Applicability of Signal in Safety Critical System Development

I.M. Smarandache and N. Nissanke

The University of Reading, Department of Computer Science
Whiteknights, PO Box 225, Reading RG6 6AY, United Kingdom
Tel.: (44) 118 931 8608, Fax: (44) 118 975 1994
I.M.Smarandache@reading.ac.uk, N.Nissanke@reading.ac.uk

Abstract

This paper concerns the specification of functional and safety requirements of a safety critical system independently, and subsequent validation of the integrated requirements in the synchronous data-flow formalism Signal. *The* Signal *language provides a unified framework for hierarchical development of systems by refinement of specifications. Validation of specifications can be realized through a formal calculus based on the semantics of the language or through functional simulation. In this context, the research reported attempts safety management in critical real-time systems in an application independent manner. This is based on the possibility of identifying low and high risk states in a system and expressing its functional behaviour as transitions between pairs of states of different levels of risk. Safety requirements become conditions prohibiting low to high risk transitions, or forcing high to low risk transitions in a timely manner. A safety module, which is in charge of ensuring that these conditions are observed at run-time, is developed in* Signal *in order to explore the applicability of the* Signal *language to safety-critical design and to establish both the feasibility and the extent of managing safety concerns in an application independent manner.*

1. Introduction

Real-time systems, that is, systems that provide a given service in a timely manner, become *safety critical*, if their failure could place human life in danger or pose a serious threat to property, equipment or the environment. An example is a railway signalling system, the main function of which is to detect the presence of trains on tracks and to ensure their *correct* movements [1]. Failures in the signalling system, that is, failures caused either by inconsistencies in the design, physical failures in the equipment on the ground or failures of the controlling system, might result in collisions between trains with catastrophic consequences.

The use of computers for the control of safety-critical real-time applications has given rise to the term of *safe software*, that is, software which does not generate a system-related *hazard* [2]. Safe software should be a *correct* implementation of safety requirements established by the system's hazard analysis and should incorporate safeguards against faults occurring in the software as well as in the underlying hardware [4].

1.1. A generic approach to safety management

This paper proposes a *generic* approach to handling safety in critical real-time applications. It is based on the observation that, broadly speaking, safety requirements of different applications belong to several categories and that they can be handled in a uniform manner. A classification of safety requirements suitable for this purpose may be found in our previous work [5, 6, 7], albeit devoted to formal specification of safety requirements. The latter works demonstrate this in the Z notation, deontic logic and duration calculus respectively, but follow the framework due to Leveson [4].

Our approach is based on the observation that hazards of a system can be associated with high risk states of its components and the safety requirements can be specified with respect to transitions between low risk (LR) and high risk (HR) system states. Functional requirements also concern the transitions in a given component, or the system itself, from a low risk to a high risk state in order to deliver a service. These are referred to here as low-to-high risk transitions (LH). Conversely, a high-to-low risk transitions (HL) are typically associated with the fulfillment of the service and the return of the system to a state where another service can be delivered. However, safety is not an issue to be addressed as a part of functional requirements. Safety requirements deal with forbidden or mandatory transitions between low risk and high risk system states, irrespective of whether the relevant transitions are required as part of a service or in order to avert a hazard or a failure.

In some situations, a LH transition may lead the entire system to a potentially dangerous state and, therefore, should not be permitted. This is expressed as a safety requirement that stipulates the conditions under which the respective transition should be prevented. For example, in railway signalling, the allocation of a route to a train may require certain signals along the route to change their aspect (displayed colour) from red (LR state) to green (HR state). Obviously, these transitions should not be allowed if the route concerned is already in use by another train. This is a typical safety requirement on a LH transition.

There are also safety requirements which stipulate the conditions under which a component should execute a HL transition within a given interval of time. Since they may be required in the presence of failures, they are sometimes referred to as 'fire-fighting' requirements. In railway signalling, changing the aspect of a signal from green (HR state) to red (LR state) within a specified time limit may be required on grounds of either as a precautionary measure after the intended train has passed the signal, to prevent other trains from entering a section protected by it, or as a failure handling action dealing with a failed point or a failed signal ahead of the signal concerned.

Thus, safety requirements related to procedures applicable amid failures and hazards, or intended to prevent potential failures, may be classified into a number of distinct categories. The fact that these categories of safety requirements can be found across different applications suggests the possibility of dealing with safety in an application independent, or 'generic', manner. This paper explores this idea with respect to safety in railway signalling by considering a 'safety supervisor' – a separate safety supervising module. An implication of this is that the controlling system and the safety supervisor can have two clearly defined roles, namely, the former with the responsibility foremost for the service as envisaged by the functional requirements, and the latter for its implications on safety. Typically, the safety supervisor contains *fail-safe* and *fail-soft* procedures for handling failures of the controller, as well as the failures in data communication between the controlling and controlled systems. As evident from works such as [1, 8], railway signalling is also a popular case study in computer science literature because of its many interesting safety critical features.

1.2. Formal methods in safety-critical systems design: the SIGNAL language

The concept of safety supervisor is a solution to the management of hazards and failures in the physical system. However, the safety supervisor must itself be safe and, therefore, needs to be proved correct and protected from any faults in the software implementing it. This can be done by formal verification of the design, simulation, and by means of extra fault tolerance features. With the above in mind, the controller and the safety supervisor for railway signalling is specified in SIGNAL.

Our choice for SIGNAL is based primarily on the design needs outlined above. Fault avoidance by correct design is one of the traditional uses of such a language. In this respect, on-line management of safety by prevention and handling of system failures as discussed here is a new application of SIGNAL in safety critical work. As discussed below, SIGNAL provides a unified framework for the specification of both the functional behaviour and the safety mechanisms of the application. The underlying mathematics, as well as the hierarchical process development by refinement promoted by the block oriented graphical user interface [16], makes programming in SIGNAL a relatively easy task.

SIGNAL is a *synchronous* language [9, 10] built on the data flow model and enhanced with a powerful environment suitable for a range of activities, in particular, specification, verification and simulation. Introduced in [11], the synchronous paradigm has generated a large class of semantically well understood languages, either imperative like STATECHARTS [12] and ESTEREL [13], or declarative like LUSTRE [14] and SIGNAL.

