Specifying and verifying a transformer station in Signal and SignalGTi

Hervé Marchand, Eric Rutten, Mazen Samaan

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Specifying and verifying a transformer station in Signal and SignalGTi

HERVÉ MARCHAND, ÉRIC RUTTEN, MAZEN SAMAAN

N° 2521

March 1995

PROGRAMME 2

ISSN 0249-6399

Specifying and verifying a transformer station in Signal and SignalGTi *

HERVÉ MARCHAND **, ÉRIC RUTTEN ***, MAZEN SAMAAN ****

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

Rapport de recherche n ° 2521 — March 1995 — 34 pages

Abstract: We present the specification and verification of the automatic circuit-breaking behavior of an electric power transformer station using the synchronous approach to reactive real-time systems, and particularly the SIGNAL language. The synchronous languages have a mathematical model supporting the various phases of the development of a control system: specification, verification, simulation, code generation, and implementation. Their semantics are at the base of the various tools dedicated to each phase and integrated into homogeneous design environments. The methodology associated with the data flow language SIGNAL is demonstrated on the example of the functional description of a medium tension power transformer station and the specification of its automatic behavior upon the occurrence of an electrical default. The specification of the complex hierarchical, state-based and preemptive behavior is made in SIGNAL*GTi*, an extension of SIGNAL with the notions of time intervals and preemptive tasks. For the validation of the specification, a graphical simulator is generated using the execution environment for SIGNAL. Properties required by the application are proved to hold on the specification of the controller, using the proof method associated with SIGNAL.

Key-words: Power systems, control systems, formal methods, real-time, synchronous language, data flow, task preemption, formal methods, verification tools, software safety.

(Résumé : tsvp)

*Our work in verification is supported by Électricité de France.
**INRIA e-mail hmarchan@irisa.fr
***INRIA e-mail rutten@irisa.fr
****EDF/DER service EP département CCC, 6 quai Watier 78401 Chatou Cedex France

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

Spécification et vérification d'un poste de transformation électrique en SIGNAL

Résumé : Nous présentons ici la spécification et la vérification du contrôleur automatique d'un disjoncteur d'un poste de transformation électrique, en utilisant l'approche synchrone pour les systèmes réactifs temps réels et plus particulièrement le langage SIGNAL. Les langages synchrones sont basés sur une modèle mathématique servant de support aux différentes phases de développements du système contrôlé: spécification, vérification, simulation, génération de code, et implémentation. Leurs sémantiques sont à la base d'outils divers dédiés à chaque phase et intégrés dans des environnements de conception homogènes. La méthodologie associée au langage flots de données SIGNAL est décrite par un exemple de description fonctionnelle d'un poste de transformation électrique et la spécification de son comportement automatique à l'apparition d'un défaut électrique. La spécification du comportement complexe avec hiérarchie de préemption est faite en SIGNAL*GTi*, une extention de SIGNAL avec les notions d'intervalles de temps et de tâches préemptives. La validation de la spécification est réalisée à l'aide d'un simulateur graphique, généré par l'environnement d'exécution de SIGNAL. Les propriétés requises par l'application sont prouvées en tenant compte de la spécification du contrôleur, à l'aide d'une méthode de preuve associée à SIGNAL.

Mots-clé : système de contrôle, temps réel, langage synchrone, flots de données, préemption de tâches, outils de vérification, méthode formelles, sûreté du logiciel.

1 Introduction

This paper presents an experiment in the *synchronous approach* to the specification, implementation and formal verification of *reactive real time systems* [4], and more specifically of the declarative language SIGNAL. It is applied to the design of a complex state-based, discrete event behavior of the controller of a power transformer station.

An interpretation of the synchrony hypothesis is that all the relevant values involved in a computation (input, output and internal values) are present simultaneously within the single instant of logical time when the system reacts to its input. This condition is satisfied for systems which react faster than the pace of their environment's dynamics, for instance in synchronous hardware. It guarantees determinism of the specified behaviors, and facilitates the semantical manipulations on programs, which are used in the definition of program transformations preserving equivalence of behavior. Its advantage is to provide support for a whole set of tools assisting the design of real-time applications, all along their life-cycle. The same formal bases support the operations performed by the various tools in the design environments: the analysis at the different levels of abstraction, from requirements down to code generation, the implementation on specific hardware by co-design, the performance evaluation and optimization, ... A family of languages is based on this hypothesis [9], featuring amongst other ESTEREL, LUSTRE, SIGNAL and also STATECHARTS. They provide complete design environments, with sets of tools based upon their formal semantics, and which support specification, formal verification, optimisation and generation of executable code. Their aim is to support the design of safety critical applications, especially those involving signal processing and process control. The synchronous technology and its languages are available commercially, and applied in industrial contexts [3]. Parallelly, research and experiments are going on on new features, such as the one reported on in this paper.

SIGNAL is a real-time synchronized data-flow language [11]. Its declarative style is based on equations defining the values and the synchronizations of the flows of data called signals. Processes are in fact systems of equations, and compiling a SIGNAL program involves transforming the specification into an executable code solving this system of equations at each reaction. The compilation also performs checkings on the causal and temporal consistency of the specification, and optimizations. The SIGNAL programming environment features graphical editor and simulation tools, compiler and optimization, code generation in several target languages, and a proof tool for the analysis of dynamical systems. The original equational nature of SIGNAL makes that it relies on a formal model in terms of polynomial dynamic equations systems, and the proof method is based on the theory of algebraic geometry [12, 6]. Given the synchronous data flow style of SIGNAL, its model of time is based on instants, and its actions are performed within the instants; SIGNAL*GTi* is a recent extension that provides constructs for the specification of hierarchical preemptive tasks on intervals of time [16].

We consider in this paper the application of SIGNAL and SIGNAL*GTi* to the specification, simulation and verification of the automatic control system of a power transformer station. It concerns the response to electric defaults on the lines traversing it. It involves complex interactions between communicating automata, interruption and preemption behaviors, timers and timeouts, reactivity to external events, ... The functionality of the

controller is to handle the interruption of current, the redirection of supply sources, and the re-establishment of current following an interruption. The objective is double: protecting the components of the transformer itself, and minimizing the default in the distribution of power in terms of duration and size of the interrupted sub-network. The electric defaults can be detected by sensors; the controller has to distingish between several types of defaults, and between transient and persistent ones. This selection involves a protocol, with a cycle of attempts at treating the default in reaction to perceived events.

