A Design Environment for Discrete-Event Controllers based on the Signal Language
Hervé Marchand, Patricia Bournai, Michel Le Borgne, Paul Le Guernic

To cite this version:

HAL Id: hal-00546362
https://hal.archives-ouvertes.fr/hal-00546362
Submitted on 14 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
A Design Environment for Discrete-Event Controllers
based on the SIGNAL Language

Hervé Marchand, Patricia Bournai, Michel Leborgne, Paul Le Guernic
IRISA / INRIA - Rennes, Campus Universitaire de Beaulieu, F-35042 Rennes, France
e-mail: {hmarchan, bournai, leborgne, leguerni}@irisa.fr

Abstract: In this paper, we present the integration of a controller synthesis methodology in the SIGNAL environment through the description of a tool dedicated to the algebraic computation of a controller and then to the simulation of the controlled system. The same language is used to specify the physical model of the system and the control objectives. The controller is then synthesized using the formal calculus tool SIGALI. The result is then automatically integrated in a new SIGNAL program in order to obtain a simulation of the result.

Keywords: Control theory, Polynomial dynamical system, Synchronous Methodology, SIGNAL

Introduction

In this paper, we present the integration controller synthesis techniques in the SIGNAL environment [1] through the description of a tool dedicated to the safe construction and the simulation of reactive system controllers. The system is specified in SIGNAL and the control synthesis is performed on a logical abstraction of this program, named polynomial dynamical system (PDS) over \( \mathbb{Z}/32 \) [7]. The control of the system is performed by restricting the controllable input values to values suitable with respect to the control objectives. This restriction is obtained by incorporating new algebraic equations in the initial system. The theory of PDS uses classical tools in algebraic geometry, such as ideals, varieties and homomorphisms. This theory sets the basis for the verification and the controller synthesis tool, SIGALI, of the SIGNAL environment. In this paper, we present a tool developed around the SIGNAL environment allowing the visualization of the synthesized controller by interactive simulation of the controlled system. In a first stage, the user specifies in SIGNAL both the physical model and the control objectives to be ensured. A second stage is performed by the SIGNAL compiler which translates the initial SIGNAL program into a PDS and the control objectives in terms of polynomial relations and operations. The controller is then synthesized using SIGALI. The result is a controller coded by a BDD (binary decision diagram) [3]. In a third stage, in order to visualize the new behavior of the controlled system, the controller and some simulation processes are automatically included in the initial SIGNAL program. It is then sufficient for the user to compile the resulting SIGNAL program which generates a simulator. Academic examples are used to illustrate the application of the tool.

1 The SIGNAL environment

To specify our model, we use the synchronous data flow language SIGNAL [1]. 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 [2], 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 interdependencies, and automatically generates optimized executable code ready to be embedded in environments for simulation, test, prototyping or the actual system.

1.1 The SIGNAL language

The SIGNAL language [1] manipulates signals, which denote unbounded series of typed values, indexed by time. An associated clock determines the set of instances 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 instances 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 [1].

1.2 SIGALI: The formal proof system

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) as a formal model of program behavior. The theory of PDS uses classical tools in algebraic geometry, such as ideals, varieties and morphisms [6]. The techniques consist in manipulating the system of equations instead of the sets of solutions, which avoids enumerating the state space.

1.2.1 Logical abstraction of a Signal program: To model its behavior, a Signal process is translated into a system of polynomial equations over \( \mathbb{Z}/3\mathbb{Z} \) [6]. The three possible states of a boolean signal \( X \) (i.e., present and true, present and false, or absent) are coded in a signal variable \( x \) by \( \text{present} \land \text{true} = 1 \), \( \text{present} \land \text{false} = 1 \) and \( \text{absent} = 0 \). For the non-boolean signals, we only code the fact that the signal is present (by 1) or absent (by 0). Each of the primitive processes of SIGNAL are then encoded in a polynomial equation (cf Table (2)).

