ELSEVIER

Science of Computer Programming 41 (2001) 85–104

Science of Computer Programming

www.elsevier.com/locate/scico

Formal verification of programs specified with signal: application to a power transformer station controller

H. Marchand[a, *,1], E. Rutten[b], M. Le Borgne[a], M. Samaan[c]

[a] *IRISA/INRIA – Rennes, F-35042 RENNES, France*
[b] *INRIA – Rhône-Alpes, F-38330 MONTBONNOT, France*
[c] *EDF, Unité Nationale Technique Système, CAP AMPERE 1, Place Pleyel, 93282 Saint-Denis cedex, France*

Received 9 February 2000; received in revised form 14 June 2000; accepted 19 July 2000

Abstract

We present a formal specification and verification of the automatic circuit-breaking behavior of an electric power transformer station, using the synchronous approach to reactive real-time systems implemented by the data-flow language SIGNAL. Synchronous languages have a mathematical model that supports the various phases of the development of a control system: specification, verification, simulation, code generation, and implementation. The complex hierarchical, state-based and preemptive behavior of the power station controller is specified in SIGNAL*GTi*, an extension of SIGNAL with notions of time intervals and preemptive tasks. To validate the specification, a graphical simulator is generated using SIGNAL's execution environment, and the required behavior is proven to be satisfied, using its proof method.

Keywords: Reactive systems; Synchronous language; Real time; Formal methods; Verification; Power systems; Case study

1. Introduction

This paper presents an experiment in the *synchronous approach* to specifying and formally verifying *reactive real-time systems* [5]. It applies the declarative language SIGNAL to the design of a complex, state-based, discrete event behavior for a power transformer station controller.

* Corresponding author.
E-mail addresses: hmarchan@irisa.fr (H. Marchand), eric.rutten@inrialpes.fr (E. Rutten), leborgne@irisa.fr (M.L. Borgne), mazen.samaan@edf.fr (M. Samaan).

[1] This research has been supported in part by EdF (Electricité de France) under contract number M 64/7C8321/E5/11.

0167-6423/01/$ - see front matter
PII: S0167-6423(00)00020-4

SIGNAL is a real-time synchronized data-flow language [19], and is related to the family of synchronous languages [13]. Its declarative style is based on equations defining the values and the synchronizations of flows of data called signals. Processes are represented by systems of equations, and compiling a SIGNAL program involves transforming the specification into an executable code that solves this system at each reaction. Compilation also verifies the causal and temporal consistency of the specification, and optimizes. The SIGNAL programming environment features a graphical editor and simulation tools, a compiler and optimizer, code generation in several target languages, and a proof tool for the analysis of dynamical systems. SIGNAL's synchronous data flow model of time is based on instants, and its actions are performed within the instants;. SIGNAL*GTi* is an extension that provides constructs for the specification of hierarchical preemptive tasks on time intervals [26].

The compiler verifies the causal and temporal consistency of the specification and proves some static invariant properties. This part of the verification is only briefly mentioned in this paper; see [3,19] for details. SIGNAL's equational formal model uses polynomial dynamical equation systems, with a proof method based on the theory of algebraic geometry. It is capable of proving a wide variety of dynamical properties, including *liveness*, *invariance*, *reachability* and *attractivity* [16,17].

In this paper we apply SIGNAL and SIGNAL*GTi* to the specification, simulation and verification of the automatic control system of a power transformer station. The controller determines the response to electric defects on the lines traversing the station, including interrupting the current, redirecting supply sources, and re-establishing current following an interruption. Its objectives are safety and uninterrupted service. It involves complex interactions between communicating automata, interruption and preemption behaviors, timers and timeouts, reactivity to external events, etc. Electrical defects are detected by sensors; the controller has to distinguish between several types of defects, and between transient and persistent ones. This selection involves a protocol, with a cycle of attempts at treating the defect in reaction to perceived events.

2. The synchronous data-flow language SIGNAL and its model

2.1. The SIGNAL equational data-flow real-time language

SIGNAL [19] is built around a minimal kernel of operators. It manipulates *signals* `X`, which denote unbounded series of typed values $(\mathtt{x}_t)_{t\in T}$, indexed by time t in a time domain T. An `event` is a signal characterized only by its presence, that always takes the value *true* (hence, its negation by `not` is always *false*). The clock of a signal `X` is obtained by applying the operator `event X`. It determines the set of instants at which values are present, called the clock of `X`. The constructs of the language can be used in an equational style to specify relationships between the values or the clocks of signals. Systems of equations on signals are built using a composition construct, thus defining *processes*. Data-flow applications are activities executed over a set of instants

in time. At each instant, input data is acquired from the execution environment; output values are produced according to the system of equations considered as a network of operations.

2.1.1. Kernel of the SIGNAL language

This is based on four operations defining primitive processes or equations, with a composition operation to build more elaborate processes in the form of systems of equations:

- *Functions* are instantaneous transformations of their inputs. Given a function f, the SIGNAL definition `Y := f{X1, X2,..., Xn}` means that $\forall t,\ Y_t = f(X_{1_t}, X_{2_t}, \ldots, X_{n_t})$. The signals `Y`, `X1`,...,`Xn` are constrained to have the same clock.
- *Selection* of a signal `X` according to a boolean condition `C` is written as follows: `Y := X when C`. If `C` is present and *true*, then `Y` has the presence and value of `X`. The clock of `Y` is the *intersection* of (*i.e.*, *included* in) that of `X` and that of `C` at the value *true*.
- *Deterministic merge* denoted: `Z := X default Y` has the value of `X` when it is present, or otherwise that of `Y` if it is present and `X` is not. Its clock is the *union* of (*i.e.*, *includes*) or *contains* those of `X` and `Y`.
- *Delay* gives access to past values of a signal, e.g., the equation $ZX_t = X_{t-1}$, with initial value V_0 defines a *dynamical process*. It is encoded by: `ZX := X$1` with initialization `ZX init V0`. `X` and `ZX` have equal clocks.
- *Composition* of processes is denoted "`|`" (for processes P_1 and P_2, with parentheses: `(| P1 | P2 |)`). It consists in the composition of the equation systems; it is associative and commutative. It can be interpreted as parallelism between processes; communication between them is carried by the broadcasting of signals.