In the remainder of this paper, the synchronous approach to real time systems and its programming methodology are presented in the light of the application. A detailed account of the power system itself [14] is outside the scope of this paper. Section 2 presents the SIGNAL language and the time intervals and tasks of SIGNAL*GTi*. Section 3 gives a general description of the power transformer station and the behavior of one of its cells, and details its specification in SIGNAL and SIGNAL*GTi*. The verification methods associated with SIGNAL is described in Section 4, and its application to the specification in Section 5. Discussion on results and related work is given in Section 6.

2 The synchronous data flow language SIGNAL and its extension SIGNAL*GTi*

2.1 The SIGNAL equational data-flow real-time language

SIGNAL [11] is a synchronous real-time language, data flow oriented (*i.e.,* declarative), built around a minimal kernel of operators. This language manipulates signals, which are unbounded series of typed values, with an associated clock determining the set of instants when values are present. For instance, a signal `X` denotes the sequence $(\mathtt{x}_t)_{t\in T}$ of data indexed by time t in a time domain T. Signals of a special kind called `event` are characterized only by their clock *i.e.,* their presence. Given a signal `X`, its clock is noted as `event X`, meaning the event present simultaneously with `X`. 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. Systems of equations on signals are built using a composition construct. 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.

Kernel of the language. The kernel of the SIGNAL language is based on four operations, defining primitive processes, and a composition operation to build more elaborate ones.

- *Functions* are instantaneous transformations on the data. For example, the definition of a signal Y_t by the function f:

$$\forall t,\ Y_t = f(X_{1_t}, X_{2_t}, \ldots, X_{n_t})$$

is written in SIGNAL: `Y := f{ X1, X2,..., Xn}`. The signals `Y`, `X1`,..., `Xn` are required to have the same clock.

- *Selection* of a signal `X` according to a boolean condition `C` is: `Y := X when C`. The operands and the result do not generally have identical clock. Signal `Y` is present if and only if `X` and `C` are present at the same time and `C` has the value `true`; when `Y` is present, its value is that of `X`.
- *Deterministic merge*: `Z := X default Y` defines the union of two signals of the same type. The clock of `Z` is the union of that of `X` and that of `Y`. The value of `Z` is the value of `X` when it is present, or that of `Y` if it is present and `X` is not.
- *Delay*, a "dynamic" process giving access to past values of a signal. For example, the equation: $ZX_t = X_{t-1}$, with initial value V_0 defines a dynamic process. It is encoded by: `ZX := X$1` with initialization `ZX init V0`. Signals `X` and `ZX` have the same clock.

  Derived operators include delay on N instants (`$N`), and a `window M` operation giving access to a whole window of past values (from instants $t - M$ to t), as well as combinations of both operators.
- *Composition* of processes is the associative and commutative operator "|" denoting the union of the underlying systems of equations. In SIGNAL, it is written: `(| P1 | P2 |)`, for processes P_1 and P_2. It can be interpreted as parallelism with signals supporting instantaneous communication between processes.

Derived features and example. Derived processes have been defined from the primitive operators, providing programming comfort. E.g., the instruction `synchro{X,Y}`, which specifies the synchronization of signals `X` and `Y`; `when C` giving the clock of occurrences of `C` at the value `true`; `X cell B` which memorizes values of `X` and outputs them also when `B` is true; the expression `C := # E` is a counter of occurrences of event `E` behaving like the example given just below. Arrays of signals and of processes have been introduced as well. Hierarchy, modularity and re-use of processes are supported by the possibility of defining process models, and invoking instances.

An example of a SIGNAL process is the following counter named `COUNT` (which is the expanded form of the derived operation `C := # E`):

```
process COUNT= {? E ! C}
  (| C := ZC+1
   | ZC := C$1
   | synchro{C,E}
   |)
  where ZC init 0
end
```

It has one input signal `E`, and an output `E`. The value of the counter is `C`, and it is defined as the previous value `ZC` incremented by one. The previous value `ZC` is declared locally, and

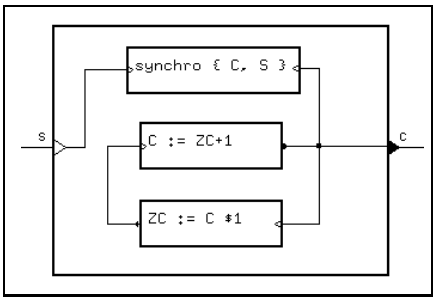

Figure 1: A counter in SIGNAL.

defined using the delay operator on signal `C`; its initial value is the parameter `V0`. It counts the occurrences of the input event `E`: this is obtained by synchronizing the presence of the counter, and by this the incrementation operation itslef, with the occurrences of `E`.

Programming environment. The SIGNAL compiler performs the analysis of the consistency of the system of equations, and determines whether the synchronization constraints between the clocks of signals are verified or not. It is based on an internal representation featuring a graph of data dependencies between operations, augmented with temporal information coming from the clock calculus. If the program is constrained so as to compute a deterministic solution, then executable code (in C or FORTRAN) can be automatically produced. The complete programming environment also contains a graphical, block-diagram oriented user interface where processes are boxes linked by wires representing signals, as illustrated in Figure 1.

The SIGNAL environment also includes a *proof system*, to verify dynamic properties of programs, involving state information (in the delayed signals) and transitions in reaction to occurrences of other signals. Hence, it is possible to formally specify and verify the satisfaction of dynamical properties of the behaviors. This aspect will be detailed in Section 4.

2.2 Task preemption in SIGNAL*GTi*

SIGNAL*GTi* is a recent extension to SIGNAL, handling tasks executing on time intervals and their sequencing [16]. The interest of this is that data flow and sequencing aspects are both encompassed in the same language framework, thus relying on the same model for their execution and analysis (for the compilation and verification of correctness of programs). In this approach, a data flow application is considered to be executed from an initial state of its memory at an initial instant α, which is before the first event of the reactive execution. A data flow process has no termination specified in itself: therefore its end at instant ω can only be decided in reaction to external events or the reaching of given values. Hence ω is part

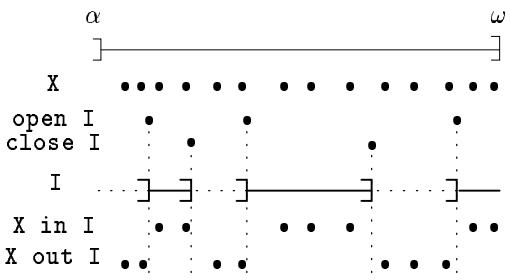

Figure 2: Time intervals sub-dividing time.

of the execution, and the time interval on which the application executes is the left-open, right-closed interval `]`α`,`ω`]`.

