INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The Signal data flow methodology applied to a production cell

Tochéou Pascalin Amagbegnon, Paul Le Guernic, Hervé Marchand, Éric Rutten

N˚ 2522

March, 1995

PROGRAMME 2

Rapport de recherche

ISSN 0249-6399

The Signal data flow methodology applied to a production cell

Tochéou Pascalin Amagbegnon, Paul Le Guernic, Hervé Marchand,
Éric Rutten *

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

Rapport de recherche n° 2522 — March, 1995 — 31 pages

Abstract: This report presents a method to specify, verify and implement a controller for a robotic production cell using the Signal approach. This work has been performed as part of a case study concerning a production cell, proposed by FZI of Karlsruhe. Our contribution to this case study aims at illustrating the methodology associated with the SIGNAL synchronous data flow language for the specification and implementation of control systems, as well as the verification of statical and dynamical properties using a proof system for SIGNAL programs. We describe the full development of the example, specifying a generic controller, safe for all scheduling scenarios. The specification is structured in a modular way, using two decomposition principles: one following the architecture of the production cell, the other one separating the controller from the model of the system to be controlled. The latter point lies the originality of the approach, compared to imperative methods: the declarative language is used to specify, in the form of equations on signals, the behaviour of a system, and a controller putting constraints on it This way, one can build hierarchies of nested controlled systems: in the case of the production cell, the scheduled behaviour is a controlled instance of the safe behaviour, which is itself a controlled instance of the natural behaviour. The model of the production cell is made in terms of events and boolean data, abstracting from the numerical nature of part of the sensor data; this enables the formal analysis of the logical properties of the system. The equational nature of the SIGNAL language leads naturally to the use of methods based on systems of polynomial dynamic equations over $\mathbb{Z}/_{3\mathbb{Z}}$ for the formal proof of the satisfaction of application's requirements.

Key-words: Specification, verification, real-time systems, synchronous language, data flow, robotic production cell.

(Résumé : tsvp)

Our work on verification is supported by EDF (Électricité de France).

---

*E-mail {`Pascalin.Amagbegnon`, `Herve.Marchand`, `Paul.LeGuernic`, `Eric.Rutten`}`@irisa.fr`

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

La méthodologie flot de données associée au langage Signal appliquée à une cellule de production

Résumé : Ce rapport présente une méthode pour spécifier, vérifier et implémenter un contrôleur pour une cellule de production robotique en utilisant l'approche Signal. Ce travail a été réalisé dans le cadre d'une étude de cas concernant une cellule de production, proposée par le FZI de Karlsruhe. Notre contribution à cette étude illustre la méthodologie associée au langage flots de données Signalpour la spécification et mise en œuvre de systèmes de contrôle, ainsi que les méthodes utilisées pour la preuve de propriétés statiques et dynamiques, à l'aide d'un outil de calcul formel développé autour de Signal. Nous décrivons le développement complet de l'exemple, et la spécification d'un contrôleur générique, sûr pour tous les scénarios d'ordonnacement. La spécification est structurée de façon modulaire, en utilisant deux pprincipes de décomposition: l'un suivant l'architecture de la cellule de production, l'autre séparant clairement le modèle du système et son contrôleur. Ce dernier point est une originalité de l'approche, comparée aux approches impératives: le language déclaratif est utilisé pour spécifier, sous forme d'équations sur des signaux, les comportements possibles d'un système, et un contrôleur les contraignant. De cette façon, on peut construire des hiérarchies de systèmes contrôlés imbriqués : dans le cas de la cellule de production, le comportement ordonnancé est une instance contrôlée du comportement sût, qui est lui même une instance contrôlée du comportement naturel. Le modèle de la cellule de production est fait en termes d'événements et de booléens. S'abstraire ainsi de la nature numérique de certains capteurs permet l'analyse formelle de propriétés du système. Le caractère équationnel du langage Signal mène naturellement à l'utilisation de méthodes fondées sur les systèmes polynomiaux dynamiques d'équations sur $\mathbb{Z}/_{3\mathbb{Z}}$pour la preuve formelle de la satisfaction des propriétés nécessaires à l'application.

Mots-clé : spécification, vérification, méthodes formelles, étude de cas, cellule de production robotique.

1 Introduction

The term *"reactive systems"* has been introduced by Pnueli to refer to systems which interact continuously with their environment. The programming of reactive systems is an essential industrial activity. Factories, cars, planes and a wide variety of everyday life objects are or will be computer controlled. The formal validation of these systems requisite necessity, since they are involved in such safety critical applications as aircraft control. To cope with the growing complexity of these control systems and the need of provably correct software, formal methods have emerged. To check the effectiveness of formal methods, a realistic case study has been proposed by the `FZI` in Karlsruhe [13]. It is a robotic production cell featuring many automated components interacting with each other. Many formal methods have been applied to this case study, among which LUSTRE , ESTEREL, ARGOS, and STATECHARTS.

In this report, we propose the SIGNAL approach to this case study. Our aim is to specify, verify and implement a controller for the production cell. Our contribution to this case study aims at illustrating the methodology associated with the SIGNAL synchronous data flow language for the specification and implementation of control systems, as well as the verification of statical and dynamical properties using a proof system for SIGNAL programs. We describe the full development of the example, specifying a generic controller, safe for all scheduling scenarios. The specification is structured in a modular way, using two decomposition principles: one following the architecture of the production cell, the other one separating the controller from the model of the system to be controlled. The latter point is the originality of the approach, compared to imperative methods: the declarative language is used to specify, in the form of equations on signals, behaviours of a system, and a controller putting constraints on them. This way, one can build hierarchies of nested controlled systems: in the case of the production cell, the scheduled behaviour is a controlled instance of the safe behaviour, which is itself a controlled instance of the natural behaviour. The model of the production cell is made in terms of events and boolean data, abstracting from the numerical nature of part of the sensor data; this enables the formal analysis of the logical properties of the system. The equational nature of the SIGNAL language leads naturally to the use of methods based on systems of polynomial dynamic equations over $\mathbb{Z}/_{3\mathbb{Z}}$ for the formal proof of the satisfaction of application's requirements.

This paper is organized as follows: in Section 2, we present a brief description of the production cell. Section 3 introduces the SIGNAL language through examples. In Section 4, we describe the specification of the production cell in SIGNAL. Finally, in Section 5, after a short presentation of the theory of polynomial dynamic systems, we show the method for proving properties and give some examples, using SIGALI.

2 Description of the production cell

In this Section, we will make a brief description of the production cell. For more details, the reader is referred to [13].

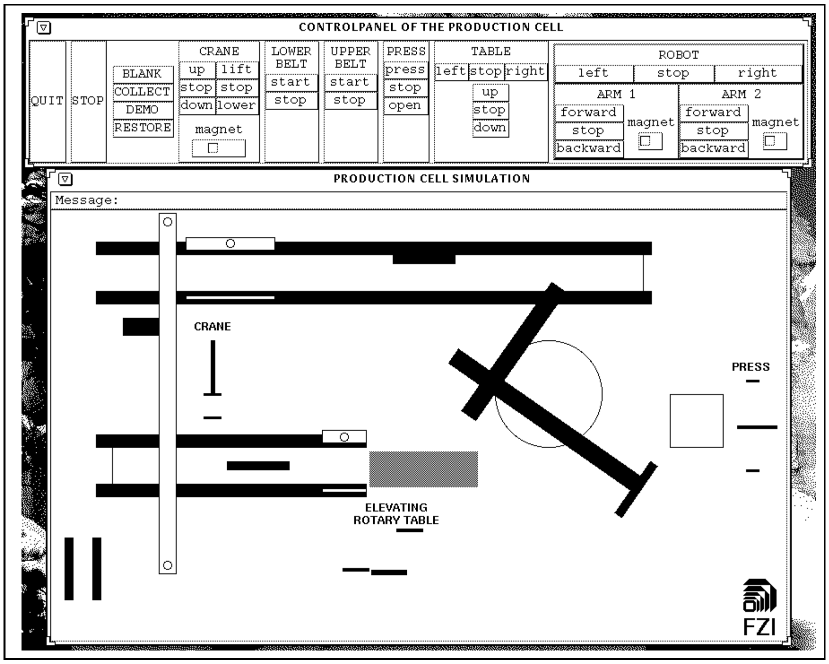


Figure 1: The Production Cell.