2.1.2. Derived features and example

Several derived processes have been defined using the primitive operators, to provide programming comfort and modularity. The instruction `synchro{X,Y}` specifies that signals `X` and `Y` are synchronous (i.e., have equal clocks); this is a synchronization constraint: the compiler will take it into account when analyzing the system of constraint equations on clocks. The unary operation `when B` gives the clock of *true*-valued occurrences of logical signal `B`. `X cell B` memorizes values of `X` and also outputs them when `B` is true. The expression `C := # S` is a counter of occurrences of event `S` behaving like the example given just below. Arrays of signals and of processes have been introduced as well. Hierarchy and re-use of the definition of processes are supported by the possibility of defining process models that can be invoked by instantiation.

An example of a SIGNAL process is given in Table 1 (counter `COUNT`) which is the expanded form of the derived operation `C := # S`:

There is one input signal `S`, and an output `C`. The value of the counter `C` is defined as the previous value `ZC` incremented by one. `ZC` is declared locally, and defined using

Table 1
Example of a SIGNAL process

```
process COUNT= {?S !C}
  (| C := ZC+1
   | ZC := C$1
   | synchro{C,S}
   |)
  where ZC init V0
end
```

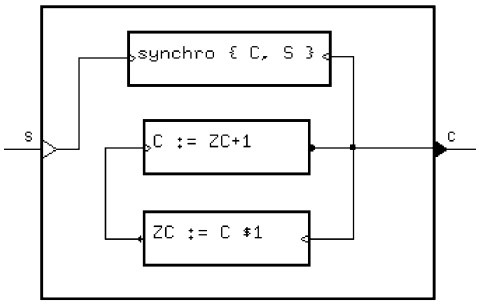

Fig. 1. A counter in SIGNAL.

the delay operator on signal `C` with initial value `V0`. `C` is synchronized with the input event `S`, and hence counts its occurrences.

2.1.3. Programming environment

The SIGNAL compiler analyzes the consistency of the equation system and determines whether the synchronization constraints between the clocks of signals are obeyed. It is based on an internal representation featuring a graph of data dependencies between operations, augmented with temporal information from the clock calculus. If the program is constrained so as to compute a deterministic solution, then executable code (in C or FORTRAN) can be produced automatically. The complete programming environment also contains a graphical, block-diagram oriented user interface where processes are boxes linked by wires representing signals, see Fig. 1.

2.1.4. Time intervals and preemptive tasks

An extension to SIGNAL, SIGNAL*GTi*, handles tasks executing on time intervals and their sequencing and preemption [26]. The motivation is to provide ways of representing behaviors that switches between *different modes of continuous interaction* with their environment. These modes are identified by time intervals delimited by discrete start and end events, within which tasks are executed. The application domain is the control of physical processes, e.g. signal processing or robotics, featuring both computations on flows of sensor data, and discrete transitions in a control automaton.

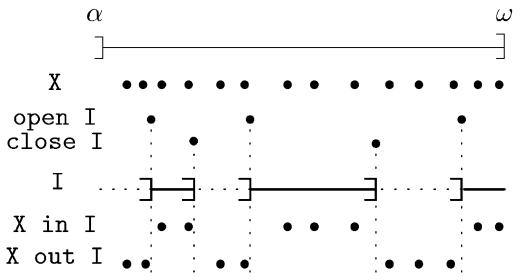

Fig. 2. Time intervals sub-dividing $]\alpha,\omega]$.

In SIGNAL*GTi*, data flow and sequencing aspects are encompassed in the same language framework, and rely on the same model for their execution and analysis (compilation and verification). In this approach, a data-flow application is considered to be executed starting from an initial state of its memory at an instant α before the first event of the reactive execution. A data-flow process has no termination specified in itself: therefore its end at instant ω can only be decided in reaction to external events or the reaching of given values. Hence, ω is part of the execution, and the time interval on which the application executes is the left-open, right-closed interval $]\alpha,\omega]$.

Time intervals are introduced in order to allow the structured decomposition of $]\alpha,\omega]$ into left-open, right-closed intervals as illustrated in Fig. 2, and their association with processes [26]. An interval `I` is delimited by occurrences of bounding events at the beginning `B` and at the end `E`. It has the value `inside` between the next occurrence of `B` and the next occurrence of `E`, and `outside` otherwise. It has an initial value `I0` (`inside` or `outside`). This is written: `I := ]B, E] init I0`. Like $]\alpha,\omega]$, sub-intervals are left-open and right-closed. This choice is coherent with the behavior expected from reactive automata or sequential circuits: a transition is made according to an input event occurrence and a current state, which results in a new state. Hence, the instant where the event occurs belongs to the time interval. The operator `compl I` defines the complement of an interval `I`, which is `inside` when `I` is `outside` and reciprocally. Operators `open I` and `close I`, respectively, give the opening and closing occurrences of the bounding events. Occurrences of a signal `X` inside interval `I` can be selected by `X in I`, and reciprocally outside by `X out I`. In this framework, `open I` is `B out I`, and `close I` is `E in I`.

With this extension, we can define the notion of *task* on an interval, which is a SIGNAL process active when the interval is `inside`, and inactive `outside`. A *suspensive task* is written `P on I`: it re-starts at its current state when re-entering `I` (see Fig. 3(a)).

An *interruptible task* is written `P each I`: it re-starts at its *initial state* (as defined by the declarations of its state variables) (see Fig. 3(b)). Processes can themselves be decomposed into sub-tasks: this way, the specification of *hierarchies* of preemptive behaviors is possible.