Time intervals. They are introduced in order to enable the structured decomposition of the interval `]`α`,`ω`]` into sub-intervals as illustrated in fig. 2, and their association with processes [16]. Such a sub-interval `I` is delimited by occurrences of bounding events `B` at the beginning and `E` at the end. It has the value `inside` between the next occurrence of `B` and the next occurrence of `E`, and `outside` otherwise. It has an initial value `I0` (`inside` or `outside`). This is written: `I := ]B, E] init I0`. Like `]`α`,`ω`]`, sub-intervals are left-open and right-closed. This choice is coherent with the behavior expected from reactive automata: a transition is made according to a received event occurrence and a current state, which results in a new state. Hence, the instant where the event occurs belongs to the time interval of the current state, not to that of the new state. The operator `compl I` defines the complement of an interval `I`, which is `inside` when `I` is `outside` and reciprocally. Operators `open I` and `close I` respectively give the opening and closing occurrences of the bounding events. Occurrences of a signal `X` inside interval `I` can be selected by `X in I`, and reciprocally outside by `X out I`. In this framework, `open I` is `B out I`, and `close I` is `E in I`.

Tasks. They consist in associating a (sub)process of the application with a (sub)interval of `]`α`,`ω`]` on which it is executed. Traditional processes in SIGNAL are tasks active on `]`α`,`ω`]`: they are *persistent* throughout the whole application. Inside the task interval, the task process is active i.e., present and executing normally. Outside the interval, the process is absent and the values it keeps in its internal state are unavailable. In some sense it is out of time, its clock being cut. Tasks are defined by the process `P` to be executed, the execution interval `I`, and the starting state (current, or initial) when (re-)entering the interval. More precisely, the latter means that, when re-entering the task interval, the process can be re-started from its current state at the instant where the task was *suspended* (*i.e.,* in a temporary fashion): this is written `P on I`. Alternately, it can be re-started from its initial state as defined by the declarations of all its state variables, if the task was *interrupted*

E	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
C	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
I	i	i	i	i	o	o	o	o	i	i	i	i	i	i	i	o	o	o	i	i	i
C1	1	2	3	4					5	6	7	8	9	10	11				12	13	14
C2	1	2	3	4					1	2	3	4	5	6	7				1	2	3

Table 1: Traces for the task of counting E, on and each interval I (i means inside; o: outside).

(meaning: aborted in a definitive fashion): `P each I`. The processes associated with intervals can themselves be decomposed into sub-tasks: this way, the specification of *hierarchies* of complex behaviors is possible.

Task sequencing and preempting. It is achieved as a result of constraining intervals and their bounding events, and associating activities to them by constructing hierarchical tasks. *Parallelism* between several tasks is obtained naturally when tasks share the same interval, or overlapping intervals. *Sequencing tasks* then amounts to constraining the intervals of the tasks, by constraining their bounding events. Using `on` and `each`, as defined above, enables control of activities and more elaborate behaviors can be specified. This way, it is possible to specify hierarchical parallel place/transition systems.

For example, a counter behaving as defined previously can be associated with an interval `I`, into a task behaving as illustrated by the traces in Table 1, where `C1 := COUNT{E} on I` and `C2 := COUNT{E} each I`. We can also define a task that can be started, suspended, resumed, and ended (i.e. interrupted) (respectiveley by events `b`, `s`, `r` and `e`). This is achieved very simply using a combination of `on` and `each`:

```
process COUNTER =
  {? event Unit, B, E, S, R ! C}
  (| (| C := COUNT{Unit} on CI
     | CI := comp I
     | I := ]S,R]
     |) each J
   | J := ]B, E]    |)
  where interval CI, I init outside,
                 J init outside
end
```

This is an example of a hierarchy of tasks. A trace of a counter controlled in this way is given in table 2.

It shows how a new counter starts on each interval `]b,e]`, while being inactive between suspension `s` and resuming `r`.

The encoding of time intervals and tasks into the SIGNAL kernel [16] is implemented as a pre-processor to the SIGNAL compiler, called SIGNAL*GTi* [17]. Applications other than the one discussed in this paper concern an interactive reflex game and the sequencing of visual servoing tasks in a robot vision system [13].

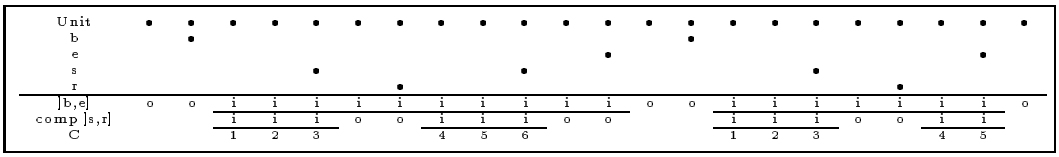

Table 2: Trace for the task of counting, controlled by events `b`, `s`, `r` and `e`.

3 Specification of the power transformer station

3.1 General description of a power transformer station

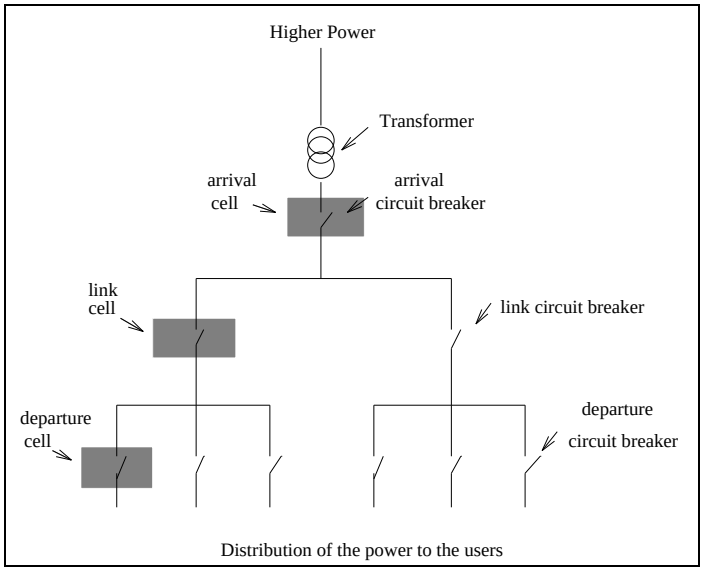

Figure 3: The topology of an electric power transformation post.