2.1 The components of the production cell

The production cell, illustrated in Figure 1, is composed by two belts, an elevating rotary table, a crane, a press and a two-armed robot. Briefly, the intended behaviour of the production cell is as follows: the blank is inserted in the cell by the feed belt. Then the feed belt conveys it to the elevating rotary table. This latter moves to a position, such that the robot can, with its first arm (`ARM1`), pick up the blank. The robot places the blank into the press which processes the metal blank and opens again. Finally the robot takes with its second arm (`ARM2`) the forged blank out of the press and drops it on the deposit belt . This belt conveys it to the travelling crane, which unloads the blank on the feed belt after a transversal movement. This makes the behaviour of the system cyclic.

The feed belt. The task of this belt is to transport a metal blank to the elevating rotary table. At the end of the feed belt, a photoelectric cell indicates the passing of a blank from the belt to the table.

The elevating rotary table. The task of the elevating rotary table is to transport the blank from an initial position (i.e. the position, where the blank arrives on the table) to a final position, where the robot can pick up the blank. To achieve that goal, the table must perform a vertical movement and a rotation movement. It can do both concurrently.

The press. The task of the press is to forge the blank. The press has three states:

- In the lower position, the press can be unloaded by the `ARM2` of the robot.
- In the middle position, the press can be loaded by the `ARM1` of the robot.
- In the closed position, it forges the blank.

It is important to mention, that `ARM1` must not load the press if there is already a blank in the press. So we can define the coordination between the press and the arms of the robot as follows: first, `ARM1` loads the press with a blank (the press is in middle position), then the blank is forged (the press is in closed position). finally, `ARM2` unloads the forged blank from the press (the press is in lower position).

The robot. The robot comprises two arms, which are not at the same vertical level. The two arms can retract or extend horizontally and they rotate jointly on a common foot.

- `ARM1`: Its goal is to carry a blank from the table to the press, so that the blank can be forged. For this, the table and the press must be in appropriate positions. The table must be in higher position with a 45^o horizontal orientation. The press must be in middle position.
- `ARM2`: Its goal is to take a blank from the press onto the deposit belt. This action requires the press to be in lower position.

The deposit belt. The task of the deposit belt is to convey the forged blank, which has been unloaded by the second arm of the robot, to the travelling crane. There is a photoelectric cell at the end of the belt, which indicates when a blank can be picked up by the travelling crane.

We could note that the two belt are in different levels.

The travelling crane. The task of the travelling crane is to transport the blank from the deposit belt to the feed belt. The crane has a horizontal and a vertical movement. The crane picks up the blank from the deposit belt with its electromagnet and transports it to the feed belt. Two switches indicate respectively that the crane is above the deposit belt or above the feed belt.

Interactions between the components of the cell. These different components interact with each other in order to perform the intended task. These interactions can be classified into two types: cooperation and competition.

An example of cooperation is the following interaction between the robot and the press:

- `ARM1` extends and picks up the blank with its electromagnet from the elevating rotary table.
- The robot rotates counterclockwise until `ARM2` points toward the press. `ARM2` is extented until it can reach the press, then it picks up the forged blank and retracts.
- The robot continues its counterclockwise rotation until `ARM2` points toward the deposit belt. `ARM2` extends and places the forged blank on the deposit belt.
- The robot rotates counterclockwise until `ARM1` points toward the press. `ARM1` is extented until it can reach the press, deposits the blank and retracts again.
- Then the robot reaches the original position and the cycle starts again

However, the same components also have a competition interaction in the sense that there are risks of collision between the arms and the press when it is between its lower and higher positions. For example, if the press is in middle position and `ARM2` is pointed towards it, an extension can bring the arm too close to the press. In order to avoid such collisions, the movements of the arms and the rotation of the robot have to be constrained according to the position of the press.

The controllers specified in Section 4 will specifically take care of these interactions.

2.2 Actuators and sensors

The system is controlled by means of actuators using information received from sensors. In the framework of the case study, the production cell is given in the form of a graphical simulator illustrated in Figure 1. The separate simulator and controller communicate through an interface. Messages exchanged from simulator to controller contain a vector of 14 sensor values, and the other way around, they contain commands towards the various actuators.

The commands towards the actuators in the system include: retracting and extending the two arms of the robot independently (e.g. `ARM1_FORWARD`, `ARM1_BACKWARD`, `ARM1_STOP`), rotating the base of the robot (`ROBOT_LEFT`, `ROBOT_STOP`, `ROBOT_RIGHT`), moving the crane and the table (`TABLE_STOP_H`, `TABLE_RIGHT`, `TABLE_LEFT`, `TABLE_DOWNWARD`, `CRANE_TO_BELT1`, `CRANE_STOP_H`,...).

The 14 sensors provide the control program with information concerning the state of the system; for example, the state of the press (`PRESS_CLOSED`, `PRESS_LOW`, `PRESS_MIDDLE`). There are also photoelectric cells on the two belts, which indicate when the blank is at the extreme end of the feed belt and when a blank is as the extreme end of the deposit belt.

2.3 The requirements

We have to make a distinction between safety requirements and liveness requirements. If a safety requirement is violated, this might result in damage of machines. The liveness ensures that the behavior will not dead-locked.

2.3.1 The safety requirements

Various safety requirements must be met. Most of them are related to limitations on machine mobility, the avoidance of machines collision,etc.

For example, in the case of the table and the crane:

- The safety requirements for the table are:
  - The elevating rotary table must not be rotated clockwise, if it is in the position required for transfering a blank to robot. It must not be rotated counterclockwise, if it is in the position required for receiving a blank from the feed belt.
  - The elevating rotary table must not be moved down ward if the sensor of the table indicates that the table is low. Respectively, the table must not be moved upward, if the sensor of the table indicates that the table is at the top.
- The safety requirements for the crane are:
  - If the crane is positionned above the feed belt, it may only move towards the deposit belt, and if it is positionned above the deposit belt, it may only move towards the feed belt.
  - The gripper of the crane must not be moved downard, if it is in the position required for picking up a blank from the deposit belt (respectively for upward).

The requirements for the press are given in Section 5.2.

2.3.2 The liveness requirement

The liveness property proposed by the case study can be phrased as follow:

> *Every blank introduced into the system via the feed belt will eventually be dropped by the crane on the feed belt again and will have been forged.*

In Section 5.3, we will prove some weaker forms of the liveness requirements.

3 A brief introduction to SIGNAL

3.1 A programming environment for real time systems

SIGNAL [12] is a real-time language, following the synchronous approach [10]. It is data flow oriented (i.e., declarative) and built around a minimal kernel. It manipulates signals, which are unbounded series of typed values, with an associated clock denoting the set of instants when values are present. For instance, a signal X denotes the sequence $(x_t)_{t \in \mathbb{N}}$ of data indexed by time-index t. Signals of a special kind called event are characterized only by their clock i.e., their presence (they are given the boolean value *true* at each occurrence). Given a signal `X`, its clock is obtained by the language expression `event X`, resulting in the event that is present simultaneously with `X`. Different signals can have different clocks: one signal can be absent relatively to the clock of another signal; the absence of a signal is noted $\perp$. Constructs of the language correspond to set-theoretic operations on these clocks. The constructs of the language can be used to specify the behavior of systems in an equational style: each equation states a relation between signals (i.e., between their values and between their clocks). Systems of equations on signals are built using the composition construct. In this sense, it is a form of constraint-based programming; when equations have a deterministic solution depending on inputs, the resulting program is reactive (i.e., input-driven); in other cases, correct programs can be demand-driven or control-driven, as we will see further.

The compiler performs the analysis of the consistency of the system of equations, and determines whether the synchronization constraints among the signals are verified or not. Also, it performs transformations on the graph of clocked data dependencies, including optimizations and the synthesis of an executable control. If the program is constrained so as to compute a deterministic solution, then executable code can be produced automatically (in `C` or `Fortran`). This code basically consists of a cyclic call to a transfer function, which computes an output, and updates state variables, according to the presence and value of input signals.

The complete programming environment also features a graphical block-diagram oriented user interface, a proof system for dynamic properties of programs called SIGALI (see Section 5). Work is in progress concerning hardware/software co-design and the synthesis of VHDL[3], extensions towards stochastic systems[4], interruptible data flow tasks and time intervals [18], compilation toward distributed architectures[14], and automated synthesis of controllers from the specification of their properties [9].