The synchronous programming paradigm [9] is based on the following hypotheses: 1) all actions in a system have zero *logical* duration; 2) two or more actions can take place at the same logical instant, such actions being termed 'simultaneous'. (Note that logical instants are characterised by the presence of values for the SIGNAL variables. The set of instants where a variable is present and has a value is referred to as the *clock*.) By considering logical instants instead of the more intricate real time, as well as their various relations with respect to succession, simultaneity, etc., these hypotheses permit a more convenient abstraction of the temporal aspects of any application and give rise to a formal calculus based on logical properties of instants for verifying correctness. Note that, in this respect, other approaches have to overcome problems related to the manipulation of durations and non-determinism. By contrast, these problems are eliminated in the synchronous paradigm by manipulating logical time. However, issues such as compliance with timing constraints are not totally removed, but deferred to later stages of the design, when the system is considered correct from the point of view of properties on logical instants.

The synchronous hypothesis and the data flow aspect of SIGNAL define its semantics, in the abstract form of a set of synchronization constraints on clocks and a graph of dependencies between data (values of variables). The abstraction constitutes the basis of a formal calculus concerning the logical properties of the program. The SIGNAL compiler performs the validation of a SIGNAL specification by verifying the consistency of the synchronization constraints expressed on clocks and by checking the absence of cycles in the data dependency graph. The result of the compilation is a data dependency graph with each arc labelled by the clock of its validity and each node having assigned the clock of its presence. The compiler also enables the derivation of a total order on clocks; when such an order can be inferred, the maximal element is used for the sequencing the operations defined by the data dependencies. Executable C code can be generated from a SIGNAL specification for functional simulation.

Clock constraints and data dependencies expressed in a SIGNAL program are examples of properties that should be satisfied by the specification. Additional properties can be attached to a program by process composition and verified by compilation or by a more powerful formal calculus using the system SIGALI [15] integrated into the SIGNAL environment. These include correct synchronization of different processes, properties such as deadlock and liveness. When neither of these approaches is applicable, the specification may be tested through simulation.

The paper is structured as follows. Section 2 focuses on the approach to specification and validation of critical systems in SIGNAL from a functional point of view. Section 3 outlines the generic approach to the management of safety. The safety supervisor incorporates three distinct categories of safety requirements which act as a filter on the events triggered by the controller, introduced in Section 2. The section contains an example of a generic safety rule in SIGNAL and concludes with a discussion on a complete and correct design of the safety supervisor. Finally, Section 4 provides a summary of results and prospects for the future.

2. Specification and validation of the railway signalling application in SIGNAL

Being a real-time system, railway signalling consists of two parts: the *controlled system* which incorporates the physical equipment on the ground, and the *controlling system* with the role in setting safe and timed configurations of the physical system in order to direct trains as requested. Communications between the two subsystems take the form of *sensor data* conveyed to the controlling system and *control signals* issued by the controlling system to the *actuators*. This kind of communications fits in neatly with SIGNAL *processes*, capable of describing the pattern of behaviour of a communicating reactive system.

2.1. Railway signalling

The signalling system consists of two major parts: the *permanent way* and the *interlocking*. The former contains the physical items such as *tracks*, *points* and *signals* placed in a certain configuration. Associated with each track is a bi-positional *relay* detecting the presence of trains on the track. *Points* are situated at junctions and can be controlled remotely in order to route a train in a specific direction. They can be in two extreme positions: *normal* and *reverse*, can be *locked* in the respective positions, and are equipped with sensors which let the controlling system know of the current position of each point. A point has to be *unlocked* prior to its movement. Finally, *signals* are the means by which train drivers are instructed to stop or are allowed to proceed. Signals are also equipped with sensors which indicate their aspect, namely, whether the signal is showing *red*, *green* or *yellow*.

The interlocking is an intermediary between the operator and the permanent way which monitors the latter and maintains it in a safe state by preventing any dangerous requests originating from the operator.

2.2. The SIGNAL language

The structure of the railway signalling system can be modelled in SIGNAL directly using its graphical user interface. Figure 1 shows a simple reactive system with two sensors and one actuator. Its controlled and controlling systems have been represented as two SIGNAL processes communicating via interconnected inputs and outputs (also known as *ports*).

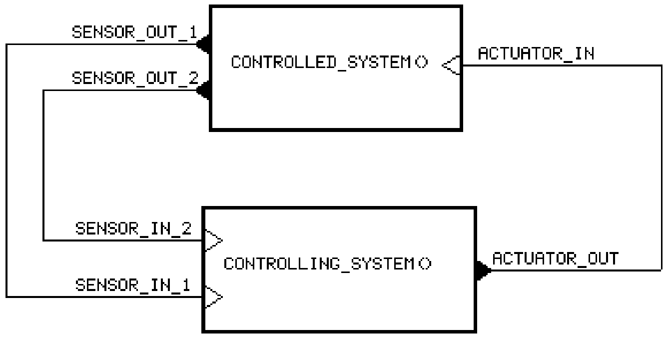

Figure 1: The structure of a reactive system in SIGNAL.

Data transmitted via a communication channel between two ports, say, for example, over the channel connecting ports **SENSOR_OUT_1** and **SENSOR_IN_1**, take the form of finite or infinite sequences of typed values. The values of a sequence belong to a SIGNAL variable which will be identified as a *signal*[1]. In the example in Figure 1, signals

[1]The word 'signal' appears in this paper in two different contexts: with respect to railway signalling and SIGNAL.

SENSOR_OUT_1 and **SENSOR_IN_1** are identical as they are characterized by the same sequence of values; so are the signals **SENSOR_OUT_2** and **SENSOR_IN_2**, and the signals **ACTUATOR_OUT** and **ACTUATOR_IN**.

Associated with each sequence of values defining a signal is a sequence of logical instants, represented usually by natural numbers, in ascending order. This set of logical instants represents, according to the semantics of SIGNAL, the times when the associated values are present and is referred to in SIGNAL as the *clock* of the signal. SIGNAL processes manipulate signals implicitly or explicitly by imposing constraints on their values and clocks. It has been observed that a process communicates with the external world through an interface consisting of its inputs and outputs. Input signals are transformed according to the actions defined in the internal structure of the process and the results are conveyed to a subsequent process through the output signals.

As illustrated in Figure 1, a SIGNAL process can be hierarchically decomposed into a set of communicating processes. SIGNAL supports four indivisible *basic* or *primitive* processes. These basic processes constitute the kernel of the language as they can be used in the construction of more complex SIGNAL processes.

