Using B as a High Level Programming Language in an Industrial Project: Roissy VAL

ClearSy and Siemens Transportation Systems

Abstract. In this article we would like to go back on B used to design software, by presenting the industrial process established through years by Siemens Transportation Systems on a real project: the VAL shuttle for Roissy Charles de Gaulle airport. In this project, the logical core of an equipment located along the tracks and driving the shuttles is designed with B.
By confronting this B software development, with the historical context, we show that B can be used as a high-level programming language offering the feature of proving properties. We show how this process is used to build, by construction, a large size software with very few design errors ever since its first release, and for a predefined cost.

1 Introduction

Historically, the B Method was introduced in the late 80s' to design correctly safe software (see [BBook96]). A wider scope use of B appeared in the mid 90s', called *Event B*, to analyze, study and specify, not only software, but also whole systems (see [Abr96]). This article presents the results of using B in an industrial context, to produce the safety critical software WCU[1] for the Roissy VAL[2] system. The whole system is developed by Siemens Transportation Systems. The B development of the WCU, has been subcontracted to ClearSy, who applied Siemens B Method to produce software. This method has been first established for the development of Paris underground metro Line 14, called “Météor”, and has been enhanced ever since (see [Behm99]). It consists in using B as a high-level programming language, including the feature of proving properties.

Controlling the development of safety critical software is a main concern for Siemens. They have to guatantee that the software complies to its requirements, and especially its safety requirements. They also need to control development costs and delays, especially for the development of a large project, where the size factor is a key issue.

To achieve this goal, Siemens designed a process for using B efficiently to build a correct piece of software by construction. That means that the first software release has already very few design errors. In this process Unit Tests are not performed, since more effort is directed to the early specification phases, to build directly correct software. The next development steps remain basically the same: the software produced with B is integrated with the rest of the software and goes through host tests (simulation tests on a workstation). Then software is integrated in the hardware of the WCU and goes through target tests, and so on.

This article focuses on this process.

- Section 2 presents the Roissy VAL system.
- Section 3 introduces the principles of the B Method. This method is decomposed into two phases: formalizing detailed specification documents into a B *Abstract Model*, and then implementing this model into a *Concrete Model*.
- Section 4 details the *Abstract Model*.
- Section 5 details the *Concrete Model*.
- Section 6 analyzes the project statistics.
- Section 7 presents the maintenance of such a project.
- Section 8 concludes.

[1] Wayside Control Unit
[2] Véhicule Automatique Léger (Light Automated Train)

2 Roissy VAL Shuttle Presentation

A new VAL shuttle system is being developed by Siemens Transportation Systems to equip Roissy Charles de Gaulle airport for ADP (Aéroports De Paris). The first line, due in June 2006, will connect Roissy terminal 1 to Roissy terminal 2, through Roissy Pole and two car parks. The VAL system is a driverless light train. Line 1, as shown in figure 1, is made up of 5 sections from CDG-1 to CDG-2, plus 2 technical sections to park and maintain trains. The line has two tracks. VAL trains usually drive on the right hand side track, however they may drive on any direction and change direction at any time.