3.2 The kernel of SIGNAL

The kernel comprises the five following features:

- **Functions** (e.g., addition, multiplication, conjunction) are defined on the types of the language. For example, the boolean negation of a signal `E` is `not E`. The signal (Y_t), defined by an instantaneous function f : "$\forall t, Y_t = f(X_{1t}, X_{2t}, ..., X_{nt})$ is specified in SIGNAL by:

  ```
  Y := f(X1,X2, ... ,Xn)
  ```

  The signals `Y`, `X1`,..., `Xn` must all be present at the same time: they are constrained to have the same clock.

- **Delay** gives the previous value `ZX` of a signal `X` , as in: $ZX_t = X_{t-1}$, with initial value V_0. It is the only dynamical operator, and the only way to access past values. The corresponding instruction is

  ```
  ZX := X$1
  ```

with initialization at the declaration of `ZX` : `ZX init V0`

Signals `X` and `ZX` have the same clock.

- **Selection** of a signal `X` according to a boolean condition `C` is written:

  ```
  Y := X when C
  ```

  the operands and the result do not have identical clock. The clock of signal `Y` is the intersection of the clock of `X` and the clock of occurrences of `C` at the value *true*. When `Y` is present, its value is that of `X`.

- **Deterministic merge** defines the union of two signals of the same type, with a priority on the first one if both are present simulaneously:

  ```
  Z := X default Y
  ```

  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 `X` is present, or else the value of `Y` if `Y` is present and `X` is not.

- **Parallel composition** of processes is made by the associative and commutative operator " | ", denoting the union of the underlying systems of equations. Systems communicate and interact through signals defined in one system and featured in others. For these signals, composition preserves constraints from all systems, especially temporal ones. This means that they are present if the equations systems allow it.

  In Signal , for processes `P1` and `P2` , composition is written:

  ```
  (| P1 | P2 |)
  ```

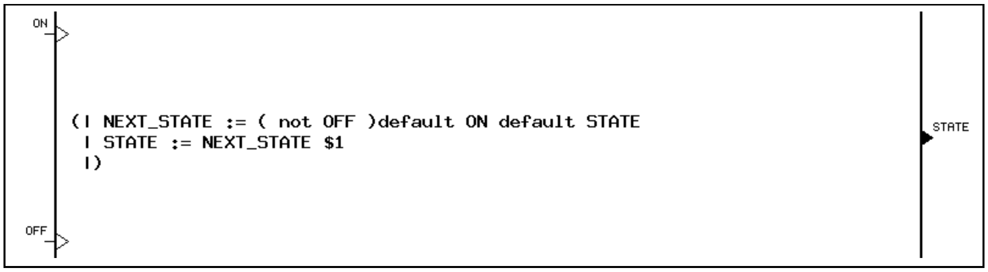
The rest of the language is built upon this kernel. A structuring mechanism is proposed in the form of process schemes, defined by a name, typed parameters, input and output signals, a body, and local declarations. Occurrences of process schemes in a program are expanded (like macro-expansion) by a pre-processor of the compiler. Derived processes have been defined from the primitive operators, providing programming comfort: e.g., `synchroX,Y` which constrains signals `X` and `Y` to be synchronous; `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*. Arrays of signals and of processes have been introduced as well.

3.3 Examples

The examples shown here are simple but meaningful: they are chosen to illustrate the originality of SIGNAL as a declarative, synchronous, equational and constraints-oriented language.

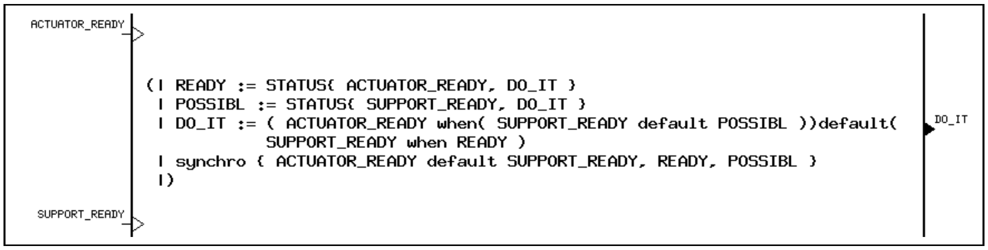
An example is the process (or system of equations) `STATUS` (illustrated in Figure 2, obtained from the graphical interface of SIGNAL [6]). It manages a boolean state variable

`STATE` (given at the output) in relation with two input events: `ON` (assigning value *true* to `STATE`), and `OFF` (putting `STATE` to the value *false*). In SIGNAL, the concept of state variable is directly available in the form of the derived operator `cell`. This example is given here just to explain the use of various primitives (delay, merge, ...) to define a slightly different behaviour. The inputs define the next value of `STATE`, which is called `NEXT_STATE`. When `OFF` is present it is *false* (i.e., the negation of `OFF`'s value, which is *true* because `OFF` is an event). When `ON` is present, the value is *true*, which is the value of event `ON`. Finally, when none of them is present, `NEXT_STATE` has its own previous value: `STATE`. `STATE` itself is the delayed value of `NEXT_STATE`. This program specifies the values of signal `STATE` in an unambiguous way. However, it does not define a deterministic presence clock: it only says that `STATE` is present at least as often as the union of `ON` and `OFF`'s clocks. The clock of the output of this process, and of its internal activity (refreshing the memory) is dependent on the constraints that might exist on `STATE` in the context in which it is used. It shows that in SIGNAL the clocks can be left under-constrained, so that the process can be re-used in more diverse contexts, with different local clocks. This feature of the language makes particularly adequate for the specification of multi-rate systems, e.g. in signal processing.



```
(| NEXT_STATE := ( not OFF )default ON default STATE
 | STATE := NEXT_STATE $1
 |)
```

Figure 2: Example of a SIGNAL process: `STATUS` manages a boolean state variable.



```
(| READY := STATUS( ACTUATOR_READY, DO_IT )
 | POSSIBL := STATUS( SUPPORT_READY, DO_IT )
 | DO_IT := ( ACTUATOR_READY when( SUPPORT_READY default POSSIBL ))default(
            SUPPORT_READY when READY )
 | synchro { ACTUATOR_READY default SUPPORT_READY, READY, POSSIBL }
 |)
```

Figure 3: Example: process `ACTION_CONTROL` makes a *rendez-vous* between two events.

Figure 3 illustrates an invocation of `STATUS` in the context of process `ACTION_CONTROL`. It performs a *rendez-vous* between two events: `ACTUATOR_READY` and `SUPPORT_READY`, each of them signalling that one device is ready for an object transmission (it is useful e.g., between feed belt and arm, or press and arm). For this, it uses two instances of `STATUS`, memorizing arrival of each of the events, and computes output event `DO_IT` when both have occurred, resetting `STATUS`es at the same time. The last line of the process (featuring the `synchro` statement) specifies that clock of the state variables `READY` and `POSSIBL` is the union of the clocks of the two input events.

4 Specification of the production cell in Signal

4.1 Overview of the specification

The specification is structured following two decomposition principles, both detailed in examples further. They have in common that they consist in composing safe sub-components with a controller managing interactions between them or with the environment safely, in order to obtain a new safe component. This ascending construction method can also be applied as a descending decomposition. This method brings flexibility in that constraints are given on behaviours progressively, localy to a composition and independently of its use in a broader context. This improves reusability because the speficiation of sub-components is not over-constrained.