The transformer station on the electric network. The purpose of an electric power transformer station is to lower the voltage of power so that it can be distributed in urban centers. The kind of transformer we are interested in receives high voltage lines, and several medium voltage lines come out of it and distribute power to end-users [14], as illustrated in Figure 3. For each high voltage line, a transformer lowers the voltage. In the course of exploitation of this system, several kinds of electrical defaults can occur (three types of electrical defaults are considered: phase `PH`, homopolar `H`, or wattmetric `W`), due to causes internal or external to the station. In order to protect the device and the environment, several circuit breakers are placed in a network of *cells* in different parts of the station (on

the arrival lines, link lines, and departure lines). These circuit breakers are alerted by sensors at different locations on the lines.

Each circuit breaker controller defines a behavior beginning with the confirmation and identification of the type of the default. If the default is confirmed, the treatment consists in opening the circuit-breaker during a given delay, then closing it again, and after another delay, if the default is still present, then repeating these operations for a certain number of cycles. The purpose of this is to treat transient defaults; in case the default is still present at the end of the cycle, the circuit-breaker is opened definitely, and the control is given to the remote operator.

One of the problems is to know which of the circuit breakers must be opened. If the default appears on the departure line, it is possible to open the circuit breaker at departure level, or at link level, or at arrival level. Obviously, it is in the interest of users that the circuit be broken at departure level, and not at a higher level, so that the fewest users are deprived of power. This requires coordination between the different circuit breaker cells.

Functional description of a departure cell. We will focus on one of the types of cell: the departure cell, because it features all the interesting aspects of the automatism behavior, even in this simplified presentation. Other cells have a behavior which is only part of this one. It is decomposed in a confirmation process, which also cares for identifying the type of the default, followed by a treatment phase in an attempt at making the default disappear.

The *confirmation phase* consists in taking the time to let transient defaults cease naturally. For each of the default types (`PH`, `H`, or `W`) a delay can assess its persistent presence. They are tested in sequence, until a default is confirmed (i.e., present at the end of the corresponding delay). However, the sequence is interrupted as soon as the default disappears, or a previously examined default appears. This latter point introduces a hierarchy of preemption as explained in Section 3.2.1.

The *treatment phase* begins when the default is confirmed. It alternates between breaking the circuit during varying delays, and closing it again to check whether the default has disappeared. Each circuit breaking begins with emitting the command to open the circuit breaker, followed by the reception of the `Open` event, upon which the current delay is started . Upon completion of the delay, the closing of the circuit breaker is required, and confirmed by the reception of `Closed`. Once the circuit is re-established, either the default has disappeared, and the cell goes into its normal state, or it is still present: then, the treatment phase goes into the next cycle after a .5s delay, or if it was the last cycle, the circuit breaker is definitely broken, and its management is left to a remote control operator. The series of delay values (in the first cycle: .3s, in the second: 15s, in the third: 30s). is treated as a signal, and the cycle is repeated as a series of activation intervals as explained in Section 3.2.2.

In the following, we give a simplified and shortened description of the power system that we are considering, keeping only the most important aspects of the original specification [14]. For example, we consider only one source of external interruption instead of several, because the mechanisms involved are the same.

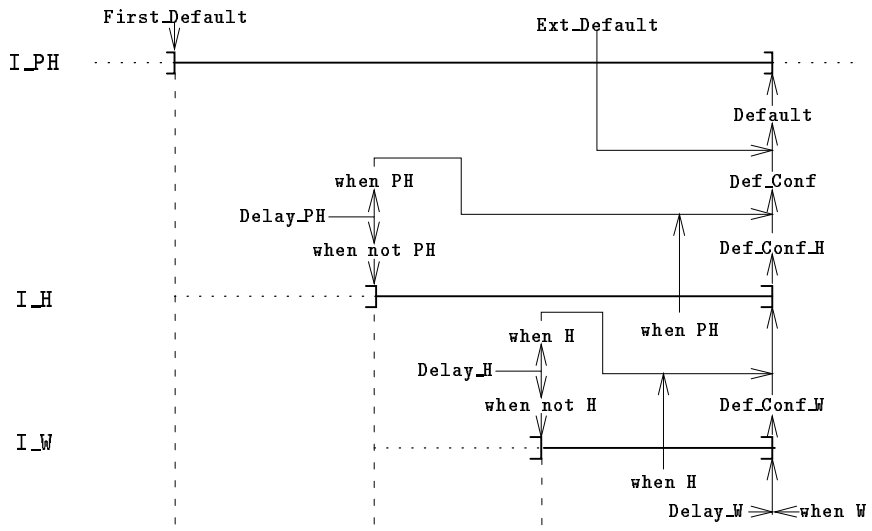

Figure 4: The confirmation phase: an interruption hierarchy.

3.2 Specification in SIGNAL and SIGNAL*GTi*

3.2.1 The confirmation phase: an interruption hierarchy

Figure 4 illustrates the `Confirmation` process specified in SIGNAL*GTi* in Figure 3. The three constant parameters `Delay_PH`, `Delay_H`, and `Delay_W` correspond to each of the three kinds of electrical defaults. The input event `Time` is the base clock i.e., it is the clock of the logical inputs `PH`, `H`, and `W` (presence of the defaults) and contains the clocks of the two other input events `Ext_Default` and `First_Default`. The process emits the output event `Def_Conf` when the default is confirmed and the output logical signal `Default` which gives the state of the cell. This latter is *true* when an external default is detected (reception of `Ext_Default`) or when the default is confirmed (`Def_Conf`), otherwise it is `false` when a default is not present. The interruption hierarchy is as follows:

- when a default is detected (`First_Default`), the interval `I_PH` is entered. It is closed by the occurrence of `Default` (causing the interruption of every sub-activity). Each time this interval is entered, a sub-process performs the confirmation of the default as follows: a counter of `Time` is activated during the delay `Delay_PH`. At the end of this delay:
  - if the logical `PH` is `true`, or if `PH` becomes *true* during the sub-interval `I_H` or if the default is confirmed at a lower level, then the default is confirmed at this level,

with the emission of an event `Def_Conf`. This will cause the interval `I_PH` to close, thereby also terminating the confirmation task.

– In the other case (`PH` is `false`), the interval `I_H` (which is a sub-interval of `I_PH`) is entered, and a sub-process performs the rest of the confirmation, in a quite similar way: a counter of `Time` is activated for the delay `Delay_H`, and:

  * if `H` is `true`, or becomes so during interval `I_W`, or if the default is confirmed at a lower level, the default is confirmed at this level, with the emission of `Def_Conf_H` causing the interval to close.
  * In the other case (`H` is `false`), the interval `I_W` (which is a sub-interval of `I_H`) is entered, and a last sub-process tries to confirm the default `W` by counting `Delay_W` and testing for `W`.