<table>
<thead>
<tr>
<th>Language Construct</th>
<th>Signal syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stepwise extensions</td>
<td>( C \coloneqq A \ \text{op} \ B )</td>
<td>where ( \text{op} ) : arithmetic/relation/boolean operator</td>
</tr>
<tr>
<td>delay</td>
<td>( ZX := X \ &amp; \ n )</td>
<td>memorization of the ( n^{th} ) past value of ( X )</td>
</tr>
<tr>
<td>extraction</td>
<td>( C := A \ \text{when} \ B )</td>
<td>( C ) equal to ( A ) when ( B ) is present and true</td>
</tr>
<tr>
<td>priority merging</td>
<td>( C := \text{A default B} )</td>
<td>if ( A ) is present ( C := A ) else if ( B ) present ( C := B ) else ( C ) absent</td>
</tr>
<tr>
<td>Process Composition</td>
<td>( [P[N]] )</td>
<td>processes are composed, common names correspond to shared signals</td>
</tr>
<tr>
<td>useful extensions</td>
<td>when ( B ) the clock of the true instants of ( B )</td>
<td>event ( B ) the presence instants of ( B )</td>
</tr>
<tr>
<td></td>
<td>( A = B )</td>
<td>Clock of ( A ) equal with clock of ( B )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signal Processes</th>
<th>Boolean Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B := \text{not} \ A )</td>
<td>( b = \neg a )</td>
</tr>
<tr>
<td>( C := A \ \text{and} \ B )</td>
<td>( c = ab )</td>
</tr>
<tr>
<td>( C := A \ \text{or} \ B )</td>
<td>( c = a + b - a \cdot b )</td>
</tr>
<tr>
<td>( C := A \ \text{default} \ B )</td>
<td>( c = a + (1 - a^2)b )</td>
</tr>
<tr>
<td>( C := A \ \text{when} \ B )</td>
<td>( c = a - b^2 )</td>
</tr>
<tr>
<td>( B := A \ $1 \ (\text{init} \ 0) )</td>
<td>( x = a + (1 - a^2)x )</td>
</tr>
<tr>
<td>( b = a^2x )</td>
<td>( x_0 = b_0 )</td>
</tr>
</tbody>
</table>

Table 2: Translation of the primitive operators.

Any SIGNAL specification can then be translated into a set of equations called polynomial dynamical system (PDS). Formally, a PDS can be reorganized into three subsystems of polynomial equations of the form:

\[
S = \begin{cases} 
X' = P(X,Y,U) \\
Q(X,Y,U) = 0 \\
Q_0(X) = 0 
\end{cases} \tag{1}
\]

where \( X, Y, U, X' \) are vectors of variables in \( \mathbb{Z}/3\mathbb{Z} \) and \( \text{dim}(X) = \text{dim}(X') = n \). The components of the vectors \( X \) and \( X' \) represent the states of the system and are called state variables. They come from the translation of the delay operator. \( Y \) is a vector of variables in \( \mathbb{Z}/3\mathbb{Z} \), called uncontrollable event variables, whereas \( U \) is a vector of controllable event variables\(^2\). The first equation is the state transition equation; the second equation is called the constraint equation and specifies which event may occur in a given state; the last equation gives the initial states. The behavior of such a PDS is the following: at each instant \( t \), given a state \( x_t \) and an admissible \( u \), we can choose some \( u_t \) which is admissible, i.e., such that \( Q(x_t, y_t, u_t) = 0 \). In this case, the system evolves into state \( x_{t+1} = P(x_t, y_t, u_t) \).

1.2.2 Control synthesis problem: Given a PDS \( S \), defined by (1) a controller is defined by a system of two equations \( C(X,Y,U) = 0 \) and \( C_0(X) = 0 \), where the latter equation \( C_0(X) = 0 \) determines initial states satisfying the control objectives and the former describes how to choose the 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 \( S \) composed with the controller is then modeled by:

\[
S_c = \begin{cases} 
X' = P(X,Y,U) \\
Q(X,Y,U) = 0 \\
Q_0(X) = 0 \\
C_0(X) = 0 
\end{cases} \tag{2}
\]

Using algebraic methods, avoiding state space enumeration, we are able to compute controllers \( (C, C_0) \) which ensure:

- the invariance of a set of states \( S_{\text{Security}}(\)\), the reachability of a set of states from the initial states of the system \( S_{\text{Reachability}}(\)\), the attractivity of a set of states \( E \) from a set of states \( F \) \( S_{\text{Attractivity}}(\)\)) [4],
- the minimally restrictive control (choice of a command such that the system evolves, at the next instant, into a state where the maximum

\(^2\)For simplicity, we can consider that the uncontrollable event variables are emitted by the system in the direction of the controller, whereas the controllable event variables are emitted by the controller in the direction of the system.
The number of uncontrollable events is admissible \((S_{\text{Free Max}})) \[^8\].