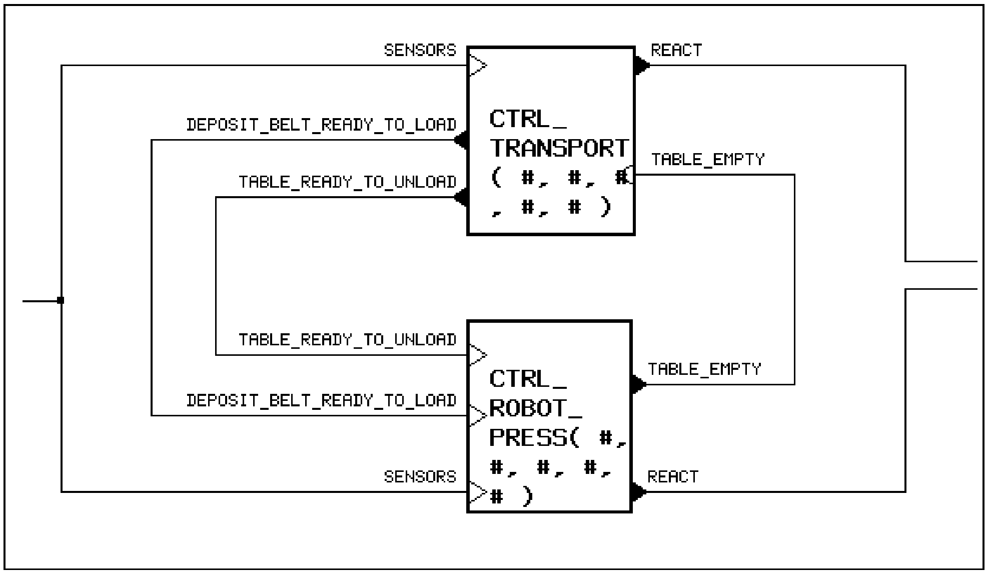

SENSORS
REACT
DEPOSIT_BELT_READY_TO_LOAD
CTRL_ TRANSPORT (#, #, # , #, #)
TABLE_EMPTY
TABLE_READY_TO_UNLOAD
TABLE_READY_TO_UNLOAD
TABLE_EMPTY
CTRL_ ROBOT_ PRESS(#, #, #, #, #)
DEPOSIT_BELT_READY_TO_LOAD
SENSORS
REACT

Figure 4: The overall structure of the production cell in SIGNAL

One principle is the decomposition according to the configuration of the application, following the architecture or resources involved: a controller is then ideally the composition of local controllers of the sub-systems, with a sub-controller coordinating them. For example, the cell can be decomposed into two groups of devices: those devoted to the transportation

(`CTRL_TRANSPORT`) and those for the transformation (`CTRL_ROBOT_PRESS`) of the metal pieces. The controller is divided into sub-controllers for each of the components of the cell. Figure 4 shows how the two parts communicate: the transportation part only needs to know when the table is empty (event `TABLE_EMPTY`) in order to synchronize with the transformation process. The other way round, the transportation process signals to the transformation process when the table is ready to be unloaded (event `TABLE_READY_TO_UNLOAD`) (i.e., when the robot can fetch an object there), and when the deposit belt can accept a new object without risking collisions (event `DEPOSIT_BELT_READY_TO_LOAD`) (i.e., when the robot can drop an object there). Section 4.2 describes this decomposition for the transportation control process.

The other decomposition principle is quite orthogonal, and concerns the separation of the controller from the process to be controlled, in a layered way. Each process is described in all its possible, "natural" behaviours (including "wrong" ones, for example those featuring collisions). This specification of the "natural" behaviour of the components of the cell is composed with a specification of the safety rules. These rules define how to avoid dangerous situations by constraining the natural behaviour, by sending commands to the machines. This defines a safe behaviour. Finally, this specification is composed with a scheduling: this determines one possible safe behaviour, thus providing the production cell with an executable, safe and meaningful behavior. Section 4.3 details the application of this method to the controller of the transformation process.

4.2 The transportation control process

The transportation control process is a good example for a decomposition following the configuration of its composing devices, because the interactions between them is simple. The local controllers of the devices are synchronized in a quite sequential way, each metal piece going along the whole trajectory. The behavior of this part of the production cell does not involve conflicts the way the other part does (see section 4.3). Therefore there is no need for a specific process to coordonate their interactions.

The transportation control process is decomposed into four components: the two belts, the crane, and the table (see Figure 5). Each process receives sensor values as input, and outputs commands to the device it controls. Communication signals exchanged between sub-controllers are highlighted in grey: they handle the synchronization of devices, each of them having to wait for another to be ready in order to start action. Concerning synchronization with the rest of the production cell, the event `TABLE_EMPTY` is received from the transformation controller (as seen in Figure 4), while on the other side, the events `TABLE_READY_TO_UNLOAD` (corresponding in Figure 4.2 to event `READY_UNLOAD`) and `DEPOSIT_BELT_READY_TO_LOAD` are sent to the transportation control process (the latter is equal to `BELT2_START`). Event `READY_LOAD` informs the feed belt controller that the table is ready to accept a blank. Event `DROP_ACCEPT` informs the crane controller that the feed belt is ready to accept a blank. On the other side, event `PICK_ACCEPT` informs the crane controller that a blank can be picked up from the deposit belt. Command `CRANE_TO_BELT1` notifies the deposit belt controller that it has been loaded. Local controllers for each of the devices behave as follows.

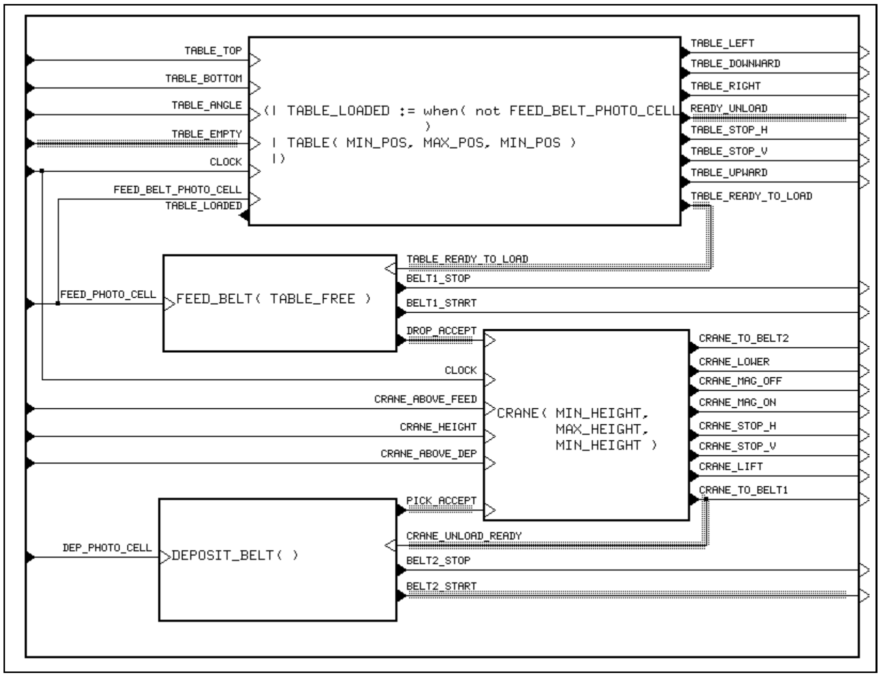


Figure 5: The transportation process, featuring feed and deposit belt, crane and table.

Controlling the table consists in making it move either from the feed belt to hte robot or the other way round. The fact that the table is loaded is acquired when the photoelectric cell changes from true to false (which is given by `TABLE_LOADED`). The table controller then starts both rotation and vertical movement simultaneously. Each movement is stopped upon reception of the corresponding sensor information. Upon arrival in high position, oriented towards the robot, it sends the event `READY_UNLOAD`. Upon reception of `TABLE_EMPTY`, the table starts back down to the feed belt, and sends `TABLE_READY_TO_LOAD` once there.

In the feed belt controller, the latter event starts the movement, and the belt is stopped if the table is loaded or if it is not in receiving position while the photo-electric sensor detects the arrival of a new blank.

The crane controller is decomposed along the same scheme as the table controller. Figure 4.2 shows that it receives the events `DROP_ACCEPT` and `PICK_ACCEPT` from the feed belt (which accepts that the crane drops a blank) and the deposit belt respectively (which is ready so that the crane can pick up a blank there). These two events contribute in triggering movements from deposit to feed belt, and conversely.

The deposit belt stops in order for the crane to deposit a blank. It restarts when the crane sends `CRANE_TO_BELT1` (i.e., `CRANE_UNLOAD_READY`), by sending `BELT2_START` which is also `DEPOSIT_BELT_READY_TO_LOAD`.