The code specifying this behavior in SIGNAL *GTi* is given in Table 3, while Figure 4 illustrates it.

```
process Confirmation = ( integer Delay_PH, Delay_H, Delay_W )
            {? logical PH, H, W; event Ext_Default, First_Default, Time
             ! event Def_Conf; logical Default }
(| Default := Ext_Default default Def_Conf default (not (when not Def))
 | Def := (PH or H or W)
 | I_PH := ] First_Default, when Default ]
 |        % confirming a PH default %
   (| End_Delay_PH := when ((#Time) = Delay_PH)
    | Def_Conf := (when PH when End_Delay_PH)
                  default (when PH in I_H) default Def_Conf_H
    | I_H := ] (when (not PH) when End_Delay_PH), Def_Conf_H ]
    |        % confirming an H default %
        (| End_Delay_H := when ((#Time) = Delay_H)
         | Def_Conf_H := (when H when End_Delay_H)
                         default (when H in I_W) default Def_Conf_W
         | I_W := ](when (not H) when End_Delay_H), Def_Conf_W]
         |        % confirming a W default %
              (| End_Delay_W := when ((#Time) = Delay_W)
               | Def_Conf_W := when W when End_Delay_W
               |)   each I_W
         |)  each I_H
   |)  each I_PH   |)
  where
     interval I_PH init outside, I_H init outside, I_W init outside;
     event Def_Conf_H, Def_Conf_W; logical Def
end %Confirmation%
```

Table 3: The `Confirmation` process.

3.2.2 The treatment phase: a series of intervals

Once the default is confirmed, the behavior enters a treatment phase constituted of a cycle of attempts at making the default disappear by cutting the circuit breaker. Each time, it is the process `Cycle` (see Table 4) which is activated.

The process `Cycle` is parameterized with a delay `Delay_C` (the duration between two circuit cut-offs in the cycle), an integer `N` (the maximum number of attempts) and an array `Delays` of `N` values for the duration of breaker opening in each attempt. Its input signals are the events `Opened` and `Closed` signalling the actual opening (respectively: closing) of the circuit breaker, the event `Time` (which is used to count the different delays), and a logical signal `Def` (the disjunction of the three default types: it is `true` if any of them is). The event `Time` is the base clock i.e., it is the clock of the logical input and contains the clocks of `Opened` and `Closed`. The output signals are the events `Req_Open_Cycle` and `Req_Close` requesting respectively the opening and closing of the circuit breaker, the event `Def_Break` signalling that the circuit breaker should be opened definitively (because the treatment could not handle the default: another, non-automatic procedure should be applied), and the event `End_Default` signalling that the default has disappeared.

```
process Cycle  = ( integer Delay_C; integer N; [N] integer Delays )
           {? event Opened, Closed, Time; logical Def
            ! event Req_Open_Cycle, Req_Close, Def_Break, End_Default}
 (| I_Break := ]Req_Open_Cycle, Closed]
  |   (| N_Cycle := (#(open I_Break))+1
       % break definitely when number of cycles exceeded %
       | Def_Break := when (N_Cycle > N)
       | Current_D := Delays[N_Cycle]
       | Delay := Current_D cell Time
       |)
  |   (| I_Open := ]Opened, Closed]
       |  %request closing when Delays[N_Cycle] elapsed after opening%
           (| Req_Close := when ((#Time) = Delay)
            |)    each I_Open
       |)
       each I_Break
  | End_Default := when not (Def out I_Break)
  |   % request opening when Delay_C elapses outside of I_Break %
      (| Req_Open_Cycle := when ( (#Time) = Delay_C )
       |)   each comp I_Break
  |)
where
  interval I_Break init inside, I_Open init outside;
  integer Delay init Delays[1], Current_D init Delays[1], N_Cycle init 1
end % Cycle %
```

Table 4: The `Cycle` process.

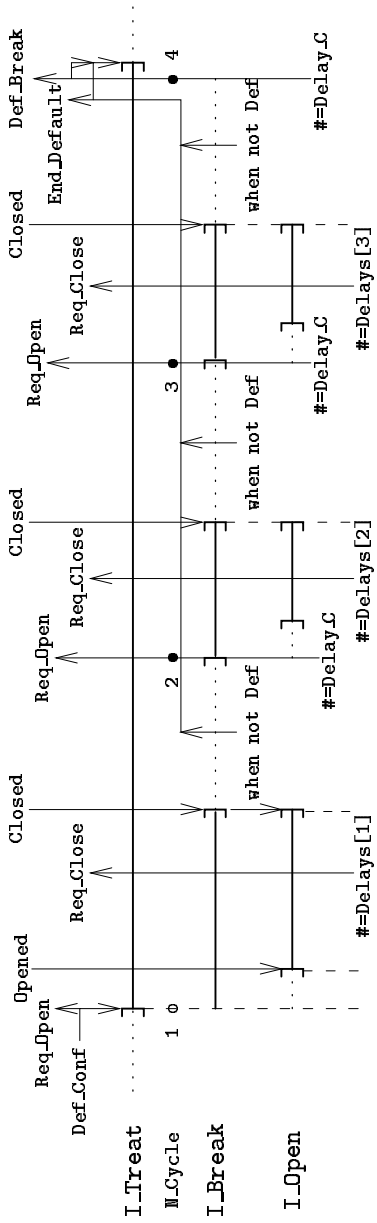

Figure 5: The treatment phase: a series of intervals.

On each entry in its activity interval, the behavior alternates between breaking the circuit in the interval `I_Break` (initially `inside`) and closing it again while testing for disappearance of the default. The cycles are counted by `N_Cycle`, initially equal to `1`, and incremented at `open I_Break`. When this counter exceeds the maximum number `N`, then `Def_Break` occurs. The current value of the Delay to be spent open in the cycle is `Current_D := Delays[N_Cycle]`. It is memorized by the `cell` and produced at the clock `Time`: this is necessary because the value of timers will be compared with `Delay` at that clock.

- Inside `I_Break`, a sub-interval `I_Open` is defined starting with the actual opening of the breaker: `Opened` (received in response to the opening request `Req_Open_Cycle`). Inside `I_Open`, a delay (of a series of values carried by the signal `Delay`) is counted on the base of `Time`. A closing request `Req_Close` is emitted when this delay is elapsed.
- Outside of `I_Break`, the breaker is closed; the event `End_Default` is emitted if the default disappears (i.e., `Def` is false) in the meantime. The duration `Delay_C` between two circuit cut-offs in the cycle is counted on `Time` outside of `I_Break`.