- the stabilization of a system (choice of a command such that the system evolves, at the next instant, into a state with minimal change for the state variable values \((S_{\text{Stab}})) \[^8\].

For more details on the way others controllers are synthesized, the reader may refer to \([4, 8]\).

### 2 Integration in the SIGNAL environment

In this section we present how the controller synthesis methodology has been integrated in the SIGNAL environment. First, to simplify the use of the tool, the same language is now used to specify the physical model of the system and the control objectives (as well as the verification objectives). Moreover, some obstacles prevent the diffusion of formal methods for logical controller synthesis. The most important deals with the abstraction of the obtained controllers, coded, in our framework, by BDDs. The result is in general too complex to be satisfactorily understood. We developed a tool allowing the controller synthesis as well as the visualization of the result by interactive simulation of the controlled system. Figure (1) sums up the different stages necessary to perform such simulations. In the first stage, the user specifies the physical model and the control objectives in SIGNAL. The second stage is performed by the SIGNAL compiler which translates the initial SIGNAL program into a PDS and the control objectives in terms of polynomial relations and operations. The controller is then synthesized, using SIGALI. In the third stage, the obtained controller is included in the initial SIGNAL program in order to perform simulation.

### 2.1 First phase: Specification of the model

The physical model is first specified in the language SIGNAL. It describes the global behavior of the system. In the same stage we specify a process, that describes all the properties that must be enforced on the system. This process can also contain some property verification objectives. Using a new extension of the SIGNAL language, named SIGNAL+, it is now possible to express the properties to be checked as well as the control objectives to be synthesized, directly, in the SIGNAL program. The syntax is shown in Table (3):

```plaintext
{ | System() (the physical specified in Signal) |
  | PROP : definition of the boolean PROP in Signal |
  | Sigali(S_Invariance(True(PROP))) |
}
```

**Table 3: Basic syntax of SIGNAL+**

The keyword Sigali means that the subexpression has to be evaluated by Sigali. The function Verif_Objective (it could be invariance, reachability, attractivity) means that Sigali has to check the verification objectives according to the boolean PROP, which can be seen as a set of states in the corresponding PDS. The function Control_Objective means that Sigali has to compute a controller in order to ensure the control objective for the controlled system (it could be one of the control objectives presented in section 1.2.2).

The complete SIGNAL program is obtained by putting in parallel the two processes (see Table (4)).

```plaintext
{ | System() (Physical model in Signal) |
  | Objectives() (verif and control Objectives) |
}
```

**Table 4: The complete SIGNAL program**

### 2.2 Second phase: Verification & Controller Synthesis

In order to perform the computation of the controller with regard to the different control objectives, the SIGNAL compiler produces a file which contains the PDS resulting from the abstraction of the complete SIGNAL program and the algebraic control (as well as verification) objectives. We thus obtain a file that can be read by SIGALI.

Suppose that we must enforce, in a SIGNAL program named “system.SIG” the invariance of the set of states where the boolean PROP is true. The corresponding SIGNAL program is then given by Table (5).

```plaintext
{ (1 | System() (the physical specified in Signal) |
  | PROP : definition of the boolean PROP in Signal |
  | Sigali(S_Invariance(True(PROP))) |
}
```

**Table 5: A part of the SIGNAL program**

The corresponding Sigali file, obtained after the compilation of the global SIGNAL program, is the following (Table (6)):

```plaintext
{ (1 | System() (the physical specified in Signal) |
  | Set_States : the polynomial expression in state variables and events, which is equal to 0 when PROP is true. |
  | Sigali(Set_States) |
}
```

**Table 4: The complete SIGNAL program**

The file “system2s2” is the PDS that represents the initial system. The PROP signal becomes a polynomial expression in state variables and events, which is equal to 0 when PROP is true. The last line of the file consists in synthesizing a controller which ensures the invariance of the set of states where the polynomial Set_States takes the value 0. This file is then interpreted by SIGALI that checks the verification objective and computes the controller. The result of the controller synthesis is a polynomial which is represented by a BDD which is saved in a file, used to perform simulation.
read("system.x3m"); => loading of the PDS
Set_States : T -> (PROP)
=> Compute the states where PROP is true
S_c : S_Invariance(S,Set_States) => Synthesize the controller that ensures the invariance of Set_States

Table 6: The resulting SIGNAL file

2.3 Third phase: Result Simulation
To obtain a simulation that allows to visualize the new behavior of the controlled system, the controller (more precisely, a resolver process) is automatically integrated in the initial SIGNAL program as well as simulation processes following the architecture of Figure (2).