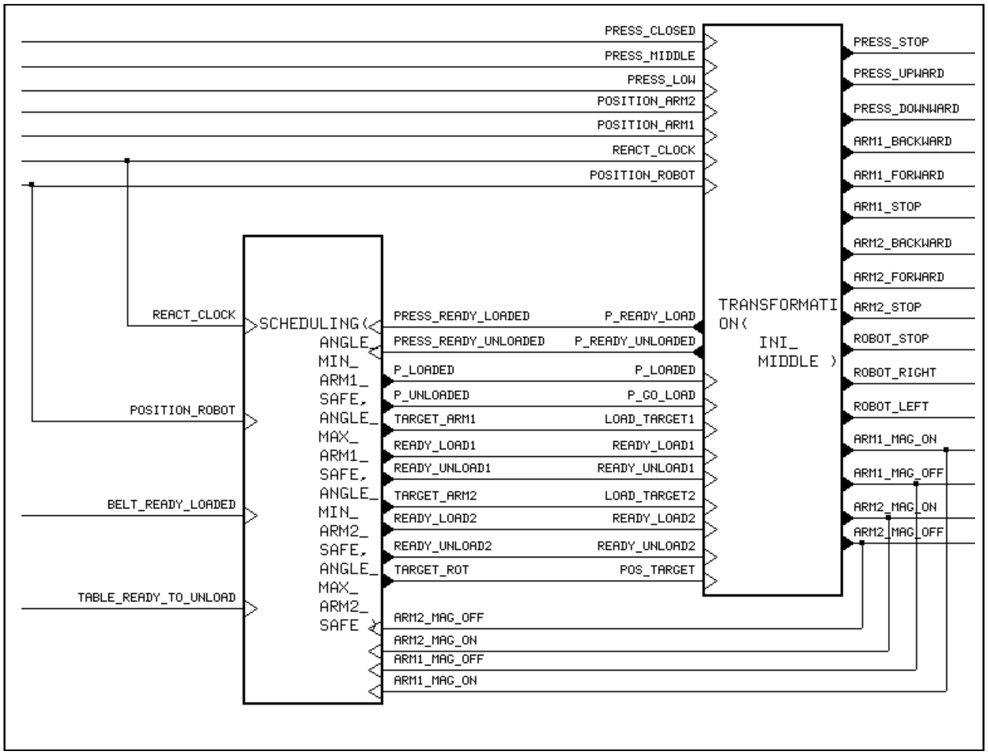


Figure 6: Scheduled behaviour: scenario and safe behaviour for the robot and the press

4.3 The transformation control process

The transformation control process (`CTRL_ROBOT_PRESS` in Figure 4) groups the controllers for the press and the robot. The reason for grouping them is the tight interaction needed to perform the collision avoidance between the two. This makes it a good example of decomposition between levels of control: here, the control of collision avoidance should be kept separate from that of the particular schedule of actions chosen. In this section we illustrate how the application can be specified hierarchically, with at each level a controller coupled with a controlled process, itself decomposed in the same way. More specifically, the scheduled behaviour is defined as the composition of a scheduling with the process defining the safe behaviours of the cell. These safe behaviors consist of a collision avoidance controller composed with the natural behaviours. Finally, these behaviors of the devices are specified in local controllers.

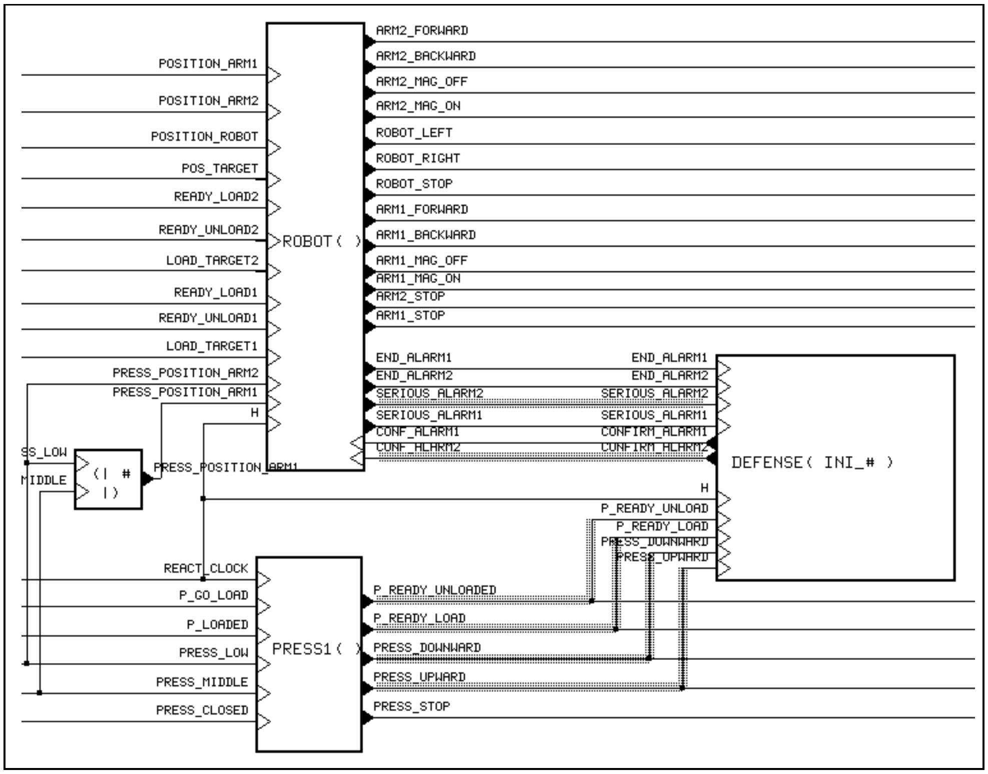


Figure 7: The transformation process: safe behaviours of the robot and the press.

4.3.1 Scheduled behavior

At the highest level, Figure 6 shows how the transformation control process is decomposed into two sub-processes: **TRANSFORMATION** specifies the safe behaviors of the sub-system, while **SCHEDULING** specifies the specific sequencing of actions to be performed according to its use in the context of the production cell.

The scheduling process receives synchronization signals from the transportation controller, as well as from the transformation process, informing it on the state of advancement in the production cell. Using this information, and state information memorizing for example whether an arm is loaded or not, it schedules goals emitted towards the robot and the arms, so as to control their movements.

4.3.2 Safe behaviors

The transformation process, illustrated in Figure 7, features the individual con trollers for the robot and for the press, interconnected by a third process **DEFENSE** wich manages all the alarm situations. The process `ROBOT` concerns the movements of the robot. It is decomposed into sub-processes which manage only the rotation and the retraction/extention of the two arms according to the goals which are sent by the process `SCHEDULING`. Two other sub-processes indicate to the process `DEFENSE` when an arm enters a dangerous region (i.e., when the robot is in a position where a collision with the press can occur).

The process `DEFENSE` specifies the synchronization constraints for collision avoidance. It receives state information from the press and signals from the robot upon entering shared regions in the physical space. According to them, it can control the robot by suspending its rotation or causing the retraction of its arms during the alarm. When the dangerous situation has been taken care of, prior movements are resumed. The process `DEFENSE` receives input signals concerning the state of the press and the position of the robot relatively to the press.

For example, the signal `SERIOUS_ALARM1` is sent by the process `ROBOT` when `ARM1` enters a dangerous region. In this case, the process `DEFENSE` checks the state of the press and sends the signal `CONFIRM_ALARM` if the press is not in safe position (for example, the press is in safe position for the `ARM1` if it is in bottom or middle position). When the process `ROBOT` receives the event `CONFIRM_ALARM`, the rotation of the robot is stopped, and it has to retract its arm. When it is done, the robot can go on with its rotation.