The next section illustrates the decomposition principle in SIGNAL with a detailed specification of the controlling system of a railway signalling system. As seen later, our decomposition follows the partitioning of the signalling system into two subsystems: one dealing with its functional behaviour and the other overseeing the compliance of safety timing requirements (see Section 1.1). Our main interest here is in the specification and validation of the controlling system with respect to its functional behaviour; the management of the safety requirements is detailed in Section 3.

The SIGNAL process modelling the points serves as an example introducing the syntax and semantic of SIGNAL, as well as illustrating their use in the specification of the required functionality of points. The example also illustrates some aspects of the validation process of the specification, in particular, synchronization constraints on clocks, the verification of their consistency by the compiler and correct manipulation of states.

2.3. Using SIGNAL for functional behaviour specification

Figure 2 depicts the controlling part of the railway signalling system as the SIGNAL process `RAILWAY_CONTROLLER`. It receives information from the permanent way through sensors and, in response to *route* requests by the operator, instructs controllable components such as railway signals and points through Boolean SIGNAL variables that activate the required functionality. Routes are dependent on the junction layout and each route is defined through subsets of points, railway signals and tracks as appropriate. Thus, routes are a logical entity with no physical counterpart. Routes are defined here statically through a parameterized controller but are selected dynamically at run-time for execution.

In order to respond to route requests in a timely and safe manner, the controller maintains an up-to-date record of the overall state of the controlled system and schedules the required services accordingly. The controller also ensures that the operator has access to information about the current state of the system, especially with respect to the required services. The controller interface supporting these needs can be defined by:

- *Parameters:* Variables, the values of which remain fixed for each execution of a SIGNAL program, are specified as process parameters. In our case, they define the junction layout and concern the number of routes, points, railway signals and tracks. This kind of information is provided by matrices `P_BLG_R` and `S_BLG_R` defining, in effect, the applicability of points and railway signals to routes. Boolean matrices `REQ_NOR_PAR`, `REQ_REV_PAR` and `REQ_YELL_PAR` `REQ_GR_PAR` specify the desired position of each point and the desired colour of railway signal for each route. Integers `PSI`, `EPS1`, ... concern the temporal behaviour of the system.

- *Inputs:* `LOCKED`, `INNOR`, `INREV` are vectors of Boolean signals describing the corresponding sensors for points. $\texttt{LOCKED}_i$, $\texttt{INNOR}_i$ and $\texttt{INREV}_i$ are *true* whenever the ith point is *locked*, in *reverse* or in *normal* position respectively. Similarly, sensors `RED,` `YELLOW` and `GREEN` describe the aspect displayed by railway signals. The sensor set `RELAYUP` indicates the position of the relays on the layout. Finally, the Boolean vector `REQ_SET` models route requests issued by the operator and the scalar `TIME` is the reference logical clock of the system.

- *Outputs:* The outputs of the controller include control signals to actuators requiring the delivery of a specific service and state variables informing the operator about the outcome of the requested service. For example, a *true* value of the Boolean signals $\texttt{NOR\_ACT}_i$, $\texttt{REV\_ACT}_i$ and $\texttt{UNL\_ACT}_i$ is equivalent to a command addressed to the ith point to move to the normal or reverse position, or to unlock itself, respectively. The operator is informed of any point failure through the Boolean scalar `POINTFAIL` and the location of the failure through the Boolean vector `POINTDEF`, a *true* value of $\texttt{POINTDEF}_i$ meaning that the ith point has failed.

An attempt to allocate, or *set*, a route may not succeed either due to an equipment failure or because of an allocation of the entire route, or a part of it, to a prior route request and for remaining so. This is true of the ith route if $\texttt{ROUTEFAIL}_i$ has the value *true*. Subsequent route requests may be handled on basis of the current state of routes in the entire system; a *true* value of $\texttt{ROUTESET}_i$ indicating that the route i is set, and unset otherwise.

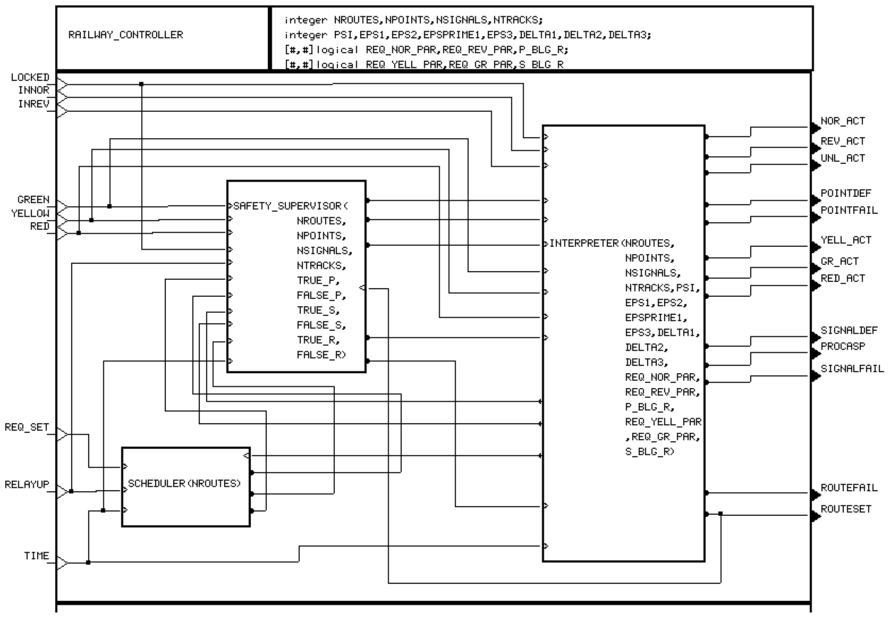

Figure 2: Modelling the railway signalling system in SIGNAL.

The process `RAILWAY_CONTROLLER` can be hierarchically decomposed into three subsystems: an `INTERPRETER` describing all 'normal' behaviours, that is, the behaviour pattern as required by functional requirements, of the physical and logical entities involved, a `SAFETY_SUPERVISOR` constraining the normal behaviour patterns in order to exclude dangerous patterns, and a `SCHEDULER` specifying the required sequencing of actions in a way that they meet the functional and safety requirements. `INTERPRETER` translates the services required into commands, which transform the state of the various entities in the system.