This extension is implemented as a pre-processor to the SIGNAL compiler [28], and is fully compatible with the environment, including the verification tools. In particular,

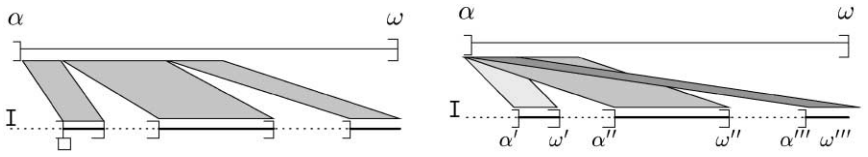

Fig. 3. Tasks associating a time interval with a process: (a) task on interval I, (b) task each interval I.

the intervals are coded by a boolean state variable, `true` when the interval is `inside` and `false` when `outside`. Occurrences of a signal `X` inside an interval `I` are coded by `X when I`. This kind of specification, using tasks and intervals, is useful for specifying properties such as "two process are not active at the same time". An example is given in Section 4.3.

In brief, our approach features an integration of preemptive and data-flow programming constructs in the language, and its direct connection to a complete programming environment, with simulation and verification. Comparable multi-paradigm approaches have also been explored in relation with combinations of ARGOS and LUSTRE [14], and recently with Mode Automata [20]. We try to remain closer the declarative style of SIGNAL. The constructs in *GTi* define quite a simple extension to SIGNAL, which could be transported immediately to the LUSTRE framework. The preemption structures in these approaches are not as rich as in ESTEREL [6]. It is possible to combine ESTEREL with a data-flow processes going through separate compilation and exchange formats [11]. The advantage is to have ESTEREL's richness, but it is a complex and low-level technique. Separation of concerns is indeed fundamental in designing language constructs. However, from a programmer's point of view, sequencing and data flow do occur in the same applications. Our approach is to make convenient programming constructs available, while retaining the underlying data-flow or equational model.

3. Verification tools for SIGNAL programs

The verification of a SIGNAL program can concern invariant properties (to be satisfied at all instants of its execution) or dynamical ones (to be satisfied on the histories of the program). Invariant properties are addressed by the compiler, which checks the consistency of constraints between the clocks and proves static properties. Several phases occur during the compilation of a SIGNAL program. One of these resolves a system of boolean equations, that encodes the constraints among the different clocks. This clock calculus relies on an algebra on sets of instants detailed in [3]. By composing the specification with the SIGNAL expression of static (i.e., *temporally invariant*) properties, the compiler checks whether they are mutually consistent. If so, their composition constitutes a correct controller that satisfies the property. An example is given in Section 4.3.

Dynamical properties are proved by a formal method based on a model of the behavior of the program. The SIGNAL environment contains a dynamical verification

and controller synthesis tool-box, SIGALI. The equational nature of the SIGNAL language leads naturally to the use of a method based on polynomial dynamical equation systems over $\mathbb{Z}/3\mathbb{Z} = \{-1, 0, 1\}$ (integers modulo 3) as a formal model of program behavior. The model deals essentially with boolean and synchronization properties, i.e. control. Polynomial equation systems characterize sets of solutions, which represent states and events. The method manipulates equation systems rather than solution sets, thus avoiding enumeration of the state space. More precisely, a set of states and/or events can actually be represented by a unique polynomial called the *principal generator*. Operations on sets are performed within the domain of polynomial functions. The tool SIGALI implements the basic set theoretic operators, fix-point computation and quantifiers [21]. It relies on an implementation of polynomials by ternary decision diagrams (TDD) (for three valued logics). These are in the same spirit as BDDs [9], but the paths in the data structures are labeled by values in $\{-1, 0, 1\}$ instead of $\{0, 1\}$.

3.1. An equational model of the behavior of SIGNAL programs

To model its behavior, a SIGNAL process is translated into a system of polynomial equations over $\mathbb{Z}/3\mathbb{Z}$ [15]. 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 (*present* and *true* $\rightarrow 1$, *present* and *false* $\rightarrow -1$, and *absent* $\rightarrow 0$). For the non-boolean signals, we only code the fact that the signal is *present* or *absent*: (*present* $\rightarrow 1$ and *absent* $\rightarrow 0$).

Each of the primitive processes of SIGNAL can be encoded in a polynomial equation. This encoding is natural in the sense that SIGNAL involves the equational specification of constraints on the relative presence and synchronizations of signals. The encoding itself [16,17] may not be particularly intuitive, but it is not meant to be visible to users. The essential point is that it leads to equations on variables which represent the presence of signals, and their values for Boolean ones. Delays must be treated specially as there is a distinction between current values (e.g., x, which was acquired in a previous instant) and next values (x', which is computed in terms of values of variables at the present instant). There are thus equations defining (i.e., having as solutions) the set of initial states, the set of admissible signals (i.e., respecting the constraints on clocks), and the next values of delayed signals.

Any SIGNAL specification can be translated into a set of equations called a polynomial dynamical system (PDS), which can be organized as follows:

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

where X is a vector of n variables in $\mathbb{Z}/3\mathbb{Z}$, called *state variables*, Y is a vector of m variables in $\mathbb{Z}/3\mathbb{Z}$, called *event variables*. The first equation is the *state transition equation*; the second equation is called the *constraint equation* and specifies which events may occur in a given state; the last equation gives the initial states. Such a

PDS behaves as follows: at each instant t, given a state x_t and an admissible y_t such that $Q(x_t, y_t) = 0$, the system evolves into state $x_{t+1} = P(x_t, y_t)$.

Thus, we have a mathematical model characterizing the behavior of dynamical systems in terms of polynomial systems. Note that for a boolean relation/function we have an exact coding of the relation/function as a polynomial function, while for a numerical function/relation, the encoding retains only the synchronization constraints between the signals involved in this relation/function. Therefore, SIGALI has reasoning capabilities only on the synchronization and logic properties of SIGNAL programs.

3.2. *Verifying and controlling* SIGNAL *programs*

Verification of a SIGNAL program (in fact, the corresponding PDS) can be carried out using algebraic operations. It is possible to check properties such as *invariance*, *reachability* and *attractivity* [21]. Here we just give here the basic definitions of each of the properties that will be used in this paper.

Liveness: If saying that a system is *alive* means that it can always make a move, i.e. if deadlock cannot occur, then this property states that no trajectory of the system ends in a sink state. In terms of polynomial dynamical systems, this definition can be formalized as follows:

Definition 1. A *state* x is alive if there exists a signal y such that $Q(x, y) = 0$ (i.e. a transition can be taken); 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 (i.e., from live states, only live states can be reached).

Safety: In terms of our formalism, it corresponds to the set of states which remains invariant for that property. If we characterize a property by the set of states which have it, the property is guaranteed to remain true if and only if the set of states is invariant for the dynamical system. The formal definition is as follows:

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

If a property characterizing a set of states E is not invariant, we can compute the largest invariant subset included in E. This subset is evaluated using a fix-point computation.

Another safety property is control-invariance.

Definition 3. A set E of states is *control-invariant* for a dynamical system if and only if for every state $x \in E$, *there exists an event* y admissible in the state x, such that the state $x' = P(x, y)$ is in E.

It is also possible to compute the largest control invariant subset of a given set E of states.

Other kinds of properties may be derived from liveness, invariance and control invariance.

Definition 4. A subset F of states is *reachable* if and only if every state $x \in F$ can be reached from the initial states E_0 of the dynamical system i.e., if and only if there exists *a trajectory* initialized in E_0 that reaches x.

To prove this property, we use the largest invariant subset of a set, as described before. Thus, a set of states F 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 subset of the *complement* of F.

Definition 5. A set of states F is *attractive* for a set of states E if and only if *every trajectory* initialized in E reaches F.

Using the definition above, we can prove that F is attractive for E if the set E is not included in the greatest control-invariant of the *complement* of F.

For a more complete review of the theoretical foundation of this approach, the reader may refer to [18,21]. Let us now see how we can apply this methodology to the power transformer station controller verification.

4. Application to a power transformer station

4.1. Specification of the power transformer station

4.1.1. The transformer stations on the power network

The French national power network operated by *Électricité de France* (EDF) contains a large number of transformer stations. For each high-voltage line, a transformer lowers the voltage, so that it can be distributed to end-users in urban centers [22]. In the course of operation, several kinds of electrical defects can occur, due to causes internal or external to the station. Three types of electrical defects are considered: phase (`PH`), homopolar (`H`), or wattmetric (`W`). In order to protect the device and the environment, several circuit breakers are placed in different parts of the station. These circuit breakers are alerted by sensors at different locations, and controlled by local control systems called *cells* (arrival cell, link cells, and departure cells) and by an operator in a remote control center. Each circuit breaker controller defines a behavior beginning with the confirmation and identification of the type of the defect. If the defect is confirmed, the circuit breaker is opened for a given period, then closed again. If the defect is still present after another delay, these operations are repeated for a certain number of cycles. The purpose of this is to treat transient defects. If the defect is still present at the end of the cycle, the circuit breaker is opened definitively, and control is given to the remote operator.

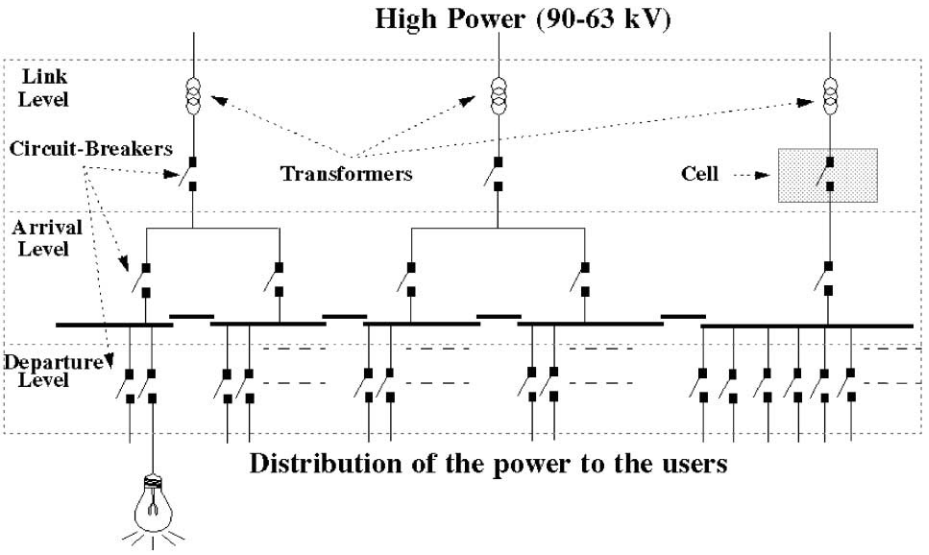

Fig. 4. Topology of a power transformer station.

One of the problems is to know which of the circuit breakers must be opened. If the defect appears on the departure line, it is possible to open the circuit breaker at departure level, or at link level, or at arrival level. Obviously, it is preferable for the circuit to be broken at the departure level, so that as few as possible users are deprived of power. This requires coordination between the different circuit breaker cells.

4.1.2. Functional description of a departure cell

We will focus on the departure cell, because it illustrates all of the interesting aspects of the controller behavior, even in this simplified presentation. The other cells have a behavior which is a subset of this one. The behavior is decomposed into a confirmation phase, which sequentially tests for the different types of defect, followed by a treatment phase, which tries to remove the defect. These behaviors feature sub-tasks which are interrupted in a nested way, and repeated in a series of activity intervals. Their specification makes use of the corresponding constructs of SIGNAL*GTi*. Here we describe only the details needed to understand the verification presented below.

The *confirmation phase* detects the occurrence of `First_Defect` and from then on, for each defect type (`PH`, `H`, or `W`), it waits to let transient defects finish naturally, and then checks for their continued presence. The defect types are tested in sequence in nested intervals. From `First_Defect`, interval `I_PH` is entered, in which the confirmation task first waits `Delay_PH`, and then enters interval `I_H` in which a task first waits `Delay_H`, and then enters interval `I_W` in which a task waits `Delay_W`. In the meantime if a defect is confirmed (i.e., `PH`, `H` or $\bar{W}$ is present at the end of the corresponding delay), the sequence is interrupted (interval `I_PH` is ended), and the defect is confirmed by emission of the boolean `Def_Conf` with value `true`. `I_PH` is also exited

if the defect disappears and the last delay elapses without defect (with `Def_Conf` at value `false`), or if another external defect occurs with emission of event `Ext_Def`.

Three properties can be verified . Firstly, if a defect is detected after the end of its corresponding delay, then the defect is confirmed by emission of `Def_Conf`. Secondly, confirmation never overlaps with treatment (i.e., the controller cannot be in states where both intervals `I_PH` and `I_Treat` are `inside`). Thirdly, if a defect appears then the defect will either be confirmed, or disappear, or an external defect will occur.

The *treatment phase* `I_Treat` begins when the defect is confirmed with the occurrence of `Def_Conf`. The task alternately breaks the circuit for varying delays, and closes it again to check whether the defect has disappeared. This continues for a certain number of cycles. Circuit breaking begins with emission of the command to open the circuit breaker, followed by the reception of an `Open` event, upon which the current delay is started. Upon completion of the delay, the circuit breaker is told to close, and this is confirmed by the reception of `Closed`. Once the circuit is re-established, if the defect has disappeared, the cell goes into its normal state. Otherwise, the treatment phase goes into the next cycle after a 0.5 s delay, or if this was the last cycle, the circuit breaker is definitively broken, a `Def_Break` signal is emitted, and the management is left to a remote human operator. The series of delay values (in the first cycle: 0.3 s, in the second: 15 s, in the third: 30 s) is treated as a signal, and the cycle is repeated as a series of activation intervals. A property to be verified is that if a defect is confirmed, either it disappears and the circuit breaker is closed, or it does not and `Def_Break` is emitted.

4.1.3. *Design in* SIGNAL *and* SIGNAL*GTi*

The confirmation phase: an interruption hierarchy: Fig. 5 illustrates the `Confirmation` process specified in SIGNAL*GTi*. The three constant parameters `Delay_PH`, `Delay_H`, and `Delay_W` correspond to each of the three kinds of electrical defects. The input event `Time` is the base clock, i.e., it is the clock of the logical inputs `PH`, `H`, and `W` (presence of the defects) and contains the clocks of the two other input events

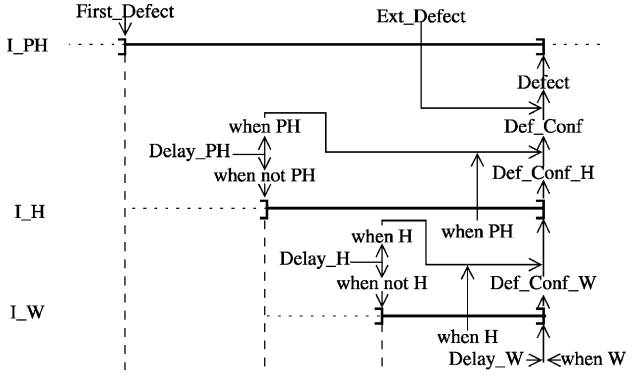

Fig. 5. Confirmation phase: interruption hierarchy.

Ext_Defect and First_Defect. The process emits the output event Def_Conf when the defect is confirmed and the output logical signal Defect which gives the state of the cell. The logical Defect is *true* when an external defect is detected (reception of Ext_Defect) or when the defect is confirmed (Def_Conf), otherwise it is false when a defect is not present (i.e., when the disjunction of the three types is false). The interval I_PH is entered when a defect is detected (First_Defect). It is closed until the next defect by the occurrence of Defect at the value *true*, causing the interruption of the confirmation task executed each I_PH (and also of its sub-tasks).

The interruption hierarchy, illustrated in Fig. 5, is as follows. Each time I_PH is entered, a counter of Time is fired during Delay_PH. At the end of this delay:

- If the logical PH is true, or if PH becomes *true* during the sub-interval I_H or if the defect is confirmed at a lower level (Def_Conf_H), then the defect is confirmed at this level (Def_Conf). This closes I_PH, thereby terminating the confirmation task.
- If PH is false, I_H (a sub-interval of I_PH) is entered with a sub-task that behaves in a similar way: a counter of Time is fired, and at Delay_H:
  - if H is true, or becomes so during interval I_W, or if the defect is confirmed at a lower level (Def_Conf_W), the defect is confirmed at this level (Def_Conf_H) causing the interval to close.
  - If H is false, I_W (a sub-interval of I_H) is entered, and a last sub-task counts Delay_W and tests for W.

This structural decomposition is reflected in the actual specification in SIGNAL*GTi* shown in Table 2, where the different levels of tasks and sub-tasks are underlined by boxes.

The treatment phase: *a series of intervals*: The treatment phase is another task. We will only sketch it here. It is forced to follow the confirmation phase simply by specifying its interval begins on the occurrence of a defect confirmation Def_Conf, which also exits the confirmation task (schematically: Treatment each I_Treat where I_Treat =]Def_Conf,End_Treat]).

The main feature of the treatment phase is its cyclical aspect. The same circuit-breaking procedure is applied, starting with Req_Open the request to open, and ending with the reception of Closed. The only thing that changes is the value of the delay fired on reception of Open. This suggests an implementation based on a signal carrying the series of delay values at a clock synchronous with Open, with sub-tasks on a series of intervals as in: One_Cycle each]Req_Open,Closed].

The complete behavior: Finally, the two processes presented above are assembled into a complete treatment behavior. The logical inputs corresponding to the defects are processed in order to produce the logical Def and the event First_Defect which signals rising edges of Def. The process Confirmation is invoked, and is composed with the process Treatment, which is active each time I_Treat is entered, and is interrupted by Def_Break (when the cycle has reached its end without achieving the defect treatment) or by End_Defect (when the defect disappears while the breaker is closed). The opening of the breaker is requested by Req_Open, in the absence of

Table 2
The Confirmation process in SIGNAL*GTi*, with code formatted to underline the hierarchical structure

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

Def_Break, when entering the treatment phase and when a request is emitted inside the cycle.

4.2. Validation by graphical simulation

Simulation is useful for the validation of specifications for which formal verification would be difficult, e.g., for insufficient knowledge of the environment, or for complex

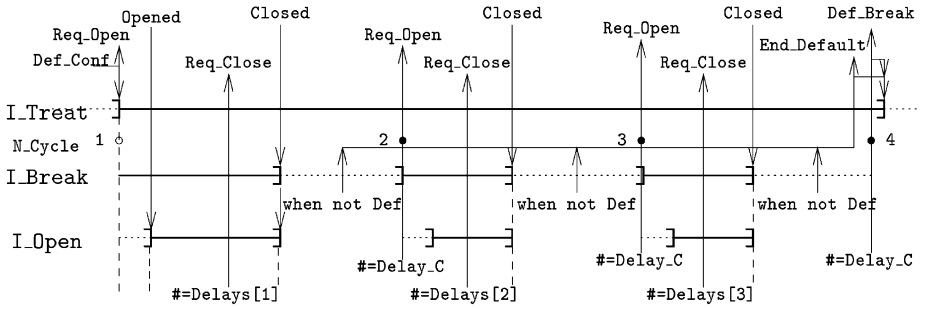

Fig. 6. The treatment phase: a series of intervals.

behaviors where the expression of properties would become either too difficult or too dependent on values which the DSP model abstracts away like complex schedulings. Other examples of graphical simulation of specific SIGNAL programs include a generic production cell controller [1,2], a speech processing system [19] and a robot vision system [27].

4.2.1. The simulation environment

The SIGNAL programming environment now includes a generic graphical simulation environment for SIGNAL specifications. It automatically constructs graphical input reading and oscilloscope-like output displaying windows for any of the interface signals of a program, as illustrated in Fig. 7.

We now display the presence and values of some of the intervals of the behavior, for some events. The oscilloscope-like display encodes intervals as 1 when `inside`, -1 when `outside` and 0 when absent. Events are encoded as 1 when present (which lasts only one instant), and 0 when absent. The logical input signals are always present, and are displayed as 1 when *true* and 0 when *false*. We will only focus on the simulation of the confirmation phase.

4.2.2. Simulation of the confirmation phase

The left column of Fig. 7 shows traces of the inputs (i.e. the three kinds of logical defects `PH`, `H` and `W`, and the event input `Ext_Defect`). The right column shows the hierarchical preemptive structure of the intervals during the confirmation phase, as well as the output event `Def_Conf`.

In the particular simulation trace illustrated in Fig. 7, the first event occurs at time 20 when the logical input `W` becomes *true*. Consequently, the interval `Int_PH` (the `I_PH` of Section 4.1.2) is opened, and `Int_H` is in its initial state `outside`. At time 60, after a `Delay_PH` of 40, `open Int_H` occurs, and the interval `I_W` is in its initial value `outside`. At time 90, after a `Delay_H` of 30, `open Int_W` occurs. At time 110, before the end of `Delay_W`, the defect `PH` becomes *true*. This interrupts the confirmation of the other defects, and emits `B_Def_Conf`.

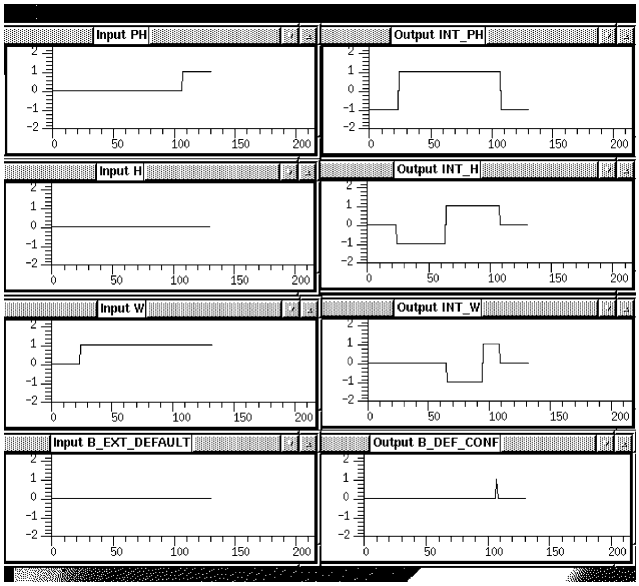

Fig. 7. The simulation of the confirmation phase.

Even though simulation is useful for partially validating a specification, formal analysis is still needed to prove that the system has good behavior, because all possible schedulings cannot be simulated.

4.3. Formal verification of the power transformer station

In this section, we apply the tools presented in Section 3.2 to check various properties of our SIGNAL implementation of the power transformer station. Translation of the SIGNAL program takes 10 s, during which the causal and temporal coherency of the program are checked and an executable code for the dynamical system is produced. The polynomial dynamical system obtained contains 12 state variables and 22 event variables, representing an automaton of 500 000 possible states. In fact, we must consider only the reachable states. For this, we have to compute the orbit of the system, which corresponds to the set of all states that can be reached from the initial ones. Using our representation by ideals and varieties, this set is characterized by a single polynomial. To obtain the number of different states, we have to count the number of solutions of the polynomial. In our case, the system contains 7000 reachable states and more than 55 million transitions.

We will now describe some of the properties that have been proved:

(1) *If a defect* `PH` *is detected after the end of its corresponding delay*, *in interval* `I_H`, *then the defect is confirmed by the emission of* `Def_Conf`: Two different methods can be used to prove this property.

The first method uses the SIGNAL compiler, which can prove static (i.e., time invariant) properties as mentioned in Section 2. We express the property in SIGNAL as an inclusion between clocks as follows:

```
(| synchro{(when PH in I_H), ((when PH in I_H) when Def_Conf)}
 |)
```

(where $A \subset B$ is expressed in the form $A = A \cap B$, with `when` as intersection, and `synchro` as equality for clocks). We compose this constraint with the controller, and the compilation of the whole checks the consistency of all the constraints on clocks in the specification including this one.

For the second method, in the treatment of the `PH` defect (i.e., inside the process that is active `each I_H`) we add the following lines to the original specification:

```
(| Error := not ( when Def_Conf when (PH in I_H))
            default when ( not Def_Conf when (PH in I_H))
 | Sigali(Reachable(B_true(Error)))
 |)
```

The `Error` signal is a boolean which takes the value *true* when the property is violated. In order to prove the property, we have to check that there does not exist any trajectory of the system which leads to the states where the `Error` signal is *true*. Using a new extension of the SIGNAL language, named SIGNAL +, it is now possible to express the property to be checked directly in the SIGNAL program. The keyword **Sigali** means that the sub-expression must be evaluated by our symbolic calculus system SIGALI. The function *Reachable* means that SIGALI has to check the reachability of the set of states where *error* is *true* (`B_true(Error)`), as described in Section 3.2. Thus, the compiler produces a file which can be read by SIGALI, in which can be found the polynomial dynamical system and the property to be checked. This file is interpreted by SIGALI, which sends back the answer as in the following trace:

```
-----------------------------------------
> load(''Controller.z3z'');
     => loading of the polynomial dynamical system
> Prop : B_true(Error);
     => Compute the set of states where Error is true
> Reachable(Prop);
     => Check for the reachability of this
      set of states from the initial states

 false
-----------------------------------------
```

(2) *The controller cannot be in states where both intervals* `I_PH` *and* `I_Treat` *are* `inside`: This property can be established by proving that the set of states corresponding to the situation where the treatment phase and the confirmation phase are both active, cannot be reached from the initial states of the controller (given by the declarations in the program). For that, we consider the two intervals `I_Treat` and `I_PH`, encoded by logical signals which are `true` when the system is in the corresponding phase, and we add to the SIGNAL program the following line:

```
(| SIGALI(Reachable(And(B_true(I_Treat),B_True(I_PH))))
 |)
```

The sub-expression `And(B_true(I_Treat),B_True(I_PH))` defines in the polynomial dynamical system, the set of states where `I_Treat`=1 and `I_PH`=1 at the same instant. Then, the proof verifies that this set of states is not reachable from the initial states of the polynomial dynamical system. In our example the result obtained is *false*.

(3) *If* `First_Defect` *occurs, then the controller will necessarily evolve in such a way that*: (a) *Either* `Def_Conf` *will be emitted with value* `true`.
(b) *Or* `Def_Conf` *will be emitted with value* `false`.
(c) *Or event* `Ext_Def` *occurs*.

To verify this property we build an observer. This is a process composed with the controller, which evaluates a boolean signal `OUT` which is *present* when any of the three possibilities occurs, `true` when (a) or (c) occur, and `false`, when (b) occurs.

```
(| First_defect := when (Defect and not (Defect $1))
                    default false
 | OUT := .... Definition of the boolean OUT in Signal
 | SIGALI(Attractivity(B_true(First_Defect),
                       B_true(event OUT)))
 |)
```

The property can be proved by checking the attractivity of the set of states where `OUT` is present, from the set of states where the defect appears (i.e. in which the event `First_Defect` occurs). The compiler produces a file, which is interpreted by SIGALI, by applying the proof system function computing attractivity, as in the following trace:

```
-----------------------------------------------
> load(‘‘controller.z3z’’);
> First_Defect: when (Defect and not (Defect_1)) default -1;
> OUT : ...Definition of the polynomial OUT;
```

```
> Prop_1 : B_true(First_Defect);
> Prop_2 : B_true(OUT^2);
   => OUT^2 means the set where the boolean OUT is present
> Attractivity(Prop_1,Prop_2);

 true
 -------------------------------------------
```

(4) *If* `Def_Conf` *occurs, then the controller will necessarily evolve in such a way that*: (a) *either the defect does not disappear and the signal* `Def_Break` *will be emitted,*

(b) *or the defect does disappear, with the circuit-breaker closed.*

To prove this property we use the same method as for (3). We compute the set of states E, where the defect is confirmed (i.e., `Def_Conf`$=1$), and the set of states F, where (a) or (b) are verified. Using the function that computes attractivity, we prove that F is an attractive set of states from the set of states E.

By combining formal verification with SIGALI and simulation, the most important requirements on the behavior of the departure cell (and also the link and arrival cells) have been validated.

5. Conclusion

This paper presents the synchronous approach to the specification and verification of discrete event control systems, applied to the preemptive controller of a circuit breaker for a power transformer station.

The specification, validation and implementation of complex control systems, implying permanent interaction with an environment, is treated by the data-flow language SIGNAL in a discrete event system framework. The possibility of formal verification makes SIGNAL particularly suitable for safety-critical applications. Transitions between different modes of activity, e.g. the sequencing of hierarchical data-flow tasks, are handled by the extension SIGNAL*GTi* [28], which is a language-level integration of the data flow and task preemption frameworks. In this way, the whole application can be specified in SIGNAL from the discrete event driven state-based behavior down to the servoing loops.

The verification of the power transformer station is based on the model underlying SIGNAL i.e. systems of polynomial dynamical equations over $\mathbb{Z}/3\mathbb{Z}$ [18]. These characterize a set of solutions which encodes the states and events. The method manipulates equation systems rather than solution sets, so that enumeration of the state space is avoided. The operations used on the equation systems are based on algebraic geometry (varieties, ideals and morphisms). They allow the treatment of *safety*, *liveness*, *reachability* properties. The SIGNAL approach to the verification of control systems has also been tested on other applications, such as a robotic production cell [1].

The equational nature of the SIGNAL language makes it natural to use an equational framework for modeling behaviors and proving their properties. This description of dynamical systems using equations is quite common in the fields of control theory and digital circuits, but not in verification and model checking. This aspect is an originality of the SIGNAL approach compared to others based on explicit transition systems. For example, the reactive languages ESTEREL [8] and LUSTRE [12] are compiled into finite state automata; hence they naturally interface with tools based on these formalisms like AUTO and AUTOGRAPH. The compilation of ESTEREL has recently also gone to Boolean equations-based representations of automata instead of explicit ones [7]. In principle, the two methods are equivalent, but in practice each is suited to a certain class of problems. In particular, compact representations based on systems of equations avoid the combinatorial explosion of explicit state-based representations. Both models support verification by the methods of model checking and comparison (bisimulation or behavioral equivalence), and as in the case of LUSTRE, some properties or observers can be specified in the language [12]. Given that polynomial dynamical systems are an implicit description of transition systems, it is possible to give a semantics of temporal logic formulae (for example the computational tree logic CTL) in terms of the algebraic operators, and perform symbolic model checking by evaluating them on a polynomial model. Note that the synchronous language LUSTRE uses the same methodology for verifying its programs, using Binary Decision Diagrams to encode the formulae [12].

Another possible use of the polynomial model is the automated synthesis of controllers, where algebraic methods are used to derivate, from a model of the system, a controller satisfying given properties and objectives such as *invariance* or *attractivity* [4,10,24,25]. In our application, this method is used to synthesize the interaction controller linking the various cells of the transformer station controller [23]. Another possible extension would be to prove properties that depend on the behavior of numerical variables, or in general on data other than the presence/absence and Boolean values which are currently handled.

Acknowledgements

The authors wish to thank the reviewers for their useful comments, and William Triggs for his help in improving the language.

References

[1] T. Amagbegnon, P. Le Guernic, H. Marchand, E. Rutten, Signal – the specification of a generic, verified production cell controller, in: Formal Development of Reactive Systems – Case Study Production Cell, Lecture Notes in Computer Science, vol. 891, January 1995, pp. 115–129 (Ch. VII).

[2] T. Amagbegnon, P. Le Guernic, H. Marchand, E. Rutten, The signal data flow methodology applied to a production cell, Technical Report No. 917, IRISA, March 1995.

[3] T.A. Amagbegnon, L. Besnard, P. Le Guernic, Implementation of the data-flow synchronous language signal, ACM SIGPLAN Notices 30 (6) (1995) 163–173.

[4] S. Balemi, G.J. Hoffmann, H. Wong-Toi, G.F. Franklin, Supervisory control of a rapid thermal multiprocessor, IEEE Trans. Automat. Control 38 (7) (1993) 1040–1059.
[5] A. Benveniste, G. Berry, Real-time systems designs and programming, Proc. IEEE 79 (9) (1991) 1270–1282.
[6] G. Berry, Preemption in concurrent systems, in: Proc. 13th Conf. on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science, vol. 761, Springer, Bombay, India, December 1993.
[7] G. Berry, The Constructive Semantics of Pure Esterel. Draft book available at `www.esterel.org`, 1999.
[8] F. Boussinot, R. de Simone, The ESTEREL language, Proc. IEEE 9 (79) (1991) 1293–1304.
[9] R.E. Bryant, Graph-based algorithms for boolean function manipulations, IEEE Trans. Comput. C-45 (8) (1986) 677–691.
[10] B. Dutertre, M. Le Borgne, Control of polynomial dynamic systems: an example, Research Report No. 798, IRISA, January 1994.
[11] T. Gautier, P. Le Guernic, Code generation in the sacres project, in: Towards System Safety, Proceedings of the Safety-Critical Systems Symposium, SSS'99, Huntingdon, UK, Springer, Berlin, February 1999.
[12] N. Halbwachs, F. Lagnier, C. Ratel, Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE, IEEE Trans. Software Eng. 18 (9) (1992) 785–793.
[13] N. Halbwachs, Synchronous Programming of Reactive Systems, Kluwer, Dordrecht, 1993.
[14] M. Jourdan, F. Lagnier, F. Maraninchi, F. Raymond, A multiparadigm language for reactive systems, Proc. IEEE Int. Conf. on Computer Languages, ICCL'94, Toulouse, France, May 1994.
[15] M. Le Borgne, Systèmes dynamiques sur des corps finis, Ph.D. Thesis, Université de Rennes I, September 1993.
[16] M. Le Borgne, A. Benveniste, P. Le Guernic, Polynomial ideal theoric methods in discrete event and hybrid dynamical systems, Proc. 28th IEEE Conf. on Decision and Control, Tampa, FL, December 1989, pp. 2695–2700.
[17] M. Le Borgne, A. Benveniste, P. Le Guernic, Dynamical systems over galois fields and deds control problems, Proc. 30th IEEE Conf. on Decision and Control, 1991, pp. 1505–1510.
[18] M. Le Borgne, A. Benveniste, P. Le Guernic, Polynomial dynamical systems over finite fields, in: G. Jacob, F. Lamnabhi-lagarrigue (Eds.), Algebraic Computing in Control, vol. 165, Lecture Notes in Computer and Information Sciences, March 1991, pp. 212–222.
[19] P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire, Programming real-time applications with signal, Proc. IEEE 79 (9) (1991) 1321–1336.
[20] F. Maraninchi, Y. Rémond, Mode-automata: about modes and states for reactive systems, in: Eur. Symp. on Programming, Lisbon, Portugal, Lecture Notes in Computer and Information Sciences 1381, Springer, Berlin, March 1998.
[21] H. Marchand, M. Le Borgne, The supervisory control problem of discrete event systems using polynomial methods, Research Report No. 1271, Irisa, October 1999.
[22] H. Marchand, E. Rutten, M. Samaan, Specifying and verifying a transformer station in signal and signalgti, Research Report No. 2521, INRIA, March 1995.
[23] H. Marchand, M. Samaan, Incremental design of a power transformer station controller using controller synthesis methodology, in: World Congr. on Formal Methods, FM'99, Lecture Notes in Computer Science, vol. 1709, Toulouse, France, Springer, Berlin, September 1999, pp. 1605–1624.
[24] P.J. Ramadge, W.M. Wonham, Supervision of discrete event processes, Proc. 21st IEEE Conf. Decision and Control, Orlando, FL, December 1982, pp. 1228–1229.
[25] P.J. Ramadge, W.M. Wonham, The control of discrete event systems, Proc. IEEE (Special issue on Dynamics of Discrete Event Systems) 77 (1) (1989) 81–98.
[26] E. Rutten, P. Le Guernic, The sequencing of data flow tasks in SIGNAL, Proc. ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems, Orlando, FL, June 1994.
[27] E. Rutten, E. Marchand, F. Chaumette, An experiment with reactive data-flow tasking in active robot vision, Software – Practice Exp. 27 (5) (1997) 599–621.
[28] E. Rutten, F. Martinez, Signalgti: implementing task preemption and time intervals in the synchronous data flow language signal, Proc. 7th Euromicro Workshop on Real Time Systems, Odense, Denmark, Juin 14–16, 1995.