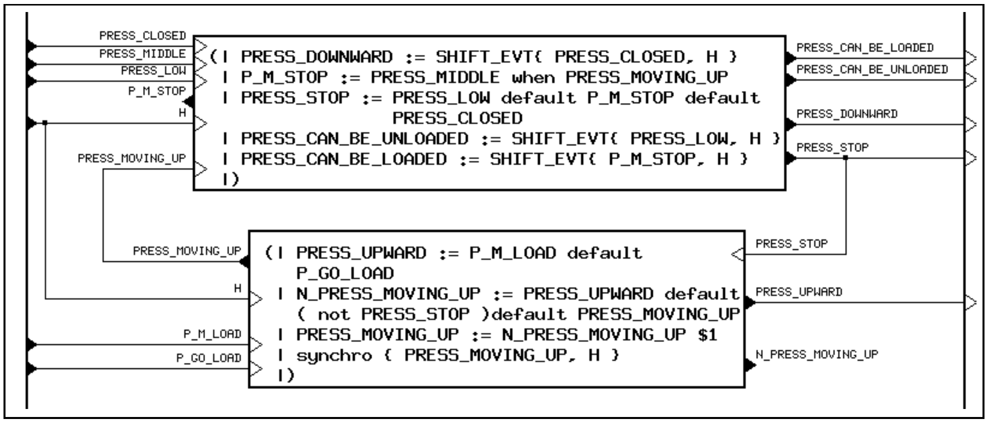
4.3.3 Natural behaviors

Finally, specifying the natural behavior of devices is illustrated here in the case of the press. This specification controls the movements of the press independently of the management of collisions and of the schedule handled at higher levels.

The process `PRESS` (see Figure 8) takes three sensor inputs: `PRESS_CLOSED`, `PRESS_MIDDLE`, `PRESS_LOW`; these are events occurring on the instants where the press arrives at the corresponding positions. Two other inputs are events received from the robot controller: `P_M_LOAD` and `P_GO_LOAD`. They respectively mean that the press is loaded with a new blank (when it has been dropped there by `ARM1`), and that the press is unloaded, and may go and load a new blank (when the previously transformed one has been unloaded by `ARM2`). The last input, `H`, is a base clock for the state variables.

In Figure8 , the lower sub-process encodes the state of the press. The boolean signal `PRESS_MOVING_UP` is true when the press moves upward, and false otherwise. It is the memorization of `N_PRESS_MOVING_UP` (the next value of the state). This latter signal is true when the command `PRESS_UPWARD` is present, false when `PRESS_STOP` is emitted, and equal to its previous value otherwise. This state variable is given the clock `H` (like in the case of the process `STATUS` in Section 3.3. The command `PRESS_UPWARD` is emitted when `P_M_LOAD` or `P_GO_LOAD` is present; i.e., it moves upward either when it has received a new blank (in order to press it) or when it has been unloaded (in order to go from low to middle position, and be ready to receive a new one).

The upper sub-process in Figure 8 computes the commands towards the press, and the synchronization signals towards the robot controller. The command `PRESS_STOP` is sent when the press arrives in lower position, or in middle position when moving upward (`P_M_STOP`, using the state variable described above), or when closing. The command `PRESS_DOWNWARD`



```
(| PRESS_DOWNWARD := SHIFT_EVT( PRESS_CLOSED, H )
 | P_M_STOP := PRESS_MIDDLE when PRESS_MOVING_UP
 | PRESS_STOP := PRESS_LOW default P_M_STOP default
                 PRESS_CLOSED
 | PRESS_CAN_BE_UNLOADED := SHIFT_EVT( PRESS_LOW, H )
 | PRESS_CAN_BE_LOADED := SHIFT_EVT( P_M_STOP, H )
 |)
```

```
(| PRESS_UPWARD := P_M_LOAD default
   P_GO_LOAD
 | N_PRESS_MOVING_UP := PRESS_UPWARD default
   ( not PRESS_STOP )default PRESS_MOVING_UP
 | PRESS_MOVING_UP := N_PRESS_MOVING_UP $1
 | synchro ( PRESS_MOVING_UP, H )
 |)
```

Figure 8: The `press` process: natural behaviours of the press.

is emitted after the press has closed, with a delay of one instant of the clock `H` (i.e., after it was stopped) using a process called `SHIFT_EVT`. The messages are computed as follows: `PRESS_CAN_BE_UNLOADED` is emitted one instant after the press has reached its lower position, and `PRESS_CAN_BE_LOADED` one instant after the press has been stopped in the middle position when moving upward.

5 Specifying and verifying the properties

5.1 Formal verification

The properties required in this case study can be proved using a proof system for dynamical properties of SIGNAL programs called SIGALI [8]. 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 behaviour of the programs. This aspect is an originality of the SIGNAL approach compared to others, e.g. ESTEREL, using finite state automata [7]. 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.

5.1.1 Transforming a SIGNAL program to 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} [11]. 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 :

$$\left\{\begin{array}{ccc} present \wedge true & \rightarrow & +1 \\ present \wedge false & \rightarrow & -1 \\ absent & \rightarrow & 0 \end{array}\right.$$

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

$$\left\{\begin{array}{ccc} present & \rightarrow & \pm 1 \\ absent & \rightarrow & 0 \end{array}\right.$$

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 representes the initial value of y.

Figure 5.1.1 shows how all the primitive operators are translated 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:

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

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`)`	$\xi' = x + (1 - x^2)\xi$ $y = x^2\xi$ $\xi_0 = \mathbf{y}_0$

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`)`	$y^2 = x^2$

Figure 9: Translation of the primitive operators into polynomial equations.

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 behaviour of dynamical systems. In the perspective of analysing these behaviour 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 behaviours, states and transitions, while still remaining in the domain of polynomial functions, and not having to enumerate them.

5.1.2 Operations on the polynomial dynamical systems

The theory of polynomial dynamical systems uses operations in algebraic geometry such as varieties, ideals and morphisms [5, 11].

Description of the basic objects and operations. Let us define the quotient ring of polynomial functions $A[X,Y] = \mathbb{Z}/_{3\mathbb{Z}}[X,Y]/_{(X^3-X, Y^3-Y)}$ [1]: 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:

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

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

$$\left\{ \begin{array}{rcl} Q_1(X,Y) & = & 0 \\ \ldots & & \\ Q_{l'}(X,Y) & = & 0 \end{array} \right.$$

---

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

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 the dynamical behaviours. 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:

$$\text{for } p \in \mathbb{Z}/_{3\mathbb{Z}}[X]\text{: } P^*(p(X)) = P^*(p(X_1, ..., X_n)) = p(P_1(X, Y), ..., P_n(X, Y))$$

where $P_1, ..., P_n$ are the components of P. In other words $P^*(p(X))$ is obtained by substituting every X_i in p with the corresponding P_i. In fact, the comorphism can be seen as a map computing the states from which we can reach the states that are solutions of the *evolution equations*; it can be used to take the transitions backwards. This is the basic tool for analysing transitions between states. But, we do not have to compute this transition map, which has a very high computing complexity. There are relations between varieties and ideals using morphisms and comorphisms that are used to perform calculations on the properties of polynomial dynamical systems in the following.

5.1.3 Properties of polynomial dynamical systems

Liveness properties. We recall that a system is *alive* if and only if no deadlock can occur. In terms of polynomial dynamical systems, the definition can be formalized as follows:

Définition 1 The liveness definition

- *A state x is alive if there exists a signal y such that $Q(x, y) = 0$.*
- *A set of states V is alive if and only if every state of V is alive.*
- *A system is alive, if and only if $\forall(x, y)$ such that $Q(x, y) = 0$, $P(x, y)$ is an alive state.*

Using this definition we can easily prove that a polynomial dynamical system is alive if and only if

$$P^*(< Q > \cap \mathbb{Z}/_{3\mathbb{Z}}[X]) \subseteq < Q >$$

Such properties can be computed using the basic operators implemented by SIGALI.

Safety properties. Informally, whereas a liveness property stipulates that some *good things* do happen (eventually), a safety property stipulates that some *bad things* do not happen during any execution of the program [1]. In our case this kind of property covers the class of properties, describing the set of good states which remains invariant. The following definition recalls the definition of an invariant set of states :

Définition 2 *A subset E of states is invariant for a dynamical system, if and only if for every state $x \in E$ and every event y admissible in the state x, the state $x' = P(x, y)$ is in E.*

This way, if we describe a property by a set of states which verify a certain property, the property is always verified if and only if the equivalent set of states is invariant for the dynamical system.

Using this definition we could easily prove that a property, represented by a set of states E is invariant considering a polynomial dynamical system if and only if

$$< P^*(I(E)) > \ \subseteq \ < Q > \ + I(E)\mathbb{Z}/_{3\mathbb{Z}}[X, Y]$$

It is also possible to compute the *largest invariant sub-set* included in a given set E of states. This property is evaluated using a fix-point computation.

Other kinds of properties may be derived from the invariance properties.

Reachability properties.