Figure 2: The resulting SIGNAL program

2.3.1 Integration of the resolver in a SIGNAL program & simulator building: A controller is a polynomial coded in a BDD. In most cases, several values are possible for each command, when the system evolves into a state. Therefore, an algebraic equation resolver has been developed in SIGNAL for the control part of the resolver process and in C++ for the algebraic equation resolver part. This process solves polynomial equations (i.e., controllers) according to the internal state values and the input event values. The constraint part of the controller is given by a polynomial \( C(X, Y, U) = 0 \). The resolver process provides, for given values \( x, y \), all the possible values for the command \( u \). Note that not only one but all the alternatives of commands are proposed. This process is automatically integrated in the initial SIGNAL program, following the diagram of Figure (2). The links (i.e., the connections through signals) between the process resolver and the process which specifies the system are automatically added in order to obtain the new SIGNAL program.

At the same time, the user has the option of adding in this new program some generic processes of simulation. These SIGNAL processes perform, after compilation, the automatic construction of graphical input acquisition buttons and output display windows for the signals of the interface of the programs, in an oscilloscope-like fashion\(^3\); with regard to the commands, the graphical acquisition button processes are automatically added in the SIGNAL program when the resolver is included. We finally compile the resulting SIGNAL program that generates executable code ready for simulation.

2.3.2 Simulation principle: The event values are chosen by the user under the control of the resolver through an interactive dialogue box.

When a choice is performed by the user, this choice is automatically sent to the algebraic resolver, which returns the set of possible values for the remaining commands. In fact, each time a new choice is made by the user, a new controller is computed, in the sense that one variable of the polynomial controller has been instantiated. New constraints can then appear on the commands which are not totally specified. During this exchange between the dialogue box and the resolver, some commands can be totally specified by the resolver in which case their values are then imposed. The choice of the command values can be performed step by step by the user, or using a random process for a step of simulation. In the second case, the resolver chooses the command values. The user can also ask for a random simulation during an indeterminate number of simulation steps.

3 Some examples

This section illustrates the application of our design environment to two classical examples of control synthesis problems: the cat and mouse example [9] and a flexible manufacturing cell control problem [5].

3.1 The cat and mouse example
A cat and a mouse are placed in a maze shown in Figure (3). The animals can move through doors represented by arrows in this figure. Doors \( C_1, \ldots, C_7 \) are exclusively for the cat, whereas the doors \( M_1, \ldots, M_6 \) are exclusively for the mouse. Each doorway can be traversed in only one direction, with the exception of the door \( C_7 \). A sensor associated with each door detects the passages and a control mechanism allows each door to be opened or closed, except for door \( C_7 \) which always stays opened.

Figure 3: The cat and mouse example.

Initially, the cat and the mouse are in room 2 and 4 respectively. The problem is to control the doors in order

\(^3\)We are also able to perform real graphical animation in order to simulate the behavior of the system (see section 3)
to guarantee the two following requirements:

1. The cat and the mouse never occupy the same room simultaneously.
2. It is always possible for the animals to return to their initial positions.

In order to control the system, we assume that the controllable events are door opening and closing requests.

**Specification in Signal:** The complete behavior of the system is specified in Signal. Two processes compose the system. One describes the state of the doors (open or closed) and the second describes the state of the rooms (i.e., in which room the cat and the mouse are). Table (7) represents a part of this process.

```
(1 | Mouse_Room_0 := (when Z_Mvt_Mouse_3
    default (when Z_Mvt_Mouse_6)
    default (false when Z_Mvt_Mouse_1)
    default (false when Z_Mvt_Mouse_4)
    default Z_Mouse_Room_0
   | Z_Mouse_Room_0 := Mouse_Room_0 $1)
(1 | Mouse_Room_1 := (when Z_Mvt_Mouse_2)
    default (false when Z_Mvt_Mouse_3)
    default Z_Mouse_Room_1
   | Z_Mouse_Room_1 := Mouse_Room_1 $1)
| Mouse_Room_1 := Mouse_Room_0)
```

