Formal development of software in railways safety critical systems

B. Dehbonei & F. Mejia
GEC-ALSTHOM, 33, rue des Bateliers, 93400, Saint-Ouen, France

1 Introduction

Software is increasingly involved in the new generation of railways signalling systems. In systems such as interlocking, train routing devices or automatic train protection, electronic or electromechanical devices that previously provided safety critical functions are being replaced by computers. While safety procedures for developing critical electronic and electromechanical systems are fully mastered, no similar procedures are available for computerized modules. SACEM was the first french railways signalling system where software played a major role in safety critical functions. SACEM is an automatic train protection used in the Paris RER line A. Its purpose is to allow the train interstation interval to be decreased from 2.5 minutes down to 2 minutes without lose of passegers safety. The validation phase of SACEM consisted of a formal proof of the code using C.A.R. Hoare's pre and post-assertions technique. The certification authorities asked for an extra validation process consisting of the verification of the post-assertions. At that point, the B method has been used in order to give an *a posteriori* formal specification of the safety-critical modules. The post-assertions were then produced from the formal text. The post-assertions were more adequate and more complete. Since this experience was satisfactory, it has been decided that, for forthcoming projects, the B formal method should be used in the development of the safety-critical functions.
The B method has been devised by J.R.Abrial [1] who formerly initiated the design of the formal method Z at Oxford University [2]. Z, a formal framework based on set theory, is mostly dedicated to the description of the abstract behavior of a system. The matter of how a software is implemented was the main concern of Abrial when designing B. In fact, B is so that not only an abstract definition of the system can be given but also the way the system is actually implemented is described.

The paper is organized as follows. Section 1 presents the safety critical development life cycle. Section 2 investigates the proof framework. Section 3 mention the role that tests play in the formal development. Section 4 presents the applications developed using B. Finally Section 5 is about the tools that support the B method.

2 Safety-Critical Software Development Life Cycle

A software development with B fits into the following framework. The first step consists of a description where only the most important properties of the system are outlined. Then, this description is progressively transformed into a form which is closer to the implementation. This transformation process is called *refinement*. Each refinement level is proved to be mathematically consistent with its upper level. The last level of refinement albeit still a mathematical description is so close to the programming style that it can be automatically translated into an imperative program in languages such as Ada or C.

The life cycle we adopted for our formal development is an adapted version of the V life cycle. Recall that in the classical V life cycle, the specification is coupled with the functional validation. Then come together the design and the corresponding integration tests and so are the coding and the unit tests. The development (specification, desing and coding) is followed by the validation (unit testing, integration testing and validation testing). In the formal development, the same philosophy is retained but with some modifications described below.

2.1 Functional Specification

The first step in the development of safety-critical systems is to analyze what the system is supposed to do. The structured analysis is a semi-formal approach which is well suited for such analysis. Structured analysis is used for the software specification phase. The output of this phase is a paper work that describes the functions of the system as well as the flow of data among these functions. The structured analysis permits also the functional partitioning of the system i.e. the major functional modules of the system are identified. Simultaneously, the functional validation scenarios are written by the validation team. It is worth noting that the functional analysis and the validation scenarios should comply with each other.

A crucial task in this phase is the *safety analysis*. It investigates all the configurations the system may have and isolates the system states for which a failure can happen. These states may be reformulated in terms of relationships among the variables of the system.

2.2 Preliminary Design

Like in the standard preliminary design, the architecture of the software must be given. Partitioning is achieved through modularity for which B provides a sophisticated framework. Each B module (called *abstract machine*) is composed of a declarative part, an execution part and composition clauses.

The declarative part contains the definition of the data structures and the variables manipulated by the procedures of the module. It also includes properties (called *invariant*) related to the module variables.

