A High Availability Vital Computer for Railway Applications: Architecture & Safety Principles

Sylvain Baro

▶ To cite this version:

Sylvain Baro. A High Availability Vital Computer for Railway Applications: Architecture & Safety Principles. Embedded Real Time Software and Systems (ERTS2008), Jan 2008, Toulouse, France. hal-02269811

HAL Id: hal-02269811
https://hal.science/hal-02269811

Submitted on 23 Aug 2019

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

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A High Availability Vital Computer for Railway Applications: Architecture & Safety Principles

Sylvain Baro[1]

1: Siemens Transportation Systems, 150 avenue de la République – BP 101 – 92323 Châtillon CEDEX

Abstract: The computers used for Railway Automation have increasing their level of safety and availability, especially for providing expected answer to *Unattended Train Operation*.

To achieve these demanding requirements, relevant answers have to be addressed. In this article, we first present the architecture of our computer, based on a vital coding processing, and its embedded redundancy feature, which both allow the achievement of an efficient architecture providing the high availability requested by the above mentioned Railway Applications.

We then present the principles used in the safety design and used to bring out the safety evidences. We will specifically highlight the safety and performance issues raised by assembling two single vital computers into a redundant configuration. Nevertheless, we have to stress that in our choice of design, the redundancy is used only to address the availability goal.

Keywords: Availability, Safety, Redundancy, Computer, Railway Applications

1. Introduction

The computers used for Railway Automation have increasing needs on safety and availability. In the case of Unattended (Manless) Train Operation those requirements became unconditional; the availability goals are very high to avoid any major disruption of operation. Such operation cannot afford a total stop of traffic due to any single failure.

In addition to these high availability requirements, the complexity of functions and the system performances are increasing significantly.

To achieve these demanding requirements, an appropriate answer has to be given. The computer design cannot be anymore seen as a single computer with a redundancy, but as a pair of computers providing a vital fault tolerant system.

Our redundancy protocol based on fault tolerant principle makes two redundant computer units work as symmetrically as possible. The achievement of the consistency of the two units makes possible to switch to any computer at anytime, without taking any care, neither at the application level nor at the system level. This valuable point has the benefit of reducing the complexity of the system and the safety analyses. We can say that the switching process is seamless from the functional part of the application and from the system point of view.

In order to understand the specificities of our redundancy protocol, we first introduce the architecture of our computer, based on a vital coding processing (Sect. 2), without the redundant architecture. We then present our previous redundancy principles (Sect. 3). We finally introduce the integrated redundancy feature, which provides the high availability requested by the above mentioned Railway Applications (Sect. 4), starting from the principles and goals, then switching to the inner design. For conclusion, we address the benefits and limitations of our vital fault tolerant system.

2. Vital Computer Overview

The DIGISAFE© XME vital computer is the upgrade of the DIGISAFE© platform, used on railways system automation since 1989: SACEM systems [5], then on METEOR (Paris L14, in Unattended Train Operation) in 1998, and New-York Canarsie Line in 2005.

This vital computer is based on vital coding techniques using a single processor. These techniques guarantee that any computation which does not comply with the relevant specified software will be detected in a safe manner. In other words, either the source code is compiled, linked, uploaded, and executed as intended; or the vital computer detects the error (with a very high probability) and set itself in a safe mode.

2.1 Architecture

The DIGISAFE vital computer is the platform used in CBTC railway applications for the *onboard vital controller* and for the *wayside vital controller*. This computer is used to fulfill SIL4 sub-systems requirements [2]. The DIGISAFE© architecture has been described precisely in [4]. For the comprehension of this paper, we nevertheless provide a short description hereafter.