**Table 7:** Specification of the states of the rooms

The control objectives are specified by another process. Table (8) describes this specification. We first introduce the signals cat_mouse_room_i (i=0,...,4) which are true when the cat and the mouse are both in room i. Then, the boolean error is true when one of the signals cat_mouse_room_i is true and it is false otherwise (in terms of automata, we describe the set of states where objective 1 is violated). To ensure the two objectives, we require Sigali to compute a controller which ensures (i) the invariance of the set of states where the boolean error is false (objective 1) and (ii) the reachability of the cat and mouse initial positions (objective 2).

```
(1 | Cat_Mouse_Room_0 := when(Z_Cat_Room_0 and Z_Mouse_Room_0)
    | . . . . . . . .
   | Error := Cat_Mouse_Room_0 default Cat_Mouse_Room_1
    default Cat_Mouse_Room_2 default Cat_Mouse_Room_3
    default Cat_Mouse_Room_4 default false $1)
(1 | Initial_States := Z_Cat_Room_2 and Z_Mouse_Room_4 $1)
   | Sigali(Security(False(Error)))
   | Sigali(S_Reachability(True(Initial_States)))
)
```

**Table 8:** Specification of the control objectives

**Controller synthesis and simulation of the results:** The global system (the model process, and the control objectives process) is automatically translated by the compiler in a PDS. Once the controller has been synthesized by Sigali it is integrated in the Signal environment as explained in Section 2.3.1. After the compilation of this new SIGNAL program, a graphical simulation is obtained (see Figure (4)). Figure (4(a)) represents the uncontrol-
**Specification in SIGNAL.** The global system has been decomposed into 10 subsystems, respectively coding the 5 work-stations, and the 5 AGV circuits (processes WorkStation_\text{i} and Agv_\text{j}). The movement in each subsystem is cadenced by a clock, possibly different for each subsystem. Synchronizations between the different subsystems are performed through exchanged messages, coding the state of each subsystem. To realize the control objective, we define the states of the system where two AGVs are at the same time in a common zone. For example, the signal Conflict\_area_\text{i} is a boolean which is true when the AGV_\text{i} and the AGV_\text{j} are both in the conflict zone 1. Each conflict zone can be specified in SIGNAL in this manner. The boolean Conflict\_area is true when one of the Conflict\_area_\text{i} is true, it is false otherwise. It corresponds to the forbidden states (i.e., the states where two AGVs share a conflict zone). We also add in the SIGNAL program the control objectives (Signal(S\_Security(False(Conflict\_Area)))). Once the PDS is obtained, the controller is computed and incorporated in the new SIGNAL program.

**Simulation.** Even if an animated simulation (similar to the cat and mouse simulation) has been realized, we choose to show here a simulation using the generic SIGNAL processes dedicated to the simulation.

![Simulation of the AGV's synthesis problem](image)

**Figure 6:** Simulation of the AGV’s synthesis problem

In this simulation, the position of an AGV (AGV_\text{j}) in each subsystem is encoded by an integer corresponding to the current position of the AGV in the sub-Petri net. The scopes WST_\text{i} code the positions inside the corresponding workstation and finally the scopes Zone_\text{i} are integers which are equal to 1 when two AGVs are in zone number \text{i} at the same time, and equal to 0 otherwise.

4 Conclusion

In this paper, we have presented the integration of a controller synthesis methodology in the SIGNAL environment through the description of a tool dedicated to the algebraic computation of a controller and then to the simulation of the controlled system.

The specification of the system is done in a discrete event framework using the language SIGNAL. In order to facilitate this step, the user can use a block-diagram graphic interface. This environment allows the user to have graphical and textual representations of the language structures. These representations may be used together during the building or the “reading” of the program. The formal verification of a SIGNAL program, as well as the automatic controller design are performed using a formal calculus system named SIGNAL.

Finally, in order to facilitate the use of the controller synthesis methodology, we have added in the SIGNAL language the possibility of directly expressing the control objectives (and the verification objectives) in the initial SIGNAL program. Therefore, it is not necessary for the user to know (or to understand) the mathematical framework that is necessary to perform the computation of the controller. Moreover, as the result is an equation encoded by a BDD, we have developed a simulator in the SIGNAL environment which allows the user to visualize the new behavior of the controlled system.

**References**