Définition 3 *A subset E of states is reachable for a dynamical system, if and only if every state $x \in E$ can be reached from the initial states of the considered dynamical system.*

To prove this property, we use the *largest invariant sub-set* of a set, briefly describe before.

This way, a set of states E is reachable from the initial states of a polynomial dynamical system if and only if the initial states are *not* included in the largest invariant sub-set of the *complement* of E (i.e., from the initial states, one is not compelled to stay in states not verifying the property).

As a conclusion, the tool SIGALI implements the basic operators: set theoretic operators, fixpoint computation, quantifiers. With these operators, we can express CTL formulae (for example) and perform symbolic model checking. The reader interested in the theoretical foundation of our approach is referred to [5], [8].

5.2 Specification of properties

In this Section, we take the press as an example and show how safety properties are specified and proved.

Consider the following property:

"The press must not be moved downward, if the press is low,
and, the press must not be moved upward if the press is closed"

It is a safety property, so in order to prove that it is true on the system, it is sufficient to prove it on the press controller. This is basically due to results in automata theory: safety properties are preserved by composition (in this case: synchronous product). The proof on the whole system is much more time consuming than the proof on a subsystem, though our tool Sigali is able to handle the whole system.

In the following, we first give a methodology to achieve that kind of proof, before applying it to the example.

5.2.1 Method of verification

The verification of a Signal process `CONTROLLER` be carried out in three steps. First, the program to be verified is composed with two other Signal process : an environment process and a property process.

- The environment process is an abstraction of the context. As a matter of fact, the Signal program might have been specified while taking into account some features of the controlled process.
- The property process models the property to be checked and computes a boolean-valued signal `Error`, which becomes *true* when the property is violated.

Then, the overall process, resulting from the composition of these three processes is translated by the Signal compiler into a polynomial dynmical system over $\mathbb{Z}/_{3\mathbb{Z}}$. Finally, using our proof system, it is easy to check that the variable `Error` is never *true*

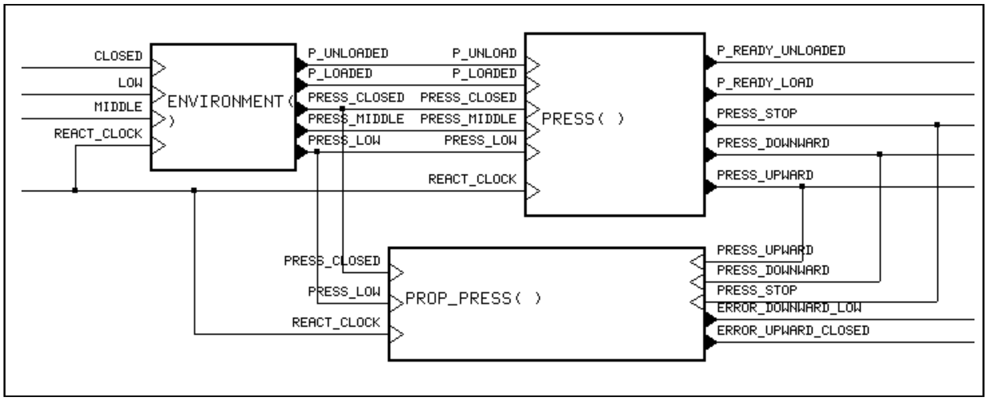


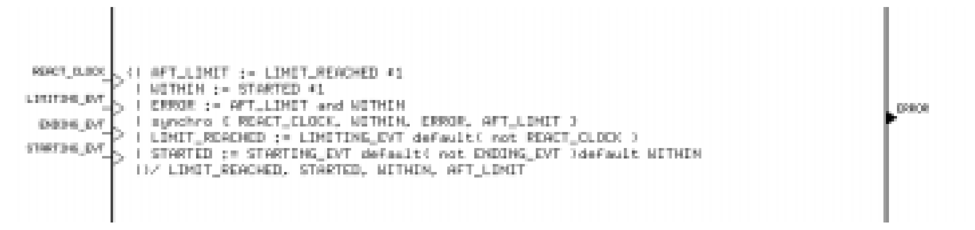
Figure 10: Verification context for the safety property of the press.

5.2.2 Example: proving the correctness of the press controller

Figure 10 shows how the verifications have been specified for the controller of the press, called `PRESS`. The process `ENVIRONMENT` models physical properties such as:

- "the physical sensors `PRESS_CLOSED` and `PRESS_LOW` and `PRESS_MIDDLE` cannot be triggered simultaneously"
- "Between the occurences of `PRESS_CLOSED` and `PRESS_LOW`, there must be an occurence of `PRESS_MIDDLE`"

`PROP_PRESS` computes the booleans `ERROR_DOWNWARD_LOW` and `ERROR_UPWARD_CLOSED`. The former is true if the press is still moving downward after the sensor `PRESS_LOW` has been triggered. The latter is true if the press is still moving upward after the sensor `PRESS_CLOSED` has been triggered



```
(| AFT_LIMIT := LIMIT_REACHED $1
 | WITHIN := STARTED $1
 | ERROR := AFT_LIMIT and WITHIN
 | synchro { REACT_CLOCK, WITHIN, ERROR, AFT_LIMIT }
 | LIMIT_REACHED := LIMITING_EVT default( not REACT_CLOCK )
 | STARTED := STARTING_EVT default( not ENDING_EVT )default WITHIN
 |)/ LIMIT_REACHED, STARTED, WITHIN, AFT_LIMIT
```

Figure 11: A generic safety property.

Let us take a closer look at the computation of the signal `ERROR_DOWNWARD_LOW`. Figure 11 describes a process named `PROP_GENERIC`; the output `ERROR` is true if the event `LIMITING_EVT` is received between an occurence of `STARTING_EVT` and an occurence of `ENDING_EVT`. As shown in Figure 12, in the case of `ERROR_DOWNWARD_LOW`, the input events `LIMITING_EVT`, `STARTING_EVT` and `ENDING_EVT` are instantiated respectively with events `PRESS_LOW`, `PRESS_DOWNWARD` and `PRESS_UPWARD` default `PRESS_STOP`.

In the framework of Sigali, the property "`ERROR_DOWNWARD_LOW` is never *true* is rephrased as "the set of states which satisfy `ERROR_DOWNWARD_LOW`=1 (for some values of the inputs) is not accessible from the initial states". So, The method consists in verifying that a state

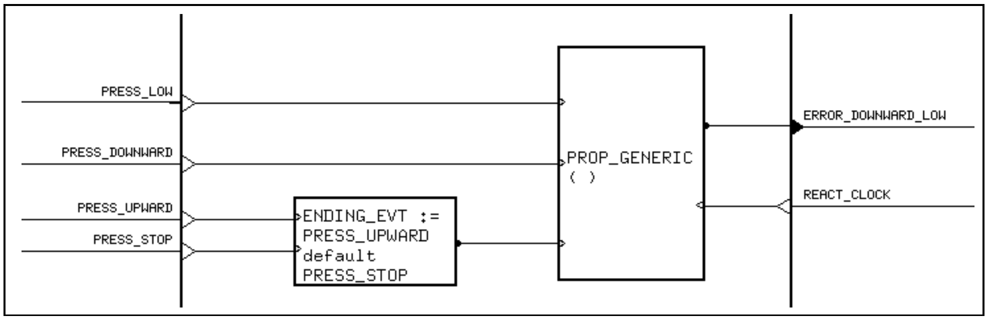


Figure 12: The process computing the safety property of the press.

where the state variable `ERROR_DOWNWARD_LOW` takes the value *true* is *not reachable* from the initial states of the dynamical system (i.e., those of the program).

The reachability of the set of states E, where `ERROR_DOWNWARD_LOW` $= 1$, can be computed by the function *reachable(prop)*, which checks that a property *prop* is never verified. In other words, it checks if the states where the properties is true can be reached from the initial states of the system. In this case, the result is *false*, in the converse, it is *true*. In our case the result is *false*.

The properties required for the table and the crane (see Section 2.3) follow exactly the same pattern. We re-used our Signal process `PROP_GENERIC` to prove them.