The `INTERPRETER`, in conjunction with the `SCHEDULER`, can provide an executable and meaningful behaviour pattern. The design of the `SCHEDULER` assumes that the route requests issued by the operator are free from deadlock and that they may not necessarily be optimal with respect to time and resources. Reliance on the operator's experience, however, may not be the best solution because of the complexity of issues such as deadlock avoidance. Although not addressed in this paper, an optimal scheduling algorithm preventing deadlock may be derived from the specification. The primary goal of the `SCHEDULER` in its present form is to prevent conflicting requests for routes with shared resources such as points and signals, as this may result in their unpredictable behaviour. For this reason, the `SCHEDULER` schedules route setting requests depending on the synchronisation signals sent by the `INTERPRETER` and if there is at least one unset route in the system. As a result, any route request rejected by the `SCHEDULER` has to be repeated by the operator. The `SCHEDULER` unsets routes only on the basis of information provided by relays on train movements.

In conjunction with the `INTERPRETER` and the `SCHEDULER`, the `SAFETY_SUPERVISOR` acts as a filter on commands addressed to the physical controllable elements. Its aim is to avoid dangerous situations that may result from concurrent access to resources or from equipment failures. The `SAFETY_SUPERVISOR` is discussed further in Section 3.

2.3.1. Functional requirement specification – an example

The `INTERPRETER` is further decomposed in accordance with the configuration of the application, that is, the architecture of the resources involved. A local controller is associated with each component, whether it is a signal, a point or a logical entity such as a route. In the case of points, the controller basically implements the expected behaviour of points following a request. As requests can be various, the controller can be further decomposed into the processes `POINTS_REQMOV` and `POINTS_UNLOCK`, which monitor the response of the points concerned, respectively, to a request to move (*reqmov*) or to unlock (*unlock*). These two processes are subsequently composed with the process `POINTS_STATE`, which is responsible for monitoring the overall state of points.

Let us consider in some detail the process `POINTS_STATE` in Figure 3; it is specified mainly by composition of basic processes supported by the SIGNAL kernel. With the exception of the `ARRAY`, each process corresponds to an equation (using the assignment symbol `:=`) on signals, which can be inputs, outputs or local variables, or to a relation such as `synchro`. The processes are then composed together by using the operator `|` to form the process `POINTS_STATE`. Local variables are confined exclusively to `POINTS_STATE` for its internal use using the restriction operator `/`. Considering the composition of the first four processes represented above, which are also reproduced in Figure 4, the composition operator has the role of connecting the inputs and outputs of the processes involved

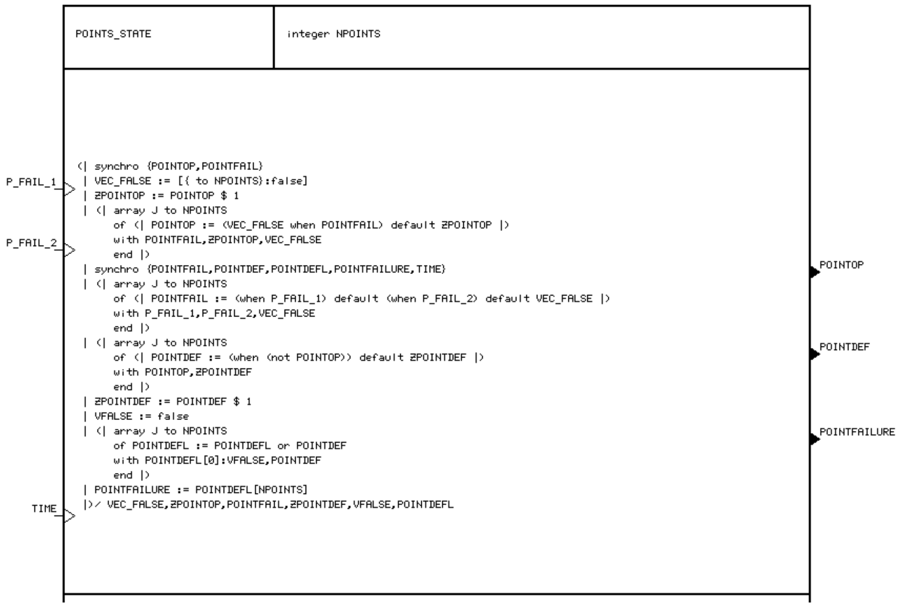

Figure 3: Management of point states in SIGNAL.

following the same principle as that on interconnecting channels in block-diagrams.

```
(1)      (| synchro {POINTOP,POINTFAIL}
(2)       | VEC_FALSE := [( to NPOINTS):false]
(3)       | ZPOINTOP := POINTOP $ 1
(4)       | (| array J to NPOINTS of
(5)             POINTOP := (VEC_FALSE when POINTFAIL) default ZPOINTOP
(6)            with POINTFAIL,ZPOINTOP,VEC_FALSE
(7)            end |)
(8)       |)
```

Figure 4: Illustration of the SIGNAL kernel.

Below is a brief description of the four basic SIGNAL processes relevant to the example in Figure 4. Note that when dealing with an equation using the symbol `:=` , the inputs appears on its right hand and the outputs on its left hand side.

- Mathematical functions and predicates (Boolean functions) have natural extensions to signals. For example, the binary function + for addition in mathematics has its counterpart in SIGNAL applicable to signals. Equation (2) is an example involving a constant function; each of the `NPOINTS` elements of the vector `VEC_FALSE` has the value *false*, when `VEC_FALSE` is present. In other words, $(\texttt{VEC\_FALSE}_t)_i = false$, with $1 \leq i \leq$ `NPOINTS`. An example of a Boolean function is `VEC := VEC_1 or VEC_2`, involving the Boolean operator `or` on signals.

- The `$ 1` operator delays a signal by one instant with respect to logical time. For example, an implication of the equation `(3)` is that $\texttt{ZPOINTOP}_t = \texttt{POINTOP}_{t-1}$ for every logical instant t when `ZPOINTOP` and `POINTOP` are both present; see the next section for more about instants of presence of signals, that is, clocks.