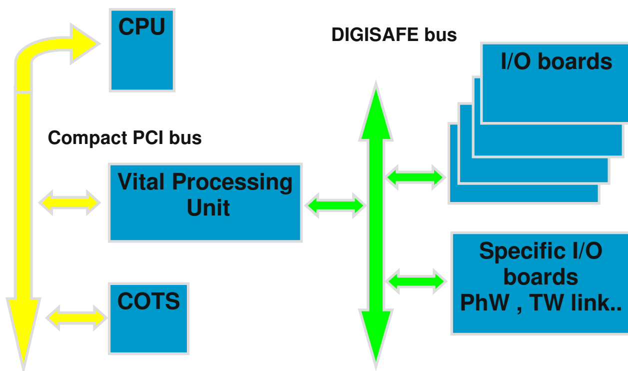

Figure 1: Architecture without redundancy

In order to fulfill this role, DIGISAFE© secures all the processing from the inputs to the outputs, including the different layers of the software architecture.

The *vital* analog wired inputs are processed by dedicated vital boards, which convert the input signal into a coded time-stamped message, which in turn is sent to the inner layers of processing. The acquisition and the coding of the inputs are both consistent with the SIL4 safety level requirements.

Cyclically, the vital application is triggered. The inputs are then used in regular computations which produce finally the outputs. In order to check that these computations are handled properly with respect to the source code, all vital data are coded. Each vital variable consists in two fields. The first field is the functional part X_f, which contains the requested value of the variable (an integer, boolean, array...) and it is processed by regular computation. The second field is the coded part X_c of the variable, it contains all the information required to secure the computations and values, it is processed by dedicated vital algorithms.

The coded part contains a static signature representative from the occurrence of the variable and of the operations used on the variable, an arithmetic part representative from X_f, and a time-stamp in order to guarantee the freshness of the variable.

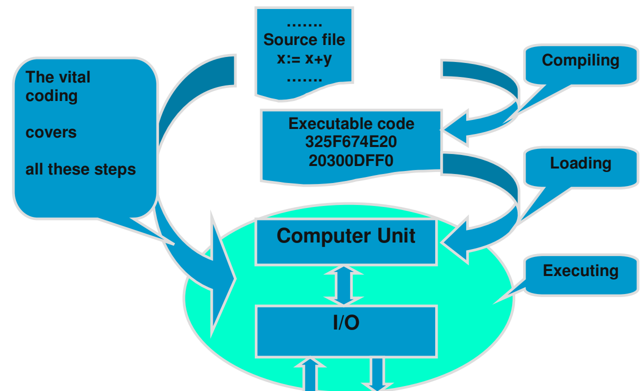

Figure 2: Logical tool- and execution-chain

Any data, message, equipment is called *off-code* when the variable (X_f,X_c) is not consistent with regard to the code, and *in-code* when it is.

To make easy the use of such coded variables, and to use them as regular variable at the application level, primitive operations on these data (*OPELs*) are defined in order to manipulate the code itself. This also brings the benefit of encapsulating the manipulation of the code in a dedicated part of the computer.

To make the check easier, a specific tracer is used. It allows propagating the code status. If one computation leads to an off-code status, the tracer remains off-code.

Vital digital messages are transmitted between equipment. These messages are transmitted through *non-vital* layer seen as black boxes, but are *packed* in a specific form, called a *vital message*. The packing function uses a specific code in order to check the integrity of the message Only vital applications are able to modify or create a message without damaging it. A damaged (*off-code*) message is detected by the receiver, thanks to the code. Given equipment are only able to pack *in-code* messages if they are themselves *in-code*.

The analog wired outputs are first worked out by the Vital Application, and then are sent to the DIGISAFE bus, in order to command through the vital output boards, the vital output signal. The vital output data is sent with its code and a *time-stamped*. The output board applies functionally the output. A vital checking is made at any time in order to verify the consistency of the actual status of the output and the expected status considered at the application level. To do so, the dynamic controller (CKD) reads vitally the code of the actual status and checks if the value complies with the expected value. If a discrepancy is detected, the outputs are de-energized (vitally guaranteed).

Dedicated sensors can be used with the DIGISAFE computer. The most common are the odometer (coded phonic wheel, or optical speed sensor) and the transponder reader. Both are vital, precise, and

involved in the process of computing the train position and speed.

Of course, apart from these vital I/O and software computations, DIGISAFE allows the handlings of non-vital streams, as well as the execution of non-vital task.