The invariant is the key of correctness of a development. It expresses the properties that must hold in all the states of the system. It should reflect all the constraints that the system must fulfil. If a state violates the invariant, this state may be the source of a potential system failure. While some invariants are easy to determine, others that are directly linked to the safety properties of the system, cannot be stated without a preliminary safety analysis. We call them *safety invariants*. They reflect the safety properties of the system. In a mandatory preliminary phase, our safety team outlines the safety constraints with respect to the informal specification of the system. The design team reformulates these properties using the mathematical language provided by B. For instance, consider an automatic train protection system where the speed instructions that must be respected, come from the track equipments. One of the safety requirements stipulates that as soon as the actual speed goes beyond the current speed instruction, the emergency brake must be activated. If the actual speed is represented $current\_speed$, the speed instruction $maximum\_allowed\_speed$ and the activation of the emergency brake by the boolean $emergency\_brake$ then the safety property is written

$$current\_speed > maximum\_allowed\_speed \Rightarrow emergency\_brake = TRUE$$

The procedures of the execution part (called *operations*) can be called by other modules. Procedures are considered to be *services* offered to other modules. Composition clauses define the way modules call each other. These paradigms are supported in other modular languages. In B, the novelty comes from the modularity of proofs. In fact, modules may be modified, developed and proved separately as long as their interfaces remain unchanged.

The functional analysis usually drives the determination of the main modules in the preliminary design. Furthermore, the flow of data among the functions identified in the functional analysis phase, defines the services that modules must offer and the way modules call each other .

Consequently, the main purpose of the preliminary design is to provide the architecture of the software through the definition of modules and their services. Besides this, in the formal development, the preliminary design provides a mathematical description of the services of each module.

Each service is defined in terms of the effects it has on the variables of the module, but, usually no information about the way the service is actually implemented is given. The implementation issues are treated in the detailed design phase.

2.3 Detailed Design

In the formal development, the detailed design is synonymous with the *refinement process.* As mentioned above, the preliminary design does not deal with the implementation of the services that modules offer. The modules are progressively transformed in such a way that they get closer to an executable form. This stepwise transformation is called the *refinement process.*

There are two major refinement techniques:

- Data refinement: Mathematical objects may not be directly implemented using standard programming data structures. Consider a service of a module that manipulates a set. Common programming languages do not provide data structures that handle sets. The *data refinemeni* concretizes a data structure and transforms the operations performed on it. In the case of a set, a data refinement may consist in transforming sets into linked lists. The concretized operation of adding an object to a set inserts the object into the linked list only if the object does not belong to the list.

- Algorithm refinement: Since in the abstract levels of design, we are not concerned with how precisely a service is implemented, we may use non deterministic mathematical constructs. For instance, if a service performs two actions A_1 and A_2, we may voluntarily omit the order in which the two actions are performed. The *algorithm refinement*'s major role is to progressively eliminate non determinism. It may also enhance the description of the service by giving more precisions about the implementation algorithm.

The detailed design finally leads to modules that contain programmable data structures and executable algorithms. Despite their mathematical notation, these modules are so close to programming languages that they may be automatically translated into executable modules.

Some general translation rules are respected. Neither dynamic allocation nor pointer manipulation is supported. Recall that such programming paradigms, albeit practical in every-day programming, are not recommended (and sometimes prohibited) in safety critical software production. Floating point numbers are not supported. Ada is the programming language generally used in our projects. B modules are translated into Ada packages and B operations in Ada procedures.

3 Proofs

One may argue that most of the features we mentioned are supported by many modern programming languages. Actually, the main difference is the proof scheme. In fact, preliminary and detailed designs are to be proved according to the invariants they define. A question arises: What a proof stands for? It is the action of verifying (manually or mechanically) conditions (or *assertions* in the mathematics terminology). Like in mathematics, these conditions are first order predicates. In B, they care called *proof obligations*. We say that a module service is *proved* if and only if all its associated proof obligations are mathematically verified. Abrial's work defines how to express the proof obligations from the content of B modules. The proof obligations are automatically generated by tools that support the B method.

Two types of proof exist :

- Preliminary design proof: The proof checks the consistency of a module service i.e. if the invariant holds then it must also hold once the service is called. We say that the service *preserves* the invariant. All the services must be checked through this procedure.

- Detailed design proof: In the detailed design, modules are refined and their services are transformed in order to be closer to the implementation. The correctness proof of a service consists in checking that its transformed version is compliant with its original form. Again the compliance is checked with respect to the invariant of the module.

