{3\mathbb{Z}}{}^n$, `a_min`$(f_1(x), f_2(x)) = min(f_1(x), f_2(x))$
- `a_max(f1,f2)` is such that $\forall x \in \mathbb{Z}/_{3\mathbb{Z}}{}^n$, `a_max`$(f_1(x), f_2(x)) = max(f_1(x), f_2(x))$
- `a_part(P,f1,f2,f3` build a cost function such that `a_part(P,f1,f2,f3`(x) = f1 if P(x)=0, f2 if P(x)=1 and f3 if P(x)=-1
- `a_margmin`(f,Y). given a cost function f defined over $X \cup Y$, `a_margmin`(f,Y) is a function, say `f'` over the variables X such that $f'(X) = min_Y(f(X,Y)$, i.e. $\forall y$, $f'(x) < f(x,y)$
- `a_margmax` (same as `a_margmin` but with the max)
- `P: a_iminv(f,n)` is a polynomial such that $P(x) = 0 \Leftrightarrow f(x) = n$
- `a_maxim(f)` is the minimum value taken by f and `a_minim` is the maximum value taken by f
- `P: a_inf(f,n)` is a polynomial such that $P(x) = 0 \Leftrightarrow f(x) \leq n$ (`a_sup` is also defined)
- `R: a_cost2rel(f1,f2)`. Given two cost functions over X1 and X2 (with the same cardinality) `a_cost2rel(f1,f2)` is a polynomial $R(X_1, X_2)$ such that $R(x_1, x_2) = 0 \Leftrightarrow f1(x_1) \leq f2(x)$.

2 Verification of systems using SIGALI

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

2.1 Liveness

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:

- `alive(syst);`

is `True` if and only if `syst` is **alive**.

2.2 Safety Properties

2.2.1 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 `system`.

2.2.2 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,

- `Invariant_under_control(syst, g);`

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

2.2.3 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:

- `greatest_inv(syst, g);`
- `greatest_c_inv(syst, g);`

gives the canonical generator of F'.

2.3 Reachability Properties

2.3.1 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,

- `Reachable(syst, g);`

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

2.3.2 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`.

3 Synthesis of controllers using Sigali

3.1 Essentials of the control synthesis problem

For **controllable** polynomial dynamic systems, the set of events Y can be partitioned into two sets Y and U, where,

- Y is the set of **uncontrollable** events,
- U is the set of **controllable** events.

The PDS can now be written as:

$$\left\{ \begin{array}{ccc} Q(X,Y,U) & = & 0 \\ X' & = & P(X,Y,U) \\ Q_0(X) & = & 0 \end{array} \right.$$

Let n, m, and p be the respective dimensions of X, Y, and U. The trajectories of a **controlled** system are sequences (x_t, y_t, u_t) in $(\mathbb{Z}/3\mathbb{Z})^{n+m+p}$ such that $Q_0(x_0) = 0$ and, for all t, $Q(x_t, y_t, u_t) = 0$ and $x_{t+1} = P(x_t, y_t, u_t)$. The events (y_t, u_t) include an uncontrollable component y_t and a controllable component u_t.

The controller: The PDS can be controlled by first selecting a particular initial state x_0 and then by choosing suitable values for $u_1, u_2, ...$ Here, we only consider **static** control policies where the value of the control u_t is instantaneously computed from the value of x_t and y_t. Such a controller is called a *static controller*. Formally, it is a system of two equations:

$$\left\{ \begin{array}{ccc} C(X,Y,U) & = & 0 \\ C_0(X) & = & 0 \end{array} \right.$$

where the latter equation determines the initial states satisfying the control objectives and the former describes how to choose instantaneous controls. When the controlled system is in state x, and an event y occurs, any value u such that $Q(x, y, u) = 0$ and $C(x, y, u) = 0$ can be chosen. The behavior of the system composed with the controller is then modelled as:

$$\left\{ \begin{array}{ccc} Q(X,Y,U) & = & 0 \\ C(X,Y,U) & = & 0 \\ X' & = & P(X,Y,U) \\ Q_0(X) & = & 0 \\ C_0(X) & = & 0 \end{array} \right.$$

Control objectives ensuring properties like **invariance**, **reachability**, **attractivity**, etc are called *traditional* control objectives. There are also other kinds of control objectives which can be expressed as partial order relations over the states of the PDS. These are called *optimization* control objectives.

Sigali provides functionalities for synthesis of controllers ensuring *traditional* as well as *optimization* control objectives. There does not exist pre-existing Sigalifunctionalities. Instead, one have to load different libraries in which Sigali functions are written.

3.2 Loading of the necessary libraries

For controller synthesis ensuring *traditional* control objectives, the `Synthesis.lib` file must be loaded.

- `S_c : S_Invariance(S,prop);` (or equivalently `S_Security(S,prop);` If *prop* encodes a set of states E, `S_Invariance(S,prop)` computes a controller that ensures the invariance of E with respect to the system S. The controlled system is the output of this function.
- `S_c : S_Reachbale(S,prop);` If *prop* encodes a set of states E, `S_Reachable(S,prop)` computes a controller that ensures the reachability of E from the initial states. The controlled system is the output of this function. To ensure the attractivity, one have to use the `S S_Attractivity(S,prop);` command.

For dealing with *optimization* control objectives, two additional files: `Synthesis_Partial_order.bib` and `Synthesis_Optimal_Control.bib` must also be loaded.

- The file `Synthesis_Partial_Order.lib` contains the definition of a function called `S_Free_Max` which helps in choosing a control such that the system evolves, in the next instant, into a state where the maximum number of uncontrollable events are admissible.
- The file `Synthesis_Partial_Order_Relation.lib` contains function definitions for the synthesis of optimal controllers. The goal is to synthesize a controller that will choose a control from amongst all the admissible controls in such a way that the system evolves into a state according to a given choice criterion. This criterion is expressed as a cost function relation on the set of states. Intuitively speaking, the cost function is used to express priority between the different states that a system can reach in one transition.

  Technical Restiction

  Input: C(X) is the cost function used for the control C_Dup(X__1) is the duplicated cost function of C(X) where X__1 is, for example obtained as follows:

  `Sigali> duplicate_states :  declare_suff(state_var(S));`

  next, one have to declare C_Dup (i.e. same as for C but with the variables of the set `duplicate_states`

  Nb. No automatic reordering -> set_reorder(0) the variable order must be as follows X1>X1__1>X2>X2__1... So, if you plan to use the functions of this library then never use the reodering after the use of declare_suff(state_var(S)) Once you plan not to use these functions anymore, then you activate again the automatic reordering.

  The different functions of the `Synthesis_Partial_Order_Relation.lib` are :

  - `Supervisor_Lower_than(S,C,C_Dup,duplicate_states)` gives access to a controlled system such that whatever the current position of the system S under control, the supervisor will make the system evolve into the state x such that forall x' reachable from the current position $C(x) \leq C\_Dup(x')$. `Lower_than(S,C,C_Dup,duplicate_states)` build the controlled system.
  - idem for `Supervisor_Greater_than(S,C,C_Dup,duplicate_states)` and `Greater_than(S,C,C_Dup,duplicate_states)` (i.e. $C(x) \geq C\_Dup(x')$).
  - idem for `Supervisor_Striclty_Lower_than(S,C,C_Dup,duplicate_states)` and `Striclty_Lower_than(S,C,C_Dup,duplicate_states)` (i.e. $C(x) < C\_Dup(x')$).
  - idem for `Supervisor_Striclty_Greater_than(S,C,C_Dup,duplicate_states)` and `Striclty_Greater_than(S,C,C_Dup,duplicate_states)` (i.e. $C(x) > C\_Dup(x')$).