2.2 Properties

The safety properties of this architecture are numerous:

The safety proof of the computer does not rely on the digital component of the computer (processor, memory, network, etc.) but only on the protection offered by the code, which complies with the SIL4 safety levels requirements;

This safety principle guarantees that any error which may occur on any vital part of the application, would lead to an error in the vital outputs (and the messages). It will cause these outputs to be off-code. This also leads to the fine property that all the software and data which have an impact on the values of the vital outputs are required to use the OPELs. In other words, the software parts which do not use the OPELs do not need to be considered during the safety analyses on the application.

The principles used for the time-stamping of the outputs code have the property that this code can only be consistent with the values of the outputs if the outputs are applied at the proper time. This allows (together with a *vital clock*) to guarantee that the proper cycle time is enforced.

To sum up, if anything leads to an error during the processing of the cycle, the outputs will be off-code, and de-energized: one DIGISAFE computer guarantees alone the safety. This is indeed sufficient to guarantee the safety behavior of our applications.

Nevertheless, the railway applications become more and more complicated, and need more than a simple control/command calculator: the computers involved in the operation of a line cannot be reset without severely impairing the system availability of the line.

This is particularly true for *Unattended Train Operation*, when a reset of an onboard computer could paralyze the train in a tunnel, until manual recovery handled by an operator.

For this reason, redundancy is required to prevent a single failure to impact the passenger service.

3. Applicative Redundancy

Our previous architecture (DIGISAFE for the Meteor – Paris L14) already used redundancy. On this platform, redundancy is handled through two redundant computers (each one being a DIGISAFE computer), among which at most one can be active (vitally guaranteed by the hardware). Both units are able to communicate through a serial link. The outputs are merged within a hardware *OR* gate. In all the Vital Application, in *every* safety functions the following is required:

- Defining the function behavior, when the unit is active.
- Defining the function behavior, when the unit is passive.
- Transmitting the proper data to the redundant unit, in order to guarantee that the passive unit has the proper information to be able to switch to the active state.
- Analyzing in order to find if a sequence of switching over is able to lead to an unsafe behavior of the system.

This leads to the interleaving of the functional part of the application with the redundancy processing; adding complexity to the design and making more difficult the safety analyses of the functions.

After Meteor, another approach was investigated, in order to implement a completely integrated redundancy solution, relying on the same hardware, and completely seamless at the application level.

4. Integrated Redundancy

The main principle governing the design of the integrated redundancy function is the complete separation between the redundancy management and the software application. This leads to an encapsulation of the applicative software in the redundancy manager (Sec_Red), as seen on Fig. 3. All the inputs are provided to Sec_Red which is able to modify them depending on the values of these same inputs for the remote unit, and depending on its internal state (redundancy mode). Each application then computes its outputs, modifying its internal states, but this computation only happen if allowed by redundancy. Lastly, the outputs are provided to Sec_Red to decide if permissive outputs are generally allowed, and if a given output must be force to a restrictive state.

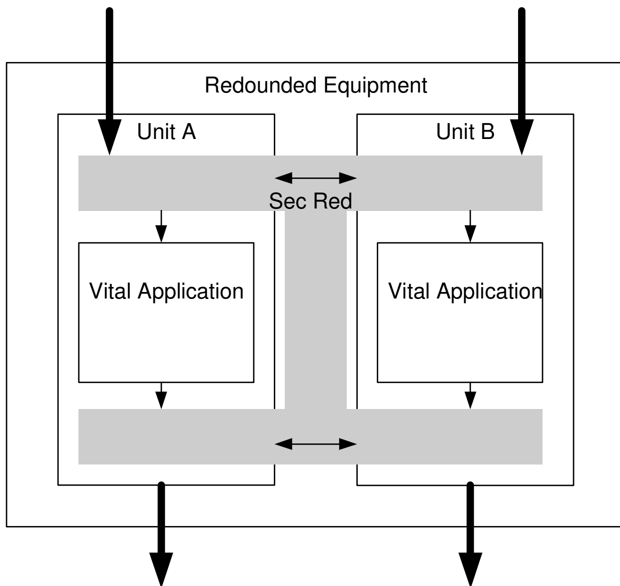

Figure 3: Logical architecture of the redundancy

4.1 Goal

The goals of the integrated redundancy protocol are:

1. to be able to maintain a service in case of any single failure in the data processing chain;
2. to guarantee that the switching between both equipment are always safe (ensuring the consistence in the behavior of both units);
3. to separate the applicative processing from the redundancy algorithms (the applicative software is never aware if it is currently active or passive);
4. to become an "off the shelf" solution, usable on all our Railway Automation projects;
5. and of course, to be safe, compliant with SIL4 requirements, and certified by an independent assessor.