5.3 Proof of the liveness requirement

The liveness property given in Section 2.3.2 requires that, when a blank is introduced in the system, it will go through the various transformations and eventually come back to the feed belt. As specified in [2], this liveness property cannot be proved by our tool. As a matter of fact, the proof requires that a particular blank be tracked along the whole process. However, we are able to prove a weaker form of this liveness property: the "non-starvation" of the production cell. As a matter of fact, from the proof of properties like:

- Each time a blank is put on the table, it will eventually be in a position such that the robot can pick it up;
- Each time a blank is ready to be picked up by the robot, it will eventually be introduced in the press;
- etc ...

we can deduce that the production cell is never blocked.

Let us examine in more detail how, in the framework of our verification method, we can prove the property:

> *"each time a blank is put on the table, it will eventually be in a position such that the robot can pick it"*.

Specification of the property. To prove that this property is verified by our controller, we must make (and model) some assumptions about the table.

Consider the following scenario:

- a blank is sensed at the end of the feed belt;
- our controller issues the commands `GO_UP` and `ROTATE` to the table;
- before the table reaches the desired position, it breaks down for some mechanical reason.

The property is not verified, but it is not the fault of our controller. That is why the property must be proved with the assumption that the table is not stuck for some mechanical reasons. This can be modelled by *fairness conditions*.

Notion of fairness. Consider a transition system and a set C of state predicates on that system. A *fair execution* of the transition system with respect to C (we note C-fair) is an infinite trajectory along which each formula of C is *true* infinitely often. A temporal logic formula F is said to be *verified under the fairness assumption* C if F is verified along the C-fair execution paths.

Checking a property with respect to some fairness condition can be done with a fixpoint calculation [17] which our tool SIGALI can handle.

Application to the table. In the case of the liveness property of the table, there are two fairness conditions.

- First condition C_1: the table is in lower position and its angle is 0
- Second condition C_2: the table is in higher position and its angle is 45 degrees.

The occurrences of C_1 and C_2 with the value *true* means that the table completes its rotation and translation movement. The liveness property on the table is then rephrased as:

> *"under the assumption that C_1 and C_2 are infinitely often true, each time the a blank in put on the table, the latter will eventually be in a position such that the robot can pick the blank"*

Note that the interleaving of C_1 and C_2 is not part of the fairness assumption, but rather a property which our controller must verify.

6 Conclusion

Our contribution is evaluated in this Section according to the criteria of the case study [13].

Work load. The total time spent on the specification, implementation, proof of the production cell in SIGNAL, and redaction of the first report on this case study [2], is approximately the equivalent of four weeks for a single person. The total work load can be roughly sub-divided into the three following parts. The specification of the controller in SIGNAL took two weeks (note that the controller was designed to be general, separating the scheduling and behavior, and not only particular to its specific use in the context of the proposed production cell). The implementation took one week (installation of the simulator, programming of the input/output interface). The specification and verification of the properties took one week (analysis, formulation in SIGNAL processes and in SIGALI operations).

Programming the cell. Programming the controller in an equational style proved adequate in that the declarativeness and modularity of the language facilitates the decomposition of the specification. The relative complexity of the program is the consequence of the fact that the controller was made as general as possible for each component of the production cell, and not just specific to the given configuration.

The genericity of the robot controller is meant to make it reusable for different schedulings and scenarios with only very little change. The modifications can be restricted very locally to the process defining the scheduling. The structure of the specification follows the object decomposition of the production cell and a layered construction, with nested compositions of a system and its separated controller. The control of each device is made by a process needing only minimal synchronization communication with its environment (i.e., other devices). Hence, adding new machines is possible without changing individual controllers.

The volume of the SignalSource code is approximately 1700 lines. It must be noted that it was produced by the Signal graphical interface from which the block-diagram illustrations in this paper are taken. The duration of compilation from Signal to C is about 10 seconds with a sparc10; the compilation of the C code takes some seconds as well. The size of the produce of C is 56,7 Kbytes, which, once compiled, gives a binary file of 90 Kbytes.

Proving the correctness of the controller. The verification is based on the model underlying Signal, i.e. systems of polynomial dynamic equations over $\mathbb{Z}/_{3\mathbb{Z}}$ [5]. The equational nature of the Signal language justifies the use of an equational framework for modeling and proving properties on behaviours. This aspect is an originality of the Signal approach compared to others based on finite-state automata [7]. The systems of polynomial equations characterize a set of solutions which encode 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. Operations used on equations systems belong to the theory of algebraic geometry, such as varieties, ideals and morphisms. They enable the treatment of properties of safety and liveness. The properties proved can be treated on the relevant parts of the controller; hence their specification and the duration of the computation of proofs can be kept small. The assumptions on the environment in which they hold are explicit.

Perspectives. Other synchronous languages like Lustre and Esterel also provide the possibility of making formal proof of the satisfaction of properties of the specified systems [7, 10]. They are treated in the FZI case study [13], as well as Statecharts and many other formal methods (not all synchronous).

The Signal approach to the specification and verification of control systems has also been tried on other applications, such as the automatic circuit-breaking behaviour of power transformation station [16], and a robotic vision system [15]. The perspectives around the Signal approach concern formal methods, hybrid systems, hardware-software co-design, distributed implementation, etc ... Among them, the automatic synthesis of DEDS controllers is also studied [9].

References

[1] B. Alpern and F. B. Schneider. *Recognizing Safety and Liveness.* Technical Report 86-727, Departement of Computer Science Cornell University, Ithaca, New York, january 1986.

[2] T. Amagbegnon, P. Le Guernic, H. Marchand, and E. Rutten. The specification of a generic, verified production cell controller. In C. Lewerentz and T. Lindner, editors, *Formal Development of Reactive Systems — Case Study Production Cell*, Springer Verlag, January 1995. Lecture Notes in Computer Science LNCS n° 891.

[3] Mohammed Belhadj. VHDL & SIGNAL: a cooperative approach. In *International Conference on Simulation and Hardware Description Languages, Western Simulation Multi-Conference*, pages 76–81, Society for Computer Simulation, Tempe, Arizona (USA), 1994.

[4] Albert Benveniste, Bernard C. Levy, Éric Fabre, and Paul Le Guernic. A calculus of stochastic systems for the specification, simulation, and hidden state estimation of hybrid stochastic/nonstochastic systems. In *3rd International School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 149–169, Springer-Verlag, September 1994. Lecture Notes in Computer Science 863.

[5] M. Le Borgne, A. Benveniste, and P. Le Guernic. Dynamical systems over Galois fields and DEDS control problems. *Proc. 30th IEEE Conf. on decision and control*, 3:1505–1509, 1991.

[6] P. Bournai and P. Le Guernic. *Un environnement graphique pour le langage* SIGNAL. rapport interne 741, IRISA, July 1993.

[7] F. Boussinot and R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, 9(79):1293–1304, September 1991.

[8] B. Dutertre. *Spécification et preuve de systèmes dynamiques: Application à* SIGNAL. PhD thesis, Université de Rennes, dec 1992.

[9] B. Dutertre and M. Le Borgne. *Control of Polynomial Dynamic Systems: an Example.* rapport de recherche 2193, INRIA, January 1994.

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

[11] M. Le Borgne. *Systèmes dynamiques sur des corps finis.* PhD thesis, Université de Rennes I, September 1993.

[12] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 9(79):1321–1336, September 1991.

[13] C. Lewerentz and T. Lindner, editors. *Formal Development of Reactive Systems : Case Study Production Cell.* Springer Verlag, January 1995. Lecture Notes in Computer Science LNCS n° 891.

[14] Olivier Maffeïs and Paul Le Guernic. From SIGNAL to fine-grain parallel implementations. In *Int. Conference on Parallel Architectures and Compilation Techniques*, pages 237–246, IFIP A-50, North-Holland, August 1994.

[15] E. Marchand, E. Rutten, and F. Chaumette. *Applying the Synchronous Approach to Real Time Active Visual Reconstruction.* Technical Report 2383, INRIA, Oct. 1994. (ftp `ftp.inria.fr`, file `/INRIA/publication/RR/RR-2383.ps.Z`).

[16] H. Marchand, E. Rutten, and Samaan M. *Specifying and verifying a transformer station in* SIGNAL *and* SIGNAL*GTi.* Publication interne 916, IRISA, March 1995.

[17] K. McMillan. *Symbolic Model Cecking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, May 1992.

[18] E. Rutten and P. Le Guernic. *Sequencing Data Flow Tasks in* SIGNAL. Rapport de recherche 2120, INRIA, November 1993.

Contents

List of Figures

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

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