3.2.3 The complete behavior

Finally, the two processes presented above are assembled into a complete treatment behavior as specified in Table 5. It is parameterized with the union of the parameters of the two preceding processes. Its interface of input and output signals is also defined on part of the interfaces of the other processes.

The logical inputs corresponding to the defaults are processed in order to produce the logical `Def` and the event `First_Default` which signals raising edges of `Def`. The process `Confirmation` of Table 3 is invoked, and in composed with the process `Cycle`, which is active each time `I_Treat` is entered, and is interrupted by `Def_Break` (when the cycle has reached its end without achieving the treatment of the default) or by `End_Default` (when the default disappears while the breaker is closed). The opening of the breaker is requested by `Req_Open`, in the absence of `Def_Break`[1], when entering the treatment phase and also when a request is emitted inside the cycle.

This completes the specification of this simplified version of the behavior of the circuit breaker of a power transformer station.

3.3 Validation by graphical simulation

Simulation is useful for the validation of specifications where formal verification would be difficult e.g., for complex behaviors where the expression of properties would become too difficult, like complex schedulings, or in case of insufficient knowledge on the environment. Therefore it can be a useful complement to formal verification. Other examples of graphical

[1] For events `E1` and `E2`, the expression `when ((not E1) default E2)` defines the set subtraction of their occurrences i.e., the occurrences of `E2` in the absence of `E1`. Indeed, when `E1` is present, it has the value *true*, hence `not E1` is present and *false*, and prioritary in the `default`; therefore, through the `when` on *false*, a possible occurrence of `E2` does not come through.

```
process Treatment =(integer Delay_PH, Delay_H, Delay_W, Delay_C, N;
                    [N] integer Delays)
        {? logical PH, H, W; event Opened, Closed, Ext_Default, Time
         ! event Req_Open, Req_Close, Def_Break, End_Default}
(| % treatment on the default signals %
   (| Def := PH or H or W
    | ZDef := Def$1
    | First_Default := (when Def and not ZDef) out I_Treat
    |)
 | Confirmation(Delay_PH,Delay_H,Delay_W)
   % the whole cycle is performed each I_Treat %
 | I_Treat := ]Def_Conf, Def_Break default End_Default]
 | Cycle(Delay_C, N, Delays) each I_Treat

 | Req_Open := when ((not Def_Break) default ((open I_Treat)
                                              default Req_Open_Cycle))
 |)
where interval I_Treat init outside; logical ZDef init false
end % Treatment %
```

Table 5: The `Treatment` process.

simulation of specific SIGNAL programs have been treated, e.g. a rail-road crossing control system [8], a speech processing system [11] and a robot vision systems [13].

3.3.1 The simulation environment

In the SIGNAL programming environment, there now exists a generic built-in graphical simulation environment for SIGNAL specifications. It performs the automatic and general construction of graphical input reading and output displaying windows, for any of the interface signals of a program, in an oscilloscope-like fashion, as illustrated in Figures 6 and 7.

We want to display the presence and values of some of the intervals of the behavior, and of some events. In order to display them on oscilloscope-like window, we embed the controller into a process transforming them by encoding them as integers present at every instants. Intervals are encoded as 1 when `inside`, -1 when `outside` and 0 when absent. Events are encoded as 1 when present (which lasts only one instant and appears graphically as a peak), and 0 when absent. The input logicals are always present, and are displayed as 1 when *true* and 0 when absent.

3.3.2 Simulation of the confirmation phase

The left column of Figure 6 describes the trace of the differents inputs (i.e. the three kinds of logical defaults `PH`, `H` and `W`, and the event input `Ext_Default`). The right column shows

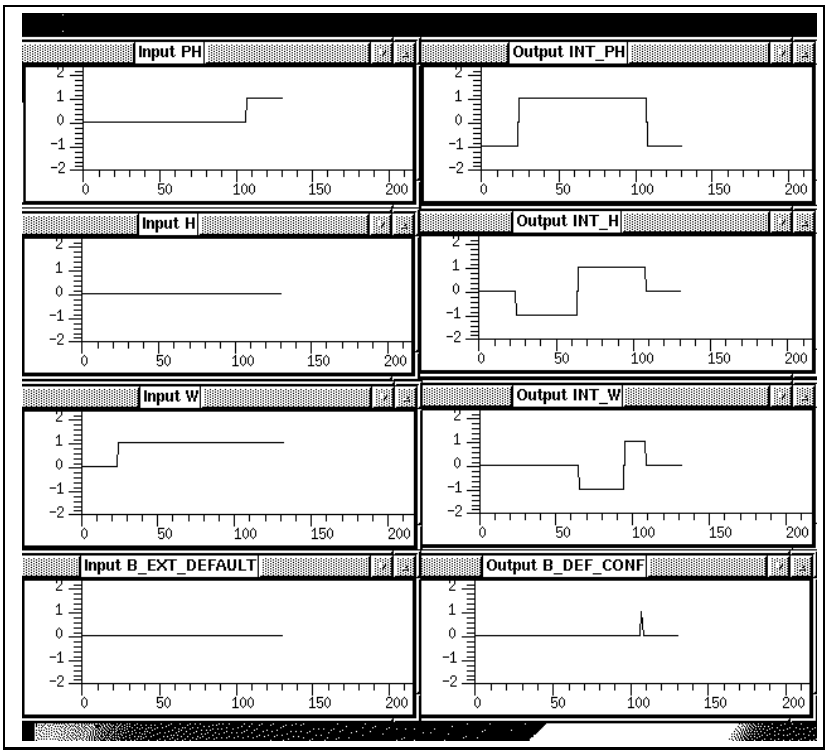

Figure 6: The simulation of the confirmation phase.

the hierarchical preemptive structure of the intervals during the confirmation phase, as well as the output event `Def_Conf`.

In the particular simulation trace illustrated in Figure 6, the first event occurs at time 20 on the horizontal scale: the logical input `W` becomes *true*. Consequently, the interval `Int_PH` (corresponding to the interval `I_PH` in Section 3.2.1) is opened, and `Int_H` is in its initial state `outside`. At the end of `Delay_PH` (for this simulation: 40) i.e. at time 60, `open Int_H` occurs, and the interval `I_W` is in its initial value `outside`. At the end of `Delay_H` (for this simulation: 30) i.e. at time 90, `open Int_W` occurs. Before the end of `Delay_W`, at time 110, the default `PH` becomes *true*: it causes interruption of the confirmation on other defaults, and emission of `B_Def_Conf`.