4 Tests

Software testing is usually synonymous with long and numerous executions, very time-consuming data collection and verification. A recent study confirmed that quantifying the reliability engineering of ultra-reliable software is almost infeasible [3]. However, in many cases, testing is unavoidable. For instance, the functional verification is meaningful only if scenarios that measures the compliance of the software with the customer requirements are provided. In some other cases, testing may be provided. Despite of intensive research work in the field of software testing, unit testing as well as integration testing remains a painful phase in the software development life cycle. B partially alleviates this problem. In fact, proof obligations cover the entire control flow graph of the procedures associated with the modules services. The first-order logic used in B permits the expression of universally quantified properties. Thus, properties are checked for all possible values of variables. The unit testing is no longer necessary since all the services of all the modules are inspected and verified through the proof. Furthermore, the

test coverage precentile obtained by performing proofs is 100%. Integration tests are also dramatically reduced. The integration of the modules written in B is again guaranteed by proofs. If a module requests the services of another module, the dedicated proof obligations check that the services are called within the interfaces that the modules should respect. We may only make sure that the interfaces among the B modules and other modules are respected. However, as mentioned above, functional tests remain necessary.

5 Applications

Our systems follow a well established system architecture. The hardware communicates with a small real-time monitor called the *operating system kernel.* This kernel bufferizes the raw data coming from the hardware. It activates periodically the *application software.* The latter is the "intelligent" part of the system. It processes the data coming from the operating system kernel and decides whether the outputs of the system must be activated or not. The application software which is sequential and fully deterministic, is developed in B. All other software parts are developed using standard techniques. Since all our applications fit into these architectural constraints, we cannot confirm how well suited B would be for the development of other types of systems (parallel, ..).

Three applications have fully been developed using B: Two automatic train protection (ATP) systems as well as an interlocking.

- The first ATP called CTDC manipulates speed instructions coming from the track equipment. The CTDC main purpose is to permanently check that the actual speed calculated from the measurement of the position is adequate with the speed instructions and the piloting mode of the train. The resulting code is about 3000 lines.

- The second ATP called KVB-KVS guarantees the respect of the security distances and the traffic signals. It is dedicated mostly to main lines. Some informations about the occupancy of the forthcoming track circuits are transmitted via beacons along the track. The ATP computes the energy of the train and the emergency braking curve that protects the train regarding the next speed limits or stop points. The emergency brake is activated if the current speed exceeds the emergency-braking curve. The final version of this software was about 19000 lines of Ada and 80 modules were required.

- The interlocking LST controls the status of the switches as well as the occupancy of track circuits in order to guarantee that a safe route can be set for a train. A supervision system asks LST to set some switches in the approriate positions in order to set up a train route. LST checks

the status of the switches, transmits commands to switches positioning engines and eventually verifies whether the desired route is actually set up. The size of the produced code is about 3000 lines of Ada.

6 Tools

Our very first experience with B (the SACEM project) has shown that without appropriate tools, no industrial utilization of B can even be envisaged. For long time, no performant tools were available on the market. We had to develop ourself a minimal environment for supporting our industry-sized B applications. The result was a programming environment for B that provides the following tools:

- Type Checker: Detects the errors related to the types of variables and interaction of modules. Without such a tool, these errors would have been trapped during the proof, but the type checking pinpoints them earlier.
- Proof Obligation Generator: Produces all the proof obligations associated to the services of a module.
- Automatic Prover: Discharges as far as possible the proof obligations. Any unprovable proof obligation may be considered as the sign of a potential error.
- Interactive Prover: Permits a thorough investigation of proof obligations.
- Ada Translator: Translates the last level of refinement into an Ada program.
- Latexer : Translates the ascii form of a B program into a mathematics notation text in the LaTeX format.
- Cross Referencer: Shows where the identifiers are defined or used.
- User Interface : Motif based interface that provides user-friendly manipulations of the tools.

References

[1] Abrial, J.,R., *The B Method*, book to appear, 1994

[2] Spivey, J.M., *Understanding Z: A Specification Language and its Formal Semantics*, Cambridge University Press, 1988.

[3] Butler, R.,W., Finelli, G.,B., The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software, *IEEE Transactions on Software Engineering*, 1993,**19**, number 1, 2-12