4.2 Overview of the functionalities

Sec_Red is governed by two modes: isolated or redundant. While isolated, only one unit (the *Active* unit) is running the Applicative Software and allowed to apply permissive outputs. While redundant, both units (*Active* and *Passive*) are running the Applicative Software, and both are applying their wired outputs, while only the Active unit sends messages to remote equipment. In this mode, it is essential to prevent inconsistent behavior or decisions of both units. Moreover, whatever the mode is, commutation between both Active/Passive states must always be safe:

- if the current mode does not authorize commutation, the equipment must fall back into a safe restrictive mode;
- the behavior of the newly Active unit must be consistent with the "promises" made by the former Active unit.

In order to maintain safety, the Applicative Software must also be executed at each cycle (on at least one unit), with fresh inputs and with a correct applicative context. The computed outputs must be applied in the cycle in which they are computed.

4.3 Assumptions

In order not to depend on the underlying applications, some assumptions must be made on the equipment and applicative software, especially on the semantic of the Inputs/Outputs. The first assumption (linked to our DIGISAFE platform), guarantees that being redundant is not necessary for safety.

> *H1: Each unit, on its own, always behaves safely.*

The redundancy protocol must be able to alter the I/O in a safe way. In particular, it must always be safe to change the value of a binary wired input or output to *False*, and it is always safe to lose a message.

> *H2: All binary wired I/O are "oriented" within the system:* True *being a permissive value and* False *being a restrictive value.*

> *H3: Discarding or losing a message (input or output) is always safe.*

Lastly, the initialization state of the applicative software must always be safe.

> *H4: The behavior of the Applicative Software must be safe (restrictive) at initialization time.*

4.4 Hardware architecture

This redundancy protocol is also designed for a specific hardware architecture made up of two redundant units, each one being a DIGISAFE computer as described above in Sect. 2. Each of these computers are linked to its own wired inputs (which must generally be provided by the same equipment) and to the inter-equipment network (used for I/O messages).

One dedicated binary input is the Active/Passive input. This input is provided by a specific device guaranteeing (vitally) that the Active/Passive input can be Active (*i.e. True*) on at most one of the units at the same time.

The wired binary outputs of both redundant units are sent through an OR gate, which has the effect of choosing the most permissive.

These two computers are linked together by a serial link used for the exchange of inter-units messages.

Both computers are also synchronized very precisely (~ 10^{-5} s). However, this synchronization is not vitally guaranteed. It is of course necessary for the equipment to operate properly, but is not considered, as soon as safety is concerned. As for the safety part, the only property that is known is that the hardware clocks of both computers are safety clocks. The relative drift of both clocks is therefore vitally bound by a given threshold.

4.5 Synchronization and time

As mentioned in Sect. 4.4, the synchronization is very precise, but is not to be used for safety (simple failures may lead to a loss of synchronization).

As far as safety is concerned, there is therefore no hardware synchronization. Nevertheless, in order to be allowed to compare the I/O of both units, a minimal synchronization is due. In order to enforce this, a counter is managed on each unit, namely the *Vital Logic Clock* (VLC). At initialization time, the passive unit synchronizes its own *Vital Logic Clock* to the active's one. At the beginning of each cycle, a message is exchanged between both units to compare both VLC.