3.3.3 Simulation of the treatment phase

Figure 7 illustrates a trace of the behavior of the `Treatment` process. When the default is confirmed at time 20 in this scale (at the peak for `B_Def_Conf`), `open I_Treat` and `Req_Open` occur. The interval `I_Break` is entered in its initial state `inside`, and interval `I_Open` in its

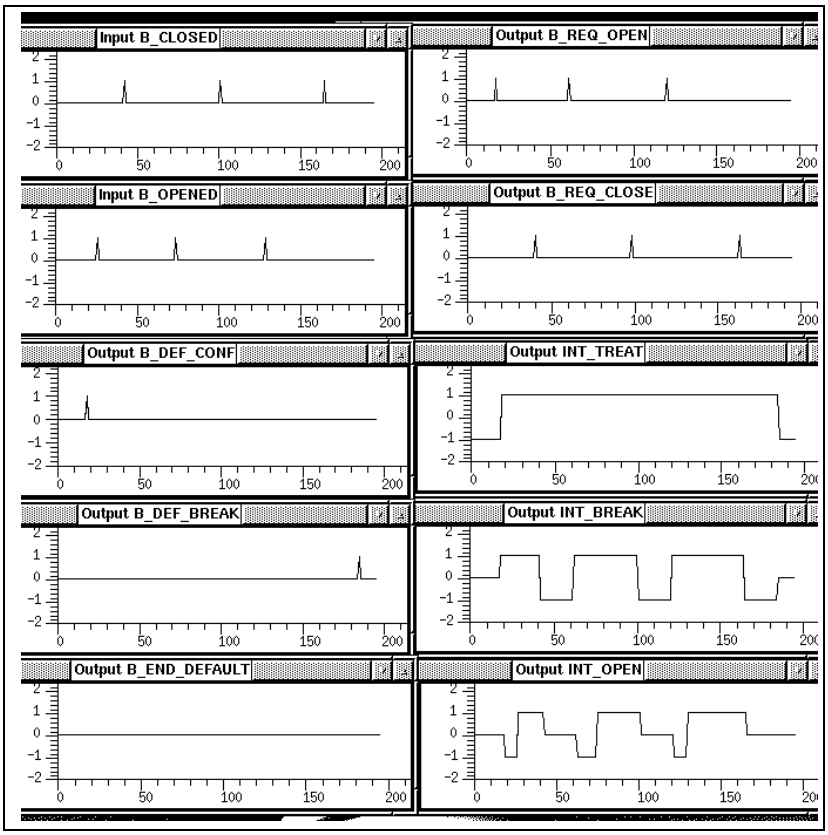

Figure 7: The simulation of the treatment phase

initial state *outside*. When `Opened` is received, at time 25, `open I_Open` occurs (and a delay is activated). At the end of this delay (`Delay[1]` is 15 in this simulation), at time 40, `Req_Close` is emitted. `Closed` is received immediately, entailing `close I_Open` and `close I_Break` (which is also `I_Open`'s presence interval). On `comp I_Break`, `Delay_C` (20 in this simulation) elapses at time 60, and causes occurrence of `Req_Open`, `open I_Break` and `I_Open` being present and initally `outside`. This event is actually the starting of a new cycle, with reception of `Opened` at 70, elapsing of `Delay[2]` (25 in this simulation) at 100, and elapsing of `Delay_C` at 120. Then comes the third cycle with reception of `Opened` at 130 elapsing of `Delay[3]` (35 in this simulation) at 165, and elapsing of `Delay_C` at 185. At this stage, the last cycle is over, and `Def_Break` occurs, entailing `close I_Treat`.

Even if the simulation is useful for a partial validation of a specification, we could not prove that the system has the good behavior, because all the schedulings could not be considered. The only way is to use formal analysis.

4 Formal verification in SIGNAL

In this Section, we briefly present the models and the methods used for the automated analysis of programs and the proof of their properties. They are used in the proof method for dynamical properties of SIGNALprograms [6]. The equational nature of the SIGNAL language leads naturally to the use of methods based on systems of polynomial dynamic equations over $\mathbb{Z}/_{3\mathbb{Z}}$ as a formal model of the behavior of the programs [12]. This aspect is an originality of the SIGNAL approach compared to others, e.g. ESTEREL, using finite state automata [5]. The systems of polynomial equations caracterize sets of solutions, which are states and events. The techniques used in the method consist in manipulating the equation systems instead of the solutions sets, which avoids the enumeration of the state space.

4.1 Transforming a SIGNAL program into a system of polynomial equations

Signals. In order to prove its dynamical properties, a SIGNAL processes is translated into a system of polynomial equations over $\mathbb{Z}/_{3\mathbb{Z}}$, i.e. integers modulo 3: {-1,0,1} [10]. The principle is to code the three possible values of a boolean signal `X` (i.e., *present* and *true*, or *present* and *false*, or *absent*) in a *signal variable* x by :

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

For the non-boolean signals, we only code the fact that the signal is *present* or *absent*:

$$\begin{cases} present & \rightarrow & \pm 1 \\ absent & \rightarrow & 0 \end{cases}$$

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 processes. 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, is different because it requires memorizing the past value of the signal into a *state variable* ξ. In order to encode `Y := X $ 1 init Y0`, we

have to introduce the three following equations:

$$\xi' = x + (1 - x^2)\xi \quad (1)$$
$$y = \xi x^2 \quad (2)$$
$$\xi_0 = y_0 \quad (3)$$

Equation (1) describes what will be the next value ξ' of the state variable. If x is *present*, ξ' is equal to x (because $(1 - x^2) = 0$), otherwise ξ' is equal to the last value of x, memorized by ξ. Equation (2) gives to y the last value of x (i.e. the value of ξ) and constrains the clocks y and x to be equal. Indeed, $y^2 = \xi^2 x^4$, and in $\mathbb{Z}/3\mathbb{Z}$ we have $x^3 = x$ (because $(-1)^3 = -1$), i.e. $x^4 = x^2$, so $y^2 = \xi^2 x^2$; $\xi^2 = 1$ because ξ is always present, hence $y^2 = x^2$. Equation (3) corresponds to the initial value of ξ, which represents the initial value of y.

The table 4.1 shows how all the primitive operators are translated into polynomial equations.