- The `when` constructor enables the definition of a sampling of a signal by a Boolean condition, while the `default` operator defines its output as a merging between values of the inputs. Consider, as an example, the equation `(5)`, rewritten as `(| VAR := VEC_FALSE when POINTFAIL | POINTOP := VAR default ZPOINTOP |)`. The signal `VAR` is present and carries the value of `VEC_FALSE`, whenever `POINTFAIL` is present and has the value *true*. Furthermore, `POINTOP` is defined either by values of `VAR` or `ZPOINTOP`, with the priority given to `VAR` when both `VAR` and `ZPOINTOP` are present.

Basic SIGNAL processes express relations on values and logical instants of presence, i.e. *clocks*, of their input and output signals. They are sufficient for the construction of any complex SIGNAL program. However, a number of extensions (macros) have been introduced for convenience in programming. The `synchro` operator forces its arguments to have the same clocks, that is, to be synchronous. The `array` is an iterative operator allowing simple and efficient expression of algorithms on multi-dimensional signals. Each signal appearing in equation `(5)` is considered indexed by `J`; the `array` process (eqs. (4)-(7)) is equivalent to a set of, possibly interconnected, processes of the form `(| POINTOP[J] := (VEC_FALSE[J] when POINTFAIL[J]) default ZPOINTOP[J] |)`, for each value of `J`.

In Figure 4, the signal `POINTOP` is a vector, each element of which expresses the state of a point, that is, whether it is operational or not. The `J`th point becomes non-operational, which is expressed by $\texttt{POINTOP}[\texttt{J}] = false$, when a failure of the point has been observed, i.e. $\texttt{POINTFAIL}[\texttt{J}] = true$; otherwise the point remains operational, which is equivalent to saying that `POINTOP` retains its last value `ZPOINTOP`; the initial value for `ZPOINTOP` being *true*. The value of `POINTFAIL` is set by the processes `POINTS_REQMOV` and `POINTS_UNLOCK`.

Concerning the controllers of points, the process `POINTS_REQMOV` shown in Figure 5 implements the functional behaviour of a set of points as a result of a request `REQ_NOR` or `REQ_REV` to move to the normal or the reverse position. According to its functional requirement [7], every point is expected to obey any command to perform a movement by executing it and to lock itself in the required position within a prescribed time interval. If it fails to accomplish this, the controller is to assume that the point concerned has failed.

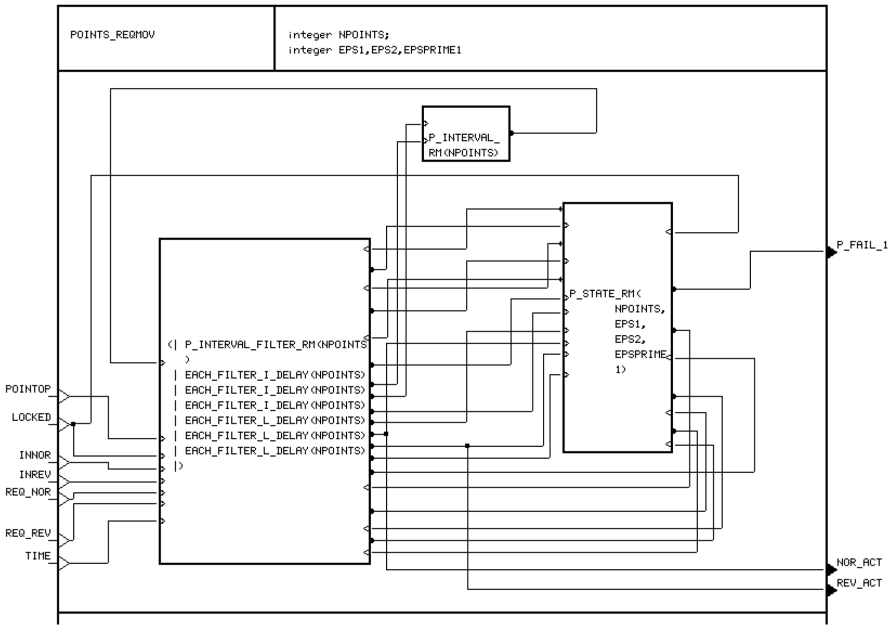

Figure 5: SIGNAL model depicting the functional behaviour of points.

In providing a service, points continually perform transitions between states where they are either out of service or operational, in the latter case being locked in the normal or reverse positions, or unlocked and free to move. Modelling states and transitions between them in SIGNAL, a data flow formalism with no notion of state, can be achieved through a *time interval* mechanism implemented by the SIGNAL preprocessor SIGNALGTI [17]. The preprocessor is entirely programmed in SIGNAL and offers the basic functionalities, but it is still in a prototype state to be used in a complex application. Therefore, we have implemented our own time interval mechanism inspired from SIGNALGTI and containing: interval definition; the possibility of associating data flow processes with intervals, thus implementing the notion of *task*; task manipulation through sequencing, preemption, etc. In the case of the controller `POINTS_REQMOV`, a time interval is defined from the logical instant a movement request has been issued and until the point has been locked in the required position or it has been declared non-operational. This can be expressed in SIGNAL by the equation `INT := ]BEGIN, END]` which defines `INT` as the interval starting with the occurrence of a *true* value for signal `BEGIN` and ending with event `END` becoming *true*.

Variable INT is a Boolean signal holding the *true* value between the limits of the interval; it is implemented by process P_INTERVAL_RM.

As its name suggests, the role of P_INTERVAL_FILTER_RM is to associate the data flow process P_STATE_RM to the interval INT by filtering away the set of actions that are not supposed to take place within the interval. Process P_STATE_RM monitors the state of each point during movement.

2.3.2. Verification of clock constraints

Functional design of the signalling system follows a process of hierarchical refinement, but hand in hand with a verification of logical correctness of each subsystem, and the system as a whole, by compilation. This kind of verification applies also to clock constraints, which amount to additional synchronization constraints on existing processes.

Clock constraints form a part of the abstraction of any kernel process, the other being a data dependency graph between signal values. In the case of more complex SIGNAL processes, constructed using basic operators, these two components are built by composition of clock constraints and juxtaposition of the data dependency graphs of basic processes. However, this must conform with the logical properties of the SIGNAL program concerned and is verified by the SIGNAL compiler. Verification of the clock constraints takes place in parallel with the synthesis of the clock with the greatest number of logical instants, which is further used for program execution.

Figure 6 presents the clock constraints (equations) associated with the SIGNAL program shown in Figure 4, the numbering being identical to the corresponding program line numbers.

$$
\begin{array}{ll}
(1) & c_{\texttt{POINTOP}} = c_{\texttt{POINTFAIL}} \\
(3) & c_{\texttt{ZPOINTOP}} = c_{\texttt{POINTOP}} \\
(5) & c_{\texttt{POINTOP}} = (c_{\texttt{VEC\_FALSE}} \wedge [\texttt{POINTFAIL}]) \vee c_{\texttt{ZPOINTOP}}
\end{array}
$$

Figure 6: Illustration of the clock calculus in SIGNAL.

Clocks of different processes are independent, unless stated explicitly, or implicitly through a SIGNAL program. The `synchro` operator in the first equation expresses a constraint explicitly on its arguments, namely, that their clocks are identical. The delay operator `$` , on the other hand, imposes such a constraint implicitly on its operands, namely, that they share the same clock. The `when` operator defines the clock of its output

$c_{\texttt{POINTOP}}$ as the intersection of the clock of the first operand $c_{\texttt{VEC\_FALSE}}$ and the set of the instants when the second argument POINTFAIL is present and has the value *true*, the latter being denoted by [POINTFAIL]. Finally, the clock of the output of the default operator is defined as the union of the clocks of its operands.

The reasoning behind the verification process is a consistency check of the above clock constraints and is as follows. VEC_FALSE being a constant signal, its clock is defined by the context, which in the case of (5) is the when operator. Therefore, we have in (5) that $c_{\texttt{VEC\_FALSE}} = [\texttt{POINTFAIL}]$. As a result, it follows from (3) and (5) that $[\texttt{POINTFAIL}] \subseteq c_{\texttt{POINTOP}}$. In the absence of any further explicit relation between $c_{\texttt{POINTOP}}$ and $c_{\texttt{POINTFAIL}}$, the process defined by (3) and (5) is non-deterministic. In other words, the process is under-specified from the point of view of its clocks and, therefore, cannot be executed. For execution, the process must be placed in an environment which provides the missing clock information but in a form consistent with its existing clock constraints. This is provided by the synchro operator in the clock constraint (1). This kind of verification has a significance at the implementation level; the SIGNAL process represented in C code is executed at the instants of the clock $c_{\texttt{POINTOP}}$, which is identical to the clock of the input POINTFAIL determined by the environment.

Another example is a process that implements a time interval. Let us consider a time interval of the form]BEGIN, END], ranging from the instant BEGIN, but excluding that instant itself, to the instant END inclusively. Let this interval concern a request for moving a point, in which case BEGIN and END can be defined as

```
BEGIN  :=  (true when (REQ_NOR or REQ_REV) when POINTOP
               when (not LOCKED)) default false
  END  :=  (true when LOCKED) default (true when (not POINTOP))
              default false
```

Implicit in the above definition are the clock constraints: $[\texttt{BEGIN}] = ([\texttt{REQ\_NOR}] \vee [\texttt{REQ\_REV}]) \wedge [\texttt{POINTOP}] \wedge [not\ \texttt{LOCKED}]$ and $[\texttt{LOCKED}] \wedge [not\ \texttt{LOCKED}] = \emptyset$. As a consequence, we have $[\texttt{BEGIN}] \wedge [\texttt{LOCKED}] = \emptyset$. In practical terms, the implication of this result is that no point can begin to move when it is locked. In a similar way can be verified that no action is initiated on a non-operational point ($[\texttt{BEGIN}] \wedge [not\ \texttt{POINTOP}] = \emptyset$). Moreover, an operation (a point movement) under execution over an interval will be protected from further requests addressed to the same point and made within the same interval and, thus, prevent any unpredictable behaviour.

3. Specifying and verifying safety provisions in SIGNAL

3.1. High risk and low risk states

As with any application, hazards in railway signalling need to be associated with states of different risk levels. This concerns an assessment of low risk and high risk states of each equipment, as well as each logical entity such as a *route*. Functional behaviour of a component consists of a sequence of transitions from LR to HR states in order to provide a required service and, at the service fulfilment, a sequence of transitions from HR to LR states in order to return the equipment to a state where it can respond to a fresh request.

In the case of points, when they are operational the *locked* state is considered to be a lower risk state than the *unlocked* one, since in the latter case a point is free to move, giving rise to a greater risk in the event of a train passing over it. Therefore, the locked state is considered *safe* and is also used as the initial state of a point (safe initialisation). On the other hand, since a broken point cannot be controlled to any state as part a failure-averting or recovery action, the *broken* state is considered to be of the highest risk of all states of a point. Therefore, the point states in the increasing order of risk are as follows:

$$locked < unlocked < broken$$

In providing a service, a point should be first unlocked and subsequently locked in the desired position. Thus, when moving a point, it is first controlled to a high risk state, and subsequently into its lowest risk state to wait for further requests. The occurrences of value *true* in `REQ_UNL` and `REQ_NOR`, or `REQ_UNL` and `REQ_REV`, represent the triggers of the above transitions; they are conveyed to the physical equipment as the control signals `UNL_ACT` and `NOR_ACT`, or `UNL_ACT` and `REV_ACT`; see Figure 5.

Note that the above behaviour has no constraints other than the timing requirements provided by the functional requirements. However, these transitions are constrained by requirements imposed on the overall signalling system, either aimed at preventing train collisions resulting from resource conflicts or designed to avert hazards resulting from equipment failures. The latter assumes that there are no fault tolerance features, or that there is a limit to fault tolerance.

3.2. The safety supervisor

In this connection, we consider a specialized module, referred to here as *safety supervisor*, with the role of handling safety aspects of the system. The safety module represents an additional layer of protection and, as envisaged here, acts as a filter on the events triggered by the controller. The filtering is realized through predicates defined on the state of the overall system, aimed either at preventing a LH transition taking place, or forcing a HL transition within a specified deadline. For example, a point is to be unlocked only when the track where it is located is clear and, in addition, other tracks preceding it on the route concerned are also clear. The negation of this condition constitutes the predicate G_P used here to prevent the above transition taking place. However, there is no filtering on the reverse transition *unlocked* $\longrightarrow$ *locked*, since a point is unlocked usually in order to be locked in a different position.

In a similar manner, LR and HR states are also associated with railway signals and routes. For example, the ordering on states of railway signals according to increasing risk is *red* < *yellow* < *green* < *broken*. The safety supervisor allows transition *red* $\longrightarrow$ *green* to execute only if the predicate G_S is true, where G_S is true only if 'the signals ahead are alight and all the tracks of the route concerned are clear'. Moreover, when $\neg G_S$ holds, the reverse transition *green* $\longrightarrow$ *red* is forced to execute within a certain time limit imposed by the specification. Normally, the controller is in charge of triggering this transition once a train has passed the signal, but this may be delayed depending on other events in the system. In this case, the role of the safety supervisor is to ensure a timely execution of the transition.

In addition to the aspects considered above, the safety supervisor must also consider the scenarios in which the information provided by sensors is not accurate, or the controller itself has failed to command equipment as required. These situations come under failures in *sensor data* and *control signals*, and are managed on the basis of the so–called *safe* and *unsafe* states. These states represent, respectively, the lowest and highest risk states of the corresponding equipment. When a failure occurs in sensor data, the corresponding item is to be treated as if it were in its unsafe state. In the case of a signal or a point, the unsafe state is the *broken* state. The aim is to ensure that, in the event of such a failure, the system continues to provide the services with degraded performance, or at a reduced functionality; this corresponds to the *fail-soft* approach. On the other hand, when a control signal fails, the corresponding item must be controlled into its safe state, a situation referred to commonly as *fail-safe*. In other words, in the context of signalling, points are locked and signals are turned to red.

Furthermore, as envisaged here, the safety supervisor is also expected to attend to matters such as the safe initialisation of the system after a recovery action (a repair of a physical fault), and up-dating the context following a change in the system configuration. With failures, the system may not provide the services in the same manner as when it is functioning normally. Consequently, the predicates and time intervals associated with transitions, defining the context of the activities related to safety, may change from one configuration to another. For example, in the case of a signal forced into its safe state (red) as a result of a fail-safe action, the signal should not be allowed to proceed to green until the controller is restored to its functional mode and it retrieves the correct system state. Obviously, this is an additional condition compared to the situation when the controller works correctly.

Safety requirements discussed above can be classified as:

- *Function-related* transitions, shown by LR $\xrightarrow{\neg G}$ HR, which execute while G is false. This kind of requirement falls under the scope of *failure preventative* category and is intended to prevent failures 'by design'.

- *Failure-averting* transitions concern primarily failure avoidance. They take the form HR $\xrightarrow{G,\theta}$ LR. Such a transition is forced to execute within θ units of time when G begins to hold.

- *Failure handling* transitions address safety procedures applicable amid failures. They are shown as (HR $\xrightarrow{G,\theta}$ LR)$_{ext}$ and are an extended form of failure-averting transitions: when G holds, the item is forced into its LR state within θ and is made to *remain in this state* until a further instruction. Failures in sensor data and control signals are covered under these requirements. For instance, when the controller fails, the safety supervisor ensures that each item enters its safe state and remains there as long as necessary.

  Note that safe initialisation after a fault repair can be described as an extended HL transition, so are the actions to be undertaken as a result of a failure of equipment, such as its isolation by forcing HL transitions on nearby equipments.

Obviously, the safety module must be equipped with mechanisms to detect failures. At the level of SIGNAL specification, failure detection can be modelled by Boolean signals turning *true*. However, in practice failure detection in the sensor data and control signals may be complex and is outside the scope of this paper. In addition to watchdog timers and other tests to detect timing failures at the level of the controller, other means for detecting failures in controllable physical equipments such as points and signals may also be necessary.

3.3. The Signal specification

Several safety requirements discussed above have been specified in Signal and incorporated into the `SAFETY_SUPERVISOR` introduced in Section 2.3. Figure 7 illustrates a function-related safety requirement for points in the system; *true* values of the predicate G_P (see Section 3.2) inhibit the transition *locked* $\longrightarrow$ *unlocked* if enabled by *true* values of `REQ_UNL` and produce, instead, `REQ_UNL_O` – the actual signal sent to the equipment on the ground. Moreover, `LOCKED = V_TRUE_P` and `UNLOCKED = V_FALSE_P` ensure that the initial and final states of the points are respectively *locked* and *unlocked*.

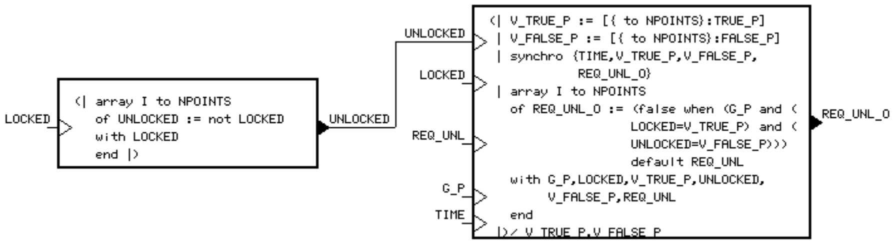

Figure 7: A function-related safety requirement for points.

The classification given in Section 3.2 is seen here as a *generic* way of defining safety requirements encountered across applications. However, the concept of a generic safety module that can be plugged into any application through an appropriate interface requires further investigation. Figure 8 shows a 'generic' abstraction of the safety requirement presented in Figure 7. The transition ensures that `Trigger_Output` has the value *true* only when `G` is *true* and the entry and exit states, belonging to some abstract type *State*, also belong to subtypes `LRISK` and `HRISK` respectively. However, the definition of a generic `G` could be complex since, in general, it depends on the global state of the system and, therefore, may involve various combinations of state variables.

A complication in the specification of failure-averting and failure-handling safety requirements in a generic way is that each requires a mechanism for ensuring that the transition is executed in a timely manner. In the case of an extended HL transition, there is the need to ensure that the system remains in the LR state after its execution, unless controlled otherwise by the safety module. This may be achieved by modifying, as nec-

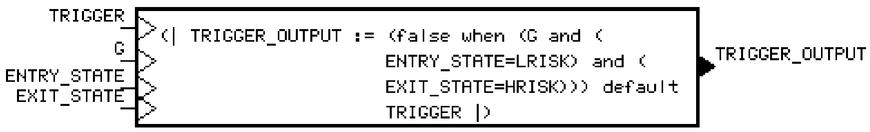

Figure 8: A generic safety requirement.

essary, the predicates associated with possible subsequent LH transitions. The resolution of these problems requires primarily a more complete specification of a safety supervisor for the railway signalling system and its validation. In this respect, the current SIGNAL model constitutes a basis for a comprehensive analysis of the system, requiring a detailed study of the low and high risk states of the system under consideration, the relevant safety requirements, and implications of change of context. The experience gained from this case study will be used in defining a generic approach to safety management. This will involve identifying categories of generic LR and HR states and providing mechanisms for their representation and manipulation, rendering LH and HL transitions completely generic, and developing an adequate interface between an application and the generic safety module.

In this context, this case study has been a valuable experience in developing a generic approach to safety management. In particular, separation of functional and safety requirements, description of such requirements in a generic manner, ways of handling physical failures in the controlled and controlling system, and specification and varification of integrated safety and functional requirements formally in a unified mathematical framework are some of the achievements. In comparison to the concept of safety kernel in [19] the safety supervisor investigated here acts as a filter on controller actions and ensures both positive and negative properties. Provision of mechanisms for containing physical faults at the level of equipment, sensors and the controller itself is another important feature.

4. Conclusion

This paper has demonstrated the application of the synchronous data flow formalism SIGNAL to the specification of functional and safety requirements of a railway signalling system separately, and subsequent validation of the integrated requirements. The case study is a critical system which is realistic in terms of both complexity and the nature of safety features. Following the usual SIGNAL approach, the specification has been developed hierarchically through a process of refinement of an abstract specification. Formal verification of process synchronizations, and other relevant temporal properties, has been performed by checking the consistency of constraints on clocks. The result is a controller which is shown to be both functional and safe. It can also be translated into executable C code by compilation, and thus be used as a simulation for exploring the functionality and safety properties further.

Another aim of this work has been to study the feasibility of a 'generic safety supervisor' with the responsibility to ensure the functionality of the application in a safe and timely manner. Central to this approach is a classification of safety requirements into several categories: function-related, failure-averting and failure-handling requirements. By viewing the pattern of behaviour as a sequence of transitions between low and high risk states, these safety requirements have been represented as inhibited, or forced and timed, transitions between the low and high risk states. A safety supervisor, incorporating a subset of these rules has been specified and validated in SIGNAL. Finally, a conceptual framework for the safety module is discussed in general terms, identifying main problems as well as future work. They concern the complete specification of the safety supervisor for railway signalling considered here for demonstrative purposes and its extension to a generic module adaptable to other applications. In this respect, the existing SIGNAL specification serves as a basis for further investigations with emphasis on their validation by formal verification of dynamic properties (uisng SIGALI) and simulation.

5. Acknowledgements

We are grateful to Paul Le Guernic in particular and the EPATR group from INRIA Rennes, France, for providing us with the SIGNAL environment and the required technical assistance, without which our work would not have been possible.

6. References

[1] CULLYER J., WONG W.: 'Application of formal methods to railway signalling - a case study', *Computing & Control Engineering Journal*, February 1993

[2] LEVESON, N. G.: Software Safety in Embedded Computer Systems, *Communications of the ACM*, February 1991, **34**, (2), pp. 34-46

[3] LEVESON, N. G.: Software Safety: Why, What and How, *Computing Surveys*, June 1987, **18**, (2)

[4] LEVESON, N. G., STOLZY, J. L.: Safety Analysis Using Petri Nets, *IEEE Trans. on Software Engineering*, 1987, **13**, (3)

[5] NISSANKE N., ROBINSON N.: 'Formal Methods in Safety Analysis'. International Conference on Computer Safety, Reliability and Security,SAFECOMP'94 (Instrument Society of America, Maggioli V.), 1994

[6] NISSANKE N.: 'Safety specification in deontic logic'. Proc. 2nd IMA Conf. Math. of Depend. Syst. (Oxford University Press), 1997.

[7] VELOUDIS S., NISSANKE N.: 'Duration Calculus in the Specification of Safety Requirements'. Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT'98, September 1998, Lyngby, Denmark

[8] HANSEN K. M.: 'Formalising railway interlocking systems'. Technical report ID/DTH KMH3/1, ProCos II, Department of Computer Science, Technical University of Denmark, DK-2000 Lyngby, Denmark, 1994

[9] BENVENISTE A., BERRY G.: 'The Synchronous Approach to Reactive and Real-time Systems', *Proceeding of the IEEE*, September 1991, **79**, (9)

[10] LE GUERNIC P., GAUTIER T., LE BORGNE M., LE MAIRE C.: 'Programming Real-time Applications with SIGNAL', *Proceedings of the IEEE*, September 1991, **79**, (9), pp. 1321-1336

[11] HAREL D., PNUELI A.: 'On the development of reactive systems', *Logics and Models of Concurrent Systems* (Springer-Verlag, New York, 1985), pp. 477-498

[12] HAREL D.: 'STATECHARTS : a visual formalism for complex systems', *Science of Computer Programming*, June 1987, **8**, (3), pp. 231-274

[13] BOUSSINOT F., DE SIMONE R.: 'The ESTEREL Language', *Proceedings of the IEEE*, **79**, (9), September 1991, pp. 1293-1304

[14] HALBWACHS N., CASPI P., RAYMOND P., PILAUD D.: 'The Synchronous Data Flow Programming Language LUSTRE', *Proceedings of the IEEE*, **79**, (9), September 1991, pp. 1305-1320

[15] MARCHAND H., RUTTEN E., SAMAAN M.: 'Specifying and Verifying a Transformer Station in SIGNAL and SIGNALGTI'. Technical report 2521, IRISA/INRIA - Rennes, France, 1995

[16] BOURNAI P., LE GUERNIC P.: 'Un environnement graphique pour le langage SIGNAL'. Technical report 2040, IRISA/INRIA - Rennes, France, 1993

[17] RUTTEN E., LE GUERNIC P.: 'Sequencing data flow tasks in SIGNAL'. Technical report 2120, IRISA/INRIA - Rennes, France, 1993

[18] ISAKSEN U., NISSANKE N.: 'Managing safety in critical systems'. Technical report, Department of Computer Science, University of Reading, United Kingdom, 1997

[19] LEVESON, N. G., SHIMEALL, T. J., STOLZY, J. L., THOMAS, J.: Design for safe software. Proc. of the American Institute for Astronautics and Aeronautics (AIAA), Space Science Meeting (Reno, Nev.), 1983, AIAA, New York