All messages sent between both units are also time-stamped with the VLC. Hence despite there is no synchronization, it is guaranteed that as soon as the asynchronism exceeds one cycle, both units are unable to communicate with each other. This is sufficient to trigger the *Passivation*, in order to switch back to isolated (unredounded) mode, as described below in Sect. 4.9

4.6 Vital Inputs

Two kinds of Vital Inputs must be handled by Sec_Red: wired binary inputs which can only be *True* or *False*; and Vital messages, sent by other equipment, which can contain any kind of values. These messages usually contain a vital header, with data such as an emission time-stamp, the identity of the sender and addressee, etc.

As for the wired binary inputs, Sec_Red only distinguished redounded inputs from "unredundant" inputs. These "unredundant" inputs are left as is, and should usually be used only by the redundancy. Examples of these are the Active/Passive binary input, or the current state of the unit.

On the other hand, redounded binary inputs are generally supposed to be the same for each unit. The inputs are exchanged (*via* the serial link), and each unit compare its own inputs with the remote unit's inputs.

If a discrepancy is detected, it could either be because of a small asynchronism (lower that one cycle, as seen above), either because of a failure in the input acquisition. Because the acquisition of the inputs is vitally guaranteed by the DIGISAFE hardware, the only possible failure is to see the input at *False* while it should be *True*.

Therefore in case of discrepancy, the permissive input is kept on both units, and sent to both Vital Applications. The particular case of an asynchronism together with a one cycle length restrictive input is detected, in order to prevent any restrictive state to be masked (see Fig. 3 below).

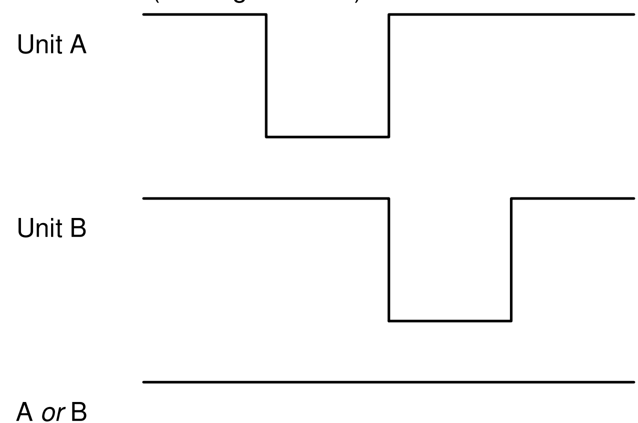

Figure 3: Masking of a restrictive input

As for the Vital Messages, they are compared between both units. If one of the units missed some messages, they are therefore sent, in order to ensure that all units will see all messages (a message is always considered as permissive data).

Of course, either for the wired binary inputs, or for the message, this process of comparing and providing the more permissive data may sometime fail (*e.g.* serial link error). In this case, either restrictive inputs or local inputs are provided to the unit. This will cause that the Vital Applications of both units will not receive the same inputs, which in turn will lead to a discrepancy in the applicative contexts of both units. At the end, this will probably cause a discrepancy in the outputs emitted by both units.

The redundancy protocol will not prevent the discrepancy to appear in the context. It will only react by controlling the inconsistent outputs, and by triggering a *Passivation* if the discrepancy lasts. This is described in Sect. 4.8 and 4.9.

It is therefore never dangerous to introduce a discrepancy in the context, as soon as the inputs provided to each unit are fresh and correct.

4.7 Application and Context

As seen above, it is not dangerous to introduce a discrepancy in the (variable) states of both units. There is nevertheless a property needed to define if the context is safe or not.

The *context* can be defined as the set of all the vital variables handled by the Vital Applications, together with their values.

A context is called *well formed* at the beginning of the cycle *n*, if one of the following conditions is true:

- either it is the initialization context of the Vital Application;
- or it was *well formed* at the beginning of the cycle $n - 1$, the Vital Application has been executed during the cycle $n - 1$, with fresh inputs;
- or the local unit is passive, the remote unit is active, the context of the remote unit at the beginning of the cycle *n* is *well formed*, and is equal to the context of the passive unit at cycle *n*.

This notion of *well formed* context is representative of an application which executes itself cycle after cycle, without ever missing to react to restrictive inputs.

This leads to three different cases. At initialization time, as soon as the unit detects it is active, it is allowed to start executing the Vital Application. It is however checked that its context is the initialization context. The active unit must continue to execute the Vital Application at each cycle, to maintain a *well formed* context.

In the other hand, the passive unit does not execute the Vital Application. But as soon as both units are ready, the active unit starts to send "context slices" to the passive unit, which copies them. This process usually takes several cycles. This process is made such that after the transmission is finished, if variable of the context changed their values, the difference is then sent to the other unit. This may also take several cycles to finish. If the performance of the serial link is too low and the context evolves too quickly, it can even become a never ending story! This way of transmitting context is close to [1].

All this transmission and copy process is not vitally guaranteed. As soon as the transmission is finished, a "photography" representative of the contexts is taken. From this point, the passive unit starts to execute the Vital Application at each cycle. Its outputs are however not applied (maybe an error occurred during the copy of the context), but are compared at each cycle to the active unit's outputs.

During a few following cycles, the "photos" are then compared, using a vital algorithm. If at the end of this comparison, the algorithm concludes that the "photos" where equal, both units are allowed to switch into redounded mode: from now, the passive unit start to apply its own outputs, and commutation is allowed and safe.

It must also be noted that all this process is managed by the redundancy software itself. The only data available to the Vital Application is a *boolean* value indicating whether or not it must be executed!

Even if both contexts are *well formed*, a discrepancy may appear between both units (usually because of a slight divergence between inputs which was not compensated by the protocol). It is therefore necessary to cover hazards at output level, as explained in the next section.

4.8 Vital Outputs

As for the vital inputs, the outputs are either binary wired outputs, or messages sent to some remote equipment. Each unit sends its outputs to the remote one, and these inputs are then processed. If the unit is not allowed to apply its outputs, the wired outputs are forced to restrictive state, and the sending of the message is forbidden (this is for example the case if the unit is passive and isolated). In the other cases, the outputs are compared in order to detect discrepancy.

Discrepancy on the Vital Output may appear for two reasons, mostly, it will be due to a divergence in the contexts. Sometime, it will also be caused by a failure in the boards responsible for the application of the vital wired outputs. In this case, the failure will be covered by the hardware OR gate which chose the permissive output between both (each unit being intrinsically safe, it is not hazardous to apply an OR in case of failure).

The other case of discrepancy appears when there is a divergence between both applicative contexts.

Three kind of hazard needs to be considered:

Hazard 1: outputs are erroneous w.r.t. the expected behavior.

Example: the inputs indicate that an event occurs, which should cause the train to brake (application of a restrictive output "No Emergency Braking"), but due to a discrepancy, the output is not commanded to restrictive.

This hazard is prevented by the following:

- the inputs are fresh and guaranteed to be safe;
- either the applicative context is well formed, or the outputs are forced to restrictive values;
- the execution of the Vital Application is guaranteed to be safe.

These three properties together guarantee that if the Emergency Braking is not applied on one of the units, it is necessarily because it does not need to be applied: therefore the error is done by the unit which commands the braking. Hence it is safe not to apply this braking.

Hazard 2: the wired outputs are correct, but the discrepancy introduces a dangerous antagonism.

Example:

The train is stopped in station, and the doors are closed. Unit A decides that the train should leave the station, therefore it *Authorizes Traction*. On the other hand, Unit B decides that the train should allow the passenger exchange, therefore it *Unlocks the Doors*.

Each of this behavior would be correct and safe on its own, but the OR gate would cause the equipment to allow both traction and door opening!

In order to avoid this case, on each unit, Sec_Red compares the output of the local unit with the output of the remote unit (receive *via* the serial link). If no discrepancy occurs, the outputs of the local unit are applied as is. If a discrepancy is detected, for each output where this discrepancy exists, the transition from restrictive to permissive is forbidden (see Fig. 4).

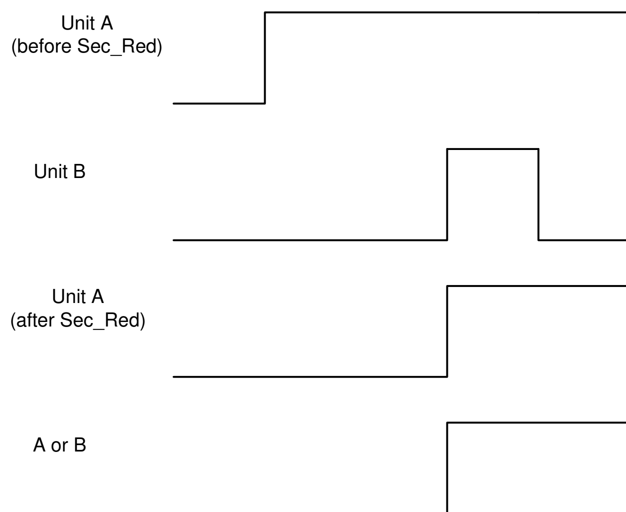

Figure 4: Inconsistent wired outputs

Hence if one the unit A was permissive and stays permissive (it was previously Authorizing Traction), while unit B switches to restrictive, the output of A (then the OR) is still permissive. But if A is not allowing to Unlock the Doors, and B did not, but now try to Unlock, the discrepancy will prevent B to switch to permissive. The Unlocking of the Doors is therefore prevented until the end of the discrepancy.

It is easy to prove that this forbidding is sufficient to avoid the case of antagonistic outputs: a dangerous discrepancy can only occur at cycle n if it exists at cycle $n - 1$, but it does not exists at cycle 0 when all the outputs are restrictive.

In this case, the consequence of the forcing to restrictive is that the train will not be able to start up, neither to open door, until both units agree, or the passive unit is *passivated* (~ 1 s).

Hazard 3: unit B does not enforce the behavior promised by unit A.

Example:

The wayside equipment sends a (not vital) message to the onboard equipment asking if it can stop before a given limit. Unit A is active, and receives the message. It decides that stopping is possible. Therefore it memorizes a mandatory stop before the limit, and sends back to the wayside controller a vital message to acknowledge the fact that the train will stop. Unit B is passive, and does not receive the message (or decides that this stop is not possible). Therefore it does not memorize the stop and does not answer to the wayside equipment.

When the wayside controller receives the message from Unit A, it is allowed *e.g.* to move a switch in downstream the acknowledged limit. But Unit A fails, a commutation is triggered, and B becomes active and isolated. The limit will not be enforced, and the train will probably run of the rails!

In this case, the safety is enforced by Sec_Red by the specific processing of the vital output messages: a message is allowed to be send only if both units are trying to send it, and both messages are equal.

In our example above, Sec_Red will therefore prevent A to send the message: even if the limit is overrun, the wayside would not have move the switch point because no permissive data would have been received.

Even if discrepancy on the outputs is covered, it is not good for the system to let them last: messages are not sent, wired outputs cannot move to permissive... the system cannot properly function while it occurs. It is therefore necessary when it occur to fall back in isolated mode when a unit alone (preferably the one that function properly) can take all the decisions. This is the purpose of the *passivation* function.

4.9 Passivation

When the discrepancy on the outputs is lasting more than a few cycles (parameter), the process of passivation is triggered. The purpose of this process is to ensure that at the end, both units will be isolated.

The passivation must occur quickly, in order to limit the number messages that will be lost during the

discrepancy (if too many messages are lost, the addressee will consider that the communication is lost, and will step in a safe and restrictive mode).

In the other hand, must also leave a sufficiently long time for a commutation to occur if needed: the commutation could take a few cycles before being effective. This is necessary if the active unit fails, and a discrepancy occur.

The passivation must also guarantee that in all case (including the loss of the serial link, several commutations...) the passive unit will be isolated before the active unit. If the active unit is isolated before the passive, it will not control its output as requested in the Sect. 4.8. The passive unit will do its own control (against restrictive outputs) thus forbidding transitions from restrictive to permissive, but the active unit will have no limitation.

The process of passivation is therefore conducted using state machines that are triggered in case of discrepancy, but also when a unit is not able to receive the inter-unit message used for the transmission of Sec_Red commands and states.

4. Conclusion

Sec_Red has been used for the first time on the on-board and wayside controllers for the resignalling into a CBTC system of the New York City Metro line L (Canarsie). It has since been used on two different kinds of equipment with a different design in both driverless metro lines (VAL) of the Roissy-Charles De Gaulle Airport in France.

This succeeded in proving that the goal to develop an "off the shelf" product to manage the redundancy in our vital systems is reached.

An issue remains on the processing of some particular inputs: the tachometers. These devices provide the on-board controller with precise (and safe) data on the kinematics of the train. Each kind of tachometer has particular properties on the timing of the data it provides. For availability purpose, it is also better to wire redundant tachometers on the redundant units, but this raises two issues:

1. The tachometers are precise, but subject to complex error models. Therefore the inputs provided by all the tachometers (at one given time) will never be equal. It is necessary for each new kind of tachometer to develop a specific *add-on* to Sec_Red, which will handle the merge of the data provided by the sensors attached to each unit.
2. In order to reach good system performances, it is unacceptable to consider that in safety, the sensors of each unit are out of synchronization of one full cycle. It is therefore necessary to provide a way to check the quality of the synchronization between both units with a far better precision that one whole cycle, and with a confidence level compliant with safety applications (SIL4).

The other main issue was that even if the redundancy protocol was seamless for the application part, at specification level, it was not as true as we hoped at implementation level. The reintegration part is costly for the CPU and our applications are bound to run in a limited cycle time. Fine optimization was therefore required in order to allow our applications to run flawlessly, these optimizations where also correlated to the way of handling the reintegration. For example, big arrays of time-out variables decremented at each cycle generate evolutions of the context that could prevent the reintegration to finish (see Sect. 4.7). It is better here to memorize the end time of the time-out, and to compare it at each cycle to the current time.

Nevertheless, these complexities are insignificant, compared to the difficulty of handling the whole redundancy processing at applicative level!

5. Acknowledgement

It is necessary to acknowledge here the work of all Siemens Transportation Systems teams, who work on the design, implementation and safety of the integrated redundancy protocol. This in particular includes David Dumont, Didier Essamé and Benoît Fumery.

6. References

[1] A. Bondavalli and al.: "*State restoration in a COTS-based N-modular architecture*", proceedings of Object-Oriented Real-Time Distributed Computing, (ISORC 98), 1998

[2] CENELEC: "*Railway Applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS*)", EN 50126, 1999.

[3] CENELEC: "*Railway Applications – Communication, signalling and processing systems – Safety related electronic for signalling*", EN 50129, 2003.

[4] P. Forin: "*Vital Coded Microprocessor: Principles and Applications for Various Transit Systems*", proceedings of IFAC-GCCT, Paris, France, 1989.

[5] C. Hennebert and G. Guilho: "*SACEM: a Fault Tolerant System for Train Speed Control*", proceedings of 23rd int. conf. on Fault-Tolerant Computing (FTCS-23), Toulouse, France, 1993.

[6] J. A. McDermid and Q. Shi: "*Secure composition of systems*", proceedings of the eighth Annual Computer Security Applications Conference, 1992

7. Glossary

CBTC: Communication Based Train Control

Context: The state of the Vital Application. Equivalent to the list of all variables handled by this application

In-code: State of a variable, message, I/O, equipment when the Vital Code is correct (no failure has been detected)

I/O: Inputs/Outputs

Non-vital: Any device or concept whose failures are innocuous as long as safety is concerned

Off-code: State of a variable, message, I/O, equipment when the Vital Code is broken (a failure has been detected)

OPEL: Primitive operations on coded data (from the French *OPération ELémentaire*)

Passivation: Sec_Red process that lead to isolated mode

Sec_Red: The redundancy manager software (from SECurity REDundancy)

SIL4: Safety Integrity Level 4, the highest level of requirements for safety applications [2]

Unit: Each redundant computer

VCP: Vital Coded Processor

VLC: Vital Logic Clock

Vital: Any device or concept whose failures may lead to an unsafe behavior of the system

Vital Application: The vital part of the software executed on the computer

Wayside: The Wayside controller (as opposed to the on-board controller) is the part of the system which tracks the train on the track (using data sent by the trains themselves) and allocates them "movement limits".