Boolean instructions	
`Y := not X`	$y = -x$
`Z := X and Y`	$z = xy(xy - x - y - 1)$ $x^2 = y^2$
`Z := X or Y`	$z = xy(1 - x - y - xy)$ $x^2 = y^2$
`Z := X default Y`	$z = x + (1 - x^2)y$
`Z := X when Y`	$z = x(-y - y^2)$
`Y := X $1 (init` y_0`)`	$\xi' = x + (1 - x^2)\xi$ $y = x^2\xi$ $\xi_0 = \mathtt{y}_0$
non-boolean instructions	
`Y := ` $f(X_1, \ldots, X_n)$	$y^2 = x_1^2 = \cdots = x_n^2$
`Z := X default Y`	$z^2 = x^2 + y^2 - x^2y^2$
`Z := X when Y`	$z^2 = x^2(-y - y^2)$
`Y := X $1 (init` y_0`)`	$y^2 = x^2$

Table 6: Translation of the primitive operators into polynomial equations.

Processes. By composing the equations representing the elementary processes, any SIGNAL specification can be translated into a set of equations called polynomial dynamic system. Using this encoding, the reaction events of the program, i.e. the value of each of the m *signal variables* and n *state variables*, are represented by a vector in $(\mathbb{Z}/3\mathbb{Z})^{n+m}$. Formally, a polynomial dynamic system can be reorganized into three sub-systems of polynomial

equations of the form:

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

where:

- X is a set of n variables, called *state variables*, represented by a vector in $(\mathbb{Z}/_{3\mathbb{Z}})^n$;
- Y is a set of m variables, called *event variables*, represented by a vector in $(\mathbb{Z}/_{3\mathbb{Z}})^m$;
- $X' = P(X,Y)$ is the *evolution equation* of the system; it can be considered as a vectorial function $[P_1, \ldots, P_l]$ from $(\mathbb{Z}/_{3\mathbb{Z}})^{n+m}$ to $(\mathbb{Z}/_{3\mathbb{Z}})^n$. It groups all the equations on the state variables, and caracterizes the dynamical aspect of the system;
- $Q(X,Y) = 0$ is the *constraints equation* of the system, it is a vectorial equation $[Q_1, \ldots, Q_{l'}]$. It groups the equations caracterizing the statical aspect of the system;
- $Q_0(X) = 0$ is the *initialization equation* of the system, it is a vectorial equation $[Q_{0_1}, \ldots, Q_{0_{l''}}]$. It groups the equations caracterizing the initialization of the system.

A polynomial dynamic system can be seen as a finite transition system. The initial states of this automaton are the solutions of the equation $Q_0(X) = 0$. When the system is in a state $x \in (\mathbb{Z}/_{3\mathbb{Z}})^n$, any event $y \in (\mathbb{Z}/_{3\mathbb{Z}})^m$ such that $Q(x,y) = 0$ can constitute a transition. In this case, the system evolves to a state x' such that $x' = P(x,y)$.

We thus have a mathematical model characterizing the behavior of dynamical systems. In the perspective of analysing these behavior by the evaluation of the satisfaction of properties, we need operations on systems of polynomials, which will correspond to the manipulation of the sets of their solutions. This way we can express ourselves about sets of behaviors, states and transitions, while still remaining in the domain of polynomial functions, and not having to enumerate them.

4.2 Operations on the polynomial dynamical systems

The theory of polynomial dynamical systems uses operations in algebraic geometry such as varieties, ideals and morphisms [12, 10].

Description of the basic objects and operations. Let us define the quotient ring of polynomial functions $A[X,Y] = \mathbb{Z}/_{3\mathbb{Z}}[X,Y]/_{(X^3-X, Y^3-Y)}$ [2]: it is the set of polynomials in $\mathbb{Z}/_{3\mathbb{Z}}$ for which the degree in each variable is ≤ 2 because of the fact that $X^3 = X$. Let E be a set of event and state variables in $(\mathbb{Z}/_{3\mathbb{Z}})^{n+m}$. The following set of polynomials:

$$I(E) = \{p \in A[X,Y] \ / \ \forall (x,y) \in E, \ p(x,y) = 0\}$$

[2] $X^3 - X$ (resp. $Y^3 - Y$) denotes all the polynomials $X_i^3 - X_i$ (resp. $Y_i^3 - Y_i$).

is called the *ideal* of E in $A[X,Y]$. This set represents all the polynomials, for which the set E is a solution. In terms of dynamical systems, it represents the set of equations characterizing the states and events in E.

Reciprocally, to any set of polynomials G, we can associate a set in $(\mathbb{Z}/_{3\mathbb{Z}})^{n+m}$, called the *variety* of G, defined as follows:

$$V(G) = \{(x,y) \in (\mathbb{Z}/_{3\mathbb{Z}})^{n+m} \ / \ \forall p \in G,\ p(x,y) = 0\}$$

This set represents all the solutions for a given set of polynomials. In terms of dynamical systems, it represents the set of states and events admissible by the dynamical systems in G.

The advantage of using ideals is that there exists a direct correspondance between an ideal and the associated variety. In fact, we can easely prove that, in the quotient ring $A[X,Y]$:

$$V(I(E)) = E \text{ and } I(V(< G >)) =< G >$$

where, for a set of polynomials G, $< G >$ is the set of all linear combinations of polynomials in G: this means that their solutions include those of G. This way, we can translate properties of sets into equivalent properties of associated ideals of polynomials. Hence, instead of manipulating explicitely and enumerating the states, this approach manipulates the polynomial functions characterizing their sets. An other important aspect is that an ideal can be represented by a single polynomial, called *the principal generator*. This particularity is used in the pratical implementation of the algorithms on ideals.

For example, for the contraint equation Q of a polynomial dynamic system: the equation $Q(X,Y) = 0$ represents a set of polynomial equations, decomposed as follows:

$$\begin{cases} Q_1(X,Y) &= 0 \\ \cdots & \\ Q_{l'}(X,Y) &= 0 \end{cases}$$

If E is the set of solutions of this system of equations, it is clear that

$$E = V(< Q_1, ..., Q_{l'} >) \text{ and } I(E) =< Q_1, ..., Q_{l'} >$$

So instead of manipulating the set of solutions of the contraint equation E, represented in our case by a variety, we can easely convert it into an ideal $I(E)$, which can be represented by a single polynomial. This way, The relations, like inclusion of set of states, between differents sets, will be computed by operations on polynomials.

Operations on dynamical behaviors. To capture the dynamical aspect of a polynomial dynamical system, we introduce the notion of morphism and comorphism. A *morphism* is a polynomial function P from $(\mathbb{Z}/_{3\mathbb{Z}})^{n+m}$ to $(\mathbb{Z}/_{3\mathbb{Z}})^{n}$ (the *evolution equation* of the system $X' = P(X,Y)$, for example).

With the morphism P is associated a *comorphism* P^* from $\mathbb{Z}/_{3\mathbb{Z}}[X]$ to $\mathbb{Z}/_{3\mathbb{Z}}[X,Y]$ defined by: