Vital software: Formal method and coded processor

Daniel Dollé[1]

1: Siemens Transportation Systems, 50 rue Barbès 92542 Montrouge Cedex France

Abstract: Siemens Transportation Systems has been developing mass transit systems for 30 years and for more than 10 years it has used the B formal method to develop and validate its safety critical software. With B, the software is derived stepwise from an abstract mathematical specification and formal proof ensures that each intermediate step is equivalent to the previous one. With the Vital Coded Processor, any run time error caused either by a compiler error or a hardware failure is detected and the unit is set in a safe state. A high level of productivity is achieved through the use of a tool that derives semi-automatically the code from the formal specification.

Keywords: automatic subways, formal specification, safety critical software, arithmetic coding

1. Introduction

1.1 The company

Siemens Transportation Systems is the international expertise centre of Siemens for automated mass transit systems. It is also the supplier in France of the products and services of the "Transportation Systems" group of Siemens AG.

1.2 Urban transit

An urban railway system is made of three main parts: the command centre, the car-borne controllers and the zone controllers.
The command centre supervises the line and regulates train traffic. It usually has no safety critical mission.
The zone controller manages a section of the line: it tracks the trains running on its section and sends them movement authorities that depend on the position of the other trains and the state of the lineside equipment (switches, signals, etc.).
The car-borne controllers manage the train's propulsion according to its mission and to the instructions received from the zone controllers. They also open and close the train doors. If platforms are equipped with screen doors, the trains and the zone controllers open and close them collaboratively.
The wayside and carborne controllers perform vital functions in software. In order to ensure the very high safety level required for these functions, Siemens Transportation Systems (STS) uses two complementary techniques:

- The B method [1] is a formal method for the specification, design and coding of software. B ensures that the source code implements the specification without any error.
- The Vital Coded Processor [2] ensures that the executables can cause no harm even in the case of hardware failures or defects in the compiler.

1.3. Vital software

The challenge of developing safety critical railway functions in software was met early on with specific technologies.

SACEM: In the '80s, the consortium [3] in charge of building the SACEM (Système d'Aide à la Conduite, à l'Exploitation et à la Maintenance) for the Paris metro authority developed the Vital Coded Processor to protect software execution against hardware and compilation errors. This technique is powerless against design error and classical validation through testing was deemed insufficient. A formal (i.e. mathematical) approach was therefore used.

- The source code (in Modula II) was enriched with pre and postconditions that were then verified semi-automatically.
- The specification of the software was rewritten in a formal language.
- The consistency of the formal specification and of the annotated code was checked manually.

This validation proved to be extremely costly because it had been done after a classical development. For the following Paris automatic subway, Météor, a formal development was required.

Météor: Opened to the public in 1998, Météor [4] is a driverless system that allows mixed traffic: trains without carborne controllers may run at the same time as equipped trains. Météor was the first application of the B method and it proved remarkably successful: all the errors were found during the development and the validation process was done efficiently without unit tests.

2. Formal development process

2.1 Organisation

The development of safety critical software would classically be organised in a development team, a test team and a safety team. The responsibility of the safety team is to ensure that the software never compromises the safety of the system. The organisation of a formal development is very similar, we simply added a team of formal method specialists in order to provide method and tool support.

Formal support team: The mission of the formal support team is to collect and disseminate best practices.

- It maintains the "B development Guide". This document is a methodological repository where we collect the experience of our formal projects.
- It provides tool and method support to the developers and ensures by means of technical reviews that the B models are provable and efficient.
- It may participate in critical parts of the development and coach new developers.

As most of our team now have experienced B developers, the role of the formal development team is becoming less crucial.

Development team: The development team carries out all the activities going from the writing of the software specification to the software target integration.

- It writes the software specifications.
- It formalises the specification in an abstract model.
- It refines the abstract model in order to obtain code that can be translated into a classical programming language.
- It proves all the proof obligations required by the B method.
- It produces the Ada code of the software. This code is either automatically translated from B implementations or manually written for the low-level input/output primitives of the application.

Test team: The test team defines and implements the tests that software must pass.

Validation team: The validation team is independent of the other teams and analyses each of the intermediate productions of the development team. Its goal is to identify early any defect that could lead to a dangerous situation.

- It validates the software specification.
- It checks that all the vital requirements of the software specification are captured in the abstract model.
- It checks the Ada code of the parts of the application that are not developed formally.
- It validates the interactive proofs.
- It specifies and implements its own tests of the vital requirements.

These activities may result in problem reports that have to be corrected by the development team.

2.2 An overview of the B language

B is a formal language invented by Jean-Raymond Abrial. Close to other formal language such as Z and VDM, B covers the full development cycle and is suitable for large scale developments. Is main characteristics are:

- B has a mathematical basis that is both powerful and easy to learn: elementary set theory and ordinary two-valued predicate logic.
- Proofs are at the heart of B: every property that is expressed in a B model must be proved.
- B gives the developer a uniform view of the formal specification and its code: the same notation is used for both.
- The B model is made of modules that provide data and operations on these data. A module is further divided into: a Machine, a Refinement and an Implementation. The machine gives the external view of the module: it shows only what is needed by users of the module. The refinement gives the internal view of the module: it specifies completely the behaviour of the module. Machines and refinements are abstract components: a data can be a set or a relation. An implementation, on the other hand, is concrete: it is directly translatable into a classical language and its data structures are very limited (integer, booleans and arrays).
- B is fully supported by a tool: the "Atelier B" [5].

2.3 Development cycle

The classical development cycle is split into the following phases: specification design, coding and test. The use of B implies some modification, they are detailed below.

Software specification: The software is specified in a text document organized along the principles of functional analysis (SADT). The aim of this document is to state clearly and rigorously what the software has to do, it uses whatever notation is best suited to that purpose: natural language, state automata or B formulas. The software specification prepares the formalisation phase, it should identify the following items:

- The *data* represent the states of the software, the input/output of the functions that specify it or its configuration.
- The *processing* specifies how a function computes its outputs from its inputs and how it modifies its internal states.
- The *properties* are logical assertions that either express safety properties or simply clarify the processing.
- The *hypotheses* are assumptions about the environment of the software.
- The *limitations* give the scope of validity of the data.

All those items will later be found in the B model.

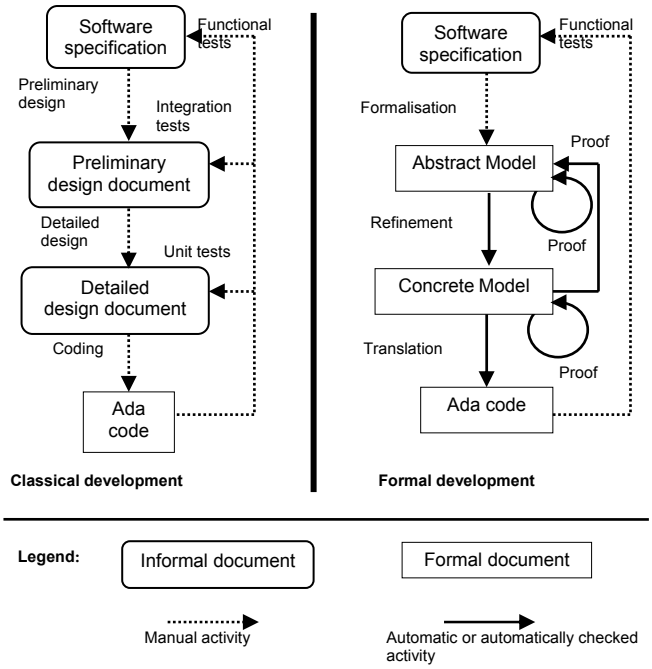

Figure 1: Comparison of the classical and formal development cycle

Formalisation: The goal of the formalisation is to express in B the content of the software specification. This phases ends when all the requirements have been captured in the *abstract model*. Ideally, it can then be refined without knowing the textual software specification.

The development of the abstract model is constrained by technical sub-goals:

- Checking the compliance of the formal model with respect to the software specification should be easy.
- The proof should be easy.
- The abstract model should not unduly constrain the coding.

The means to fulfil these sub-goals are:

- Respect the functional architecture of the software specification.
- Divide complex problems into smaller problems that can be treated separately.
- Keep the number of B implementations to a minimum. It should be noted that it has a twofold effect. Implementations are poorly readable because they cannot use the concise mathematical notation that is allowed in machines and refinements. Furthermore, since all the implementations are directly translated into code, they have to take into account efficiency constraints.

Depending on the complexity of the function to be formalised, a module can be a single machine, a machine and its refinement or a machine, its implementation and one or more machines. In all cases, the machine formalises the part of the requirements that is needed for the outside of the module. For instance: “the speed of the train is lower than a limit”.

- If the requirement is simple, it can formalised in a single machine. For instance: “the speed of the train is lower than a limit for the whole line”.
- If the requirement is more complex, a refinement is added to the machine. For instance: “the speed of the train is lower than one limit on the mainline and lower than another limit in the yard.
- If the requirement is still more complex, an implementation is added to the machine. This implementation divides the requirements in smaller requirements that are treated separately. For instance: “the speed of the train is lower than a limit that depends on the slope at the position of the train, the limit can be modified by an operator on work zones”. In this case the implementation would divide the problem into two modules that would compute the train position and handle the operator requests.

The different items of the textual specification can be found in the abstract model:

- The *data* are the variables and constants.
- The *treatments* are the operations.
- The *properties* are either the invariants of B components or the postconditions of operations.
- The *hypotheses* are either modelling choices or postconditions of the operations that interface the formal software with its environment.
- The *limitations* are the preconditions of read operations.

Refinement: The goal of the refinement phase is to produce the concrete model. It contains the abstract model as well as the implementation of all the components that were not previously implemented. The following principles apply to this phase:

- Except in the case of technical limitations, all the B components should be implemented.
- The constraints coming from coding efficiency and memory consumption should be taken into account.
- The difficulty of proof should be kept under control.

The refinement brings the abstract model down to code: the sets and relations of the model must be transformed into arrays, the non deterministic constructions of B must be transformed into classical control structure (IF and CASE). If these transformations are carried out simultaneously (each machine or refinement is refined into a single implementation) the resulting proofs can be extremely difficult. The recommended practice is to refine the control structures first and to refine the data later on.

Nowadays, the refinement is automated and the developer can concentrate on the new refinement schemes that the tool cannot yet refine and on the run time optimisations.

Proof of the B model: The B model is proved at the same time as it is developed. The goal of the activity is to demonstrate that each component is consistent and that it does not contradict the abstraction it refines. The first step of the proof is to compute all the proof obligations foreseen by the B-Book [1], it is done entirely automatically by the Atelier B. The second step is the actual demonstration of the proof obligations. The Atelier B provides an automatic prover that discharges up to 90 % of the proof obligations. The remainder is proved interactively: the developer can both drive directly the behaviour of the prover and add new proof rules. For the Météor project we added about 1200 proof rules to the internal rule base of the prover. These rules are checked by the validation team in order to ensure integrity of the proof.

Compilation: As there is no compiler for B, the implementations must be translated into a classical programming language. In our case this is done entirely automatically and the target language is a subset of Ada.

Validation of the vital software: Validating the software consists in ensuring that is execution fulfils the requirements identified in the software specification. The validation of software developed with B goes through the following steps:

- The consistency between the abstract model and the software specification is ensured by functional tests and by an analysis of the model that is carried out by the RAMS team. Ideally, the functional tests should be done early in the development and use only the abstract model. In practice they use the Ada code coming the concrete model.
- The consistency between the abstract and the concrete model is proved.
- The consistency between the B implementations and the Ada code is ensured by the use of two diversified translators. An analysis by the RAMS team guarantees that the Ada code of the machines that are not implemented in B is actually a refinement of these machines and it satisfies the requirements.
- The consistency between the Ada code and its execution is ensured by the Vital Coded Processor [2].

Some points worth noticing: As the abstract B model captures all the vital requirements of the software specification, the RAMS team can validate the software without analysing the concrete model. Similarly, all the system-level properties of the model are in its abstract part, the concrete model only contains code-level properties.

Since the goal of the tests is to detect modelling errors, the reference for the tests is the software specification and not the abstract model.

We have no unit tests, they would check that the code conforms with the detailed design and this is already done by the proof.

In the classical life-cycle, the validation is based on testing and can thus only start after the coding. In a B development it starts during the formalisation phase: the developers prove the abstract model before they start refining it.

The limits of B: There is no B construct to express explicitly real time constraints. They can nonetheless be modelled under the hypothesis that the software is periodically activated. In our case, the hardware vitally ensures a constant periodicity for software activation.

Likewise B has no construct for input/output or for file access. These functions are modelled using a machine that is implemented directly in Ada.

A further limitation is that B handles no other numbers than integers. We model real numbers as fixed points and handle the necessary scaling factors explicitly.

2.4 Safety techniques

The B method ensures that the implementations comply with the abstract model. It is also necessary to ensure that the software execution complies with the B code. Our technology cannot prevent run time errors but, since our equipments have a safe state (no traction power for a wayside equipment and stopped train for an onboard equipment) we can guarantee that the system will either function correctly or go into a safe state. We use two techniques for that purpose the Vital Coded

Processor (VCP) and the redundancy of the B/Ada translation.

The Vital Coded Processor: The Vital Coded Processor uses a probabilistic approach to detect inconsistencies between software source code and its execution. Each data of the software has two parts: 32 bits of functional information and N redundant bits. The overall safety level achieved depends on N but not on the technology used. It is, for instance, independent of the hardware reliability and no validation of the compilation tools is necessary. The redundant part is the sum of three terms:

- An arithmetic encoding of the functional value detects data alteration in memory or during their copy.
- A signature detects instructions executed in an incorrect sequence. As this signature is static, independent of the functional information, it can be pre-computed off-line.
- A time-stamp for each computation cycle ensures that the previous value of a variable cannot be used after an update.

After giving a signature to each input variable, the Signature Predetermination Tool (SPT) analyses the source code and computes the expected signatures for the output variables. These pre-computed signatures are then embedded in the executable.

A run time, the data are computed using a library of elementary operations (arithmetic, test condition computation, loops, etc.) that updates both the information and redundant parts. At the end of each computing cycle, a failsafe hardware component called the *dynamical controller* compares the actual value of the signatures with their actual value. In case of discrepancy, the dynamical controller puts the equipment in a safe state. For efficiency reasons, this comparison is not done for each variable but for a single variable that is built to have the property that it "collects" every error: as long as the computation cycle is less than 2^N operations, an error in any variable is detected with a probability $1 - 2^{-N}$. As in our implementation of the VPC 2^N is roughly equal to 10^{14}, for all practical purpose we are certain to detect any error.

The safety provided by the VPC relies on the diversity of the source code analysis done by the compiler and the SPT. We assume that there is no common failure mode because they have been developed independently (different teams and languages).

B/Ada translation: The VPC ensures that the software runs correctly with respect to its source code. It is also necessary to guarantee that the source code is correct with respect to the B implementation that it translates. We use two translators for that purpose: they have been developed fully redundantly (teams, design and programming language were different) on a common specification.

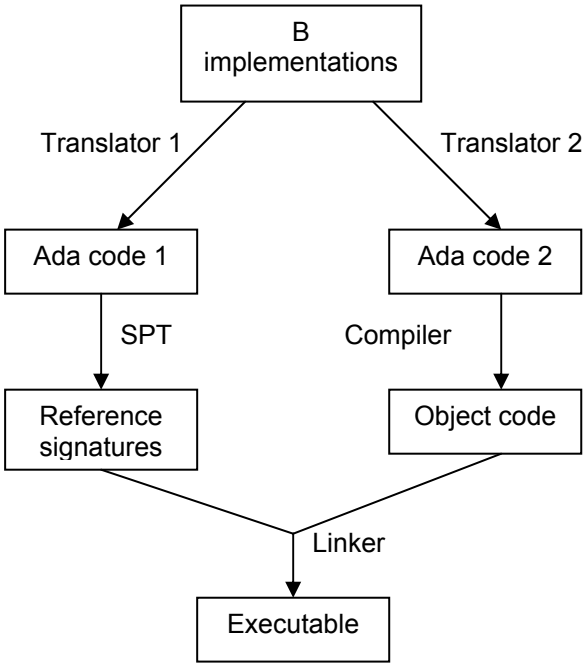

Figure 2: Translating B to Ada

As shown in figure 2, we use the VPC to compare Ada source codes: one set of source files is compiled and the other is analysed by the SPT. Any error (in the translator, the compiler, the linker or at run time) will result in a difference between the pre-computed signatures and the signatures and their run-time value.

3. Tools for the B method

3.1 Commercial tools: Atelier B

The core tool for formal development at Siemens Transportation Systems is the *Atelier B*. This product, developed and marketed by ClearSy [5], supports all the features needed to manage an industrial-scale formal development and to rigorously apply the B method.

- The model can be structured and teamwork is supported. In our case, the typical size of our applications is: five hundred files, a few hundred thousands line of B model, a few tens of thousands of proof obligations, 10 to 15 sub-models, formal development teams of up to 10 persons.
- The features available for the B method are: static checks of the B files (syntax and type checks, visibility rule checks, etc.), proof

obligation generation, automatic and interactive proof, translation of the B implementations into Ada.

3.2 Proprietary tools: EdithB

STS has developed many tools that enhance the efficiency of formal development, the most significant is an automatic refiner called EdithB.

The problem: A typical beginner's mistake is to implement simultaneously an abstract variable and the operations on this variable. This results in complex proof because abstract processing on abstract variables must be checked against concrete processing on concrete variables. An experienced B developer will proceed differently:

- Intermediate implementations are used to refine the operation one step at a time.
- Variables are refined last.

The resulting architecture looks like:

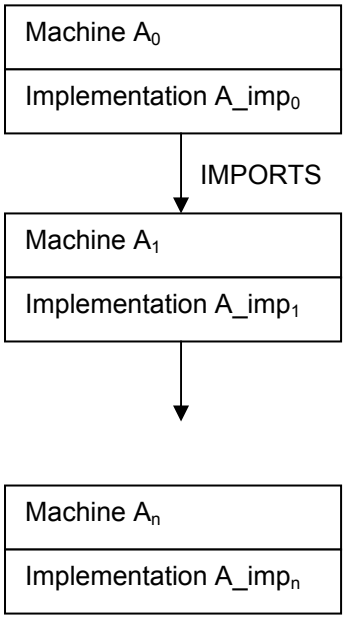

Figure 4: A typical refinement architecture

In such an architecture the variables of A_0, ..., A_n are abstract. It is only in A_imp$_n$ that the variables are implemented. Similarly, each operation Op_i of A_i is implemented in A_imp$_i$ with a sequence of operations call to operations in A_{i+1}: Op_{i1}, ..., Op_{im}. Since each of the new operations Op_{ij} only uses abstract data, it easy to prove that they refine Op_i. A drawback of this technique is that it is very repetitive: each new machine repeats piecewise information that was present in the previous machine.

Météor: One of the key results of the Météor project was the identification of a very small number (less than 10) of standard data refinement schemes. Using a small number of refinement schemes led to a great uniformity of coding style that eased the review process and reduced the number of proof rules that had to be written (a proof rule useful for a given use of the refinement scheme was likely to be reusable for another instance of this refinement scheme).

The solution: These refinement schemes were first collected in a "B Method Guide" and used during the development and review process of our models. Later, they were implemented as rewriting rules in our automatic refinement tool. The tool takes as input an abstract B component (a machine or a refinement) and produces an implementation of this component. As this implementation may import a new machine, the process is iterated until a terminal implementation is produced.

Principles: EdithB is made of two parts: a rule base and an interpreter for this rule base. The interpreter is a stable software that has not changed much since its creation in 1999. The rule base is dedicated to our applications: it can only refine the B constructs that we use. It is a growing body that has been enriched in the course of our successive formal developments. The result is that, even though we may not always be reusing code directly, we are reusing the expertise that went into its writing.

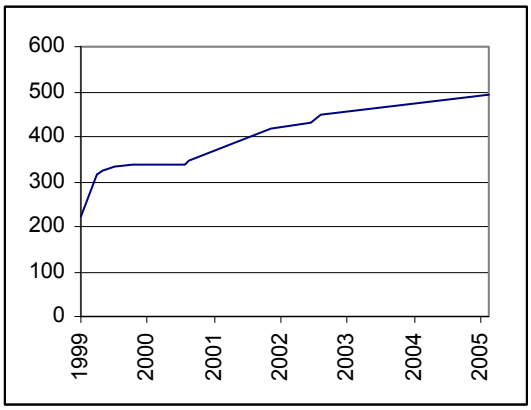

Figure 5: Number of refinement rules

An example: The rule base can be divided into two parts: rules for the refinement of variables and rules for the refinement of instructions. Their respective behaviour will be explained with an example.

- To refine an abstract variable one has to define the concrete variable that implements it and how both are related (the "gluing" invariant). A simplified example of variable refinement rule is shown below.

```
RULE set_variable IS
   ABSTRACT_VARIABLE a
   TYPE integer_set(a,n)
   INVARIANT
      a ⊂ 1..n
CONCRETE_VARIABLE
```

```
   a_r
INVARIANT
   a_r ∈ 1..n → BOOLEAN &
   a = a_r^-1[{TRUE}]
END
```

The abstract variable “a” is a set of integers, it is implemented with an array of booleans called “a_r” and the gluing is that “a” is exactly the set of indexes for which “a_r” is TRUE.

A theoretical limit of our tool is that it chooses the refinement of a variable without taking into account how this variable is used. As a consequence, it chooses the most general refinement, the one that works for every conceivable use of the variable. In the example above, if only the maximum of “a” is ever accessed, it is more efficient to refine “a” with a_max = max(a). In our experience, this limit is not significant.

- The rules that refine instructions work within a context defined by the variable refinements and by the hypotheses that are at the point of refinement. Let us suppose we wish to refine the assignment to “my_set” that is inside the IF:

```
INVARIANT
   my_set ⊂ 1..99
OPERATIONS
  my_operations =
    IF 3 ∈ my_set THEN
      my_set := my_set ∪ {my_var}
    END
```

The context available to an instruction refinement rule contains: integer_set(my_set,99) (coming from the variable refinement rule we saw previously) and 3 ∈ my_set.

- With this context the following rule would apply:

```
RULE assign_set_variable IS
WHEN integer_set(a,n)
REFINE
   a := a ∪ {b}
INTO
   local_variable_1 := b;
   IMPLEMENTATION(
      a_r(local_variable_1) := TRUE)
END
```

This rules copies b into a local variable and then updates the array that refines the set. As the second statement need not be refined further, it is marked as an implementation. The first statement is not marked, so it could be refined further. This could useful be useful if b were instantiated with a complex expression but in our case b is just “my_var”, so the resulting expression would simply be:

```
   l_1 := my_var;
   my_set_r(l_1) := TRUE;
```

Interactive refinement: The automatic refiner has an interactive mode where the context and applicable rules of each refinement step can be examined. This mode is used in two situations.

- EdithB can fail to find any rule matching the current context.
- The refined code is incorrect or not efficient enough.

In both cases, the developer debugs the refinement in interactive mode and writes the rules that give the desired refinement. These rules can then be stored for use on the current B machine or, more rarely, they can be added to the rule base if they are deemed reusable.

Validation of EdithB: Since the B components it produces are proved, no safety validation is needed for the refiner.

EdithB in use: When we built EdithB in 1999, we compared its output to the manually written code of a previous formal project. There was a small increase in code size but no significant effect on computation time. The expected savings in development time led us to generalise it for all of our formal software since that time. It is now a fundamental part of our software development process. The following diagram shows the proportion of automatically generated B code for a project that is currently nearing completion (the scale is in kilo-lines of uncommented B).

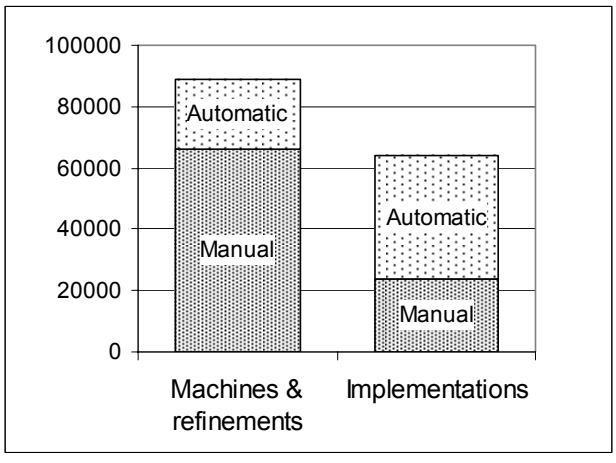

Figure 6: Canarsie Line carborne controller

Since only the implementations are translated into Ada code, around 2/3 of this provably correct application is produced automatically.

4. Conclusion

This paper has given a short overview of the development of safety critical software at STS. The technical heart of our process is the B method and the Vital Coded Processor. A highly automated development process defined around B allows us to build error-free source code in a cost efficient way. The VCP ensures that this source code runs safely on the computer platform without any safety requirement on the CPU. We can thus guarantee rigorously that no defect is introduced between the formal specification of the software and its execution. The whole process is the result of two decades of progressive improvements. It is fully consistent with the requirement for vital functions inside an industrial/product organisation.

7. References

[1] Abrial J.-R.: "*The B-Book*", Cambridge University Press, 1996.

[2] Forin P., "*Vital Coded Microprocessor : Principles and Application for various Transit Systems*", Proc. IFAC-GCCT, Paris, France, 1989.

[3] Hennebert C., Guiho G., "*SACEM : a fault tolerant system for train speed control*", 23rd Int. Conf. on Fault-Tolerant Computing (FTCS-23), Toulouse, France, 1993.

[4] Behm P., Benoît P., Faivre A., Meynadier J.-M., "*Météor : a Successful Application of B in a Large Project*", FM'99, Toulouse, France, 1999.

[5] http://www.clearsy.com

8. Glossary

CBTC Communications Based Train Control

EdithB Automatic refinement tool

RAMS Reliability Availability Maintainability and Safety

STS Siemens Transportation Systems

VCP Vital Coded Processor