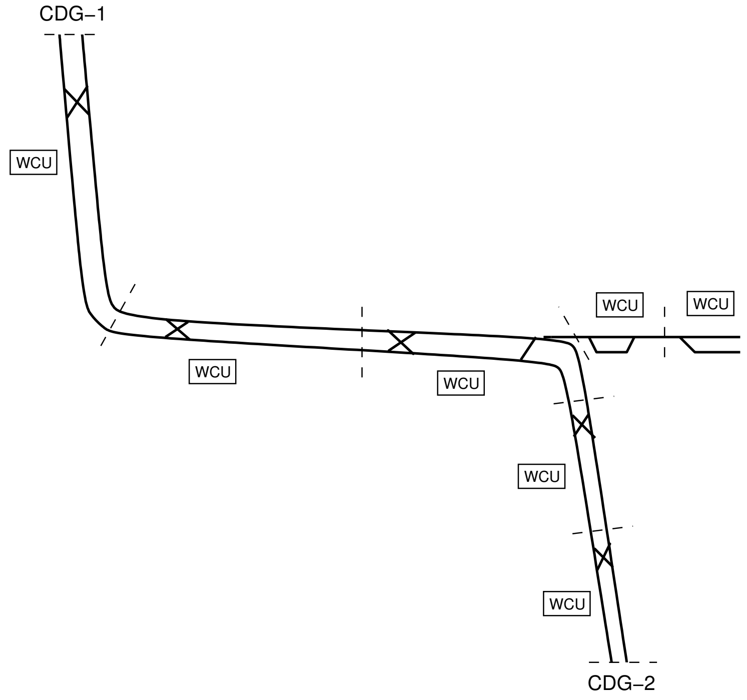

Fig. 1. Roissy VAL - Line 1

The Roissy VAL system is based on the VAL system of Chicago O'Hare airport. This system differs from the other VAL metros, like Orly-VAL (see [Dol03]). It offers higher functionalities thanks to a digital equipment called Wayside Control Unit (WCU). A WCU drives trains on a line section. Actually, a section has two redundant WCUs for availability issue. All the WCUs are linked by a redundant Ethernet network, which also connects the Control Center. The Control Center manages traffic on the line by sending route orders to the WCUs. A WCU receives these orders and then safely commands and controls automatic trains on its section.

As shuttles are light trains, they are commanded by discrete speed programs. Every speed program is physically associated to an electric current loop located on the track and covering some continuous part of the tracks. When it is powered, a speed program gives at any point of the loop a fixed speed order. Several speed programs may coexist at a certain point of the track. Examples of speed programs are cruise speed, low speed, departure speed and arrival speed. WCUs drive trains by selecting and powering the right loops. When the on-board train system detects a speed order, it adjusts the train speed according to the order.

Before sending orders to trains, WCUs should be able to localize safely trains on the line. To do so, every section is decomposed into fixed blocks, which give the localization granularity. With the help of different sensors, WCUs should safely establish whether or not a train occupies a block.

Those logical treatments are performed by a piece of software. The software is split into low-level modules that take care of input/output, task scheduling, as described into [Behm99], and into high-level modules, which handle the logical and functional treatments. The core of high-level modules, which contains Safety Critical Software, is called WCU-SCS[3]. It is designed in B and then translated into Digisafe®-ADA. The rest of the software is directly designed in Digisafe®-ADA. The Digisafe® technique intends to insure safe runtime with only one processor and replaces architectures with redundant processors. It is complementary with the B Method, which avoids design software errors.

All the WCUs does not share the exact same ADA code, as configuration data are specific to each WCU. However, the core part designed in B is the same for all WCUs.

3 Principles of the B Method Used

We shall now introduce the principles of the B Method used in the project. This B Method is part of Siemens software development process. It describes step by step how to use B to build a piece of software. The whole process is made up of guidelines, of B *generic* elements (which should be instantiated at each use in the project) and of tools also developed by Siemens.

The starting point of a project is low-level software specification documents, written in natural language and possibly using any formalism to help describing the system.

This method suits software mainly based on a discrete logic description, where basic types are Booleans, finite sets, integers or integers regarded as decimal numbers. It does not suit software based on continuous calculus or based on floating point numbers that cannot be regarded as decimal numbers.

The development process is split into two phases called *Abstract Model* and *Concrete Model*. These phases are named after the part of the B model they produce.

During the first phase, every functional piece of information or every requirement from the informal software specifications should be formalized into the Abstract Model. To make sure that the Abstract Model indeed matches its formal specification, we use inspections. A key feature of this Method is to offer the possibility of strengthening the abstract model by proving that some properties are established on the whole model.

Abstract Model data have abstract types since they are based on sets, relations and functions of scalars. Abstract data types make the Abstract Model compact and close from informal specifications. However, these data and their treatments cannot be directly implemented.

The goal of the second phase is to build the Concrete Model starting from the not implementable parts of the Abstract Model. This task is completely systematic and does not require any knowledge on the informal specifications. We have to prove that the Concrete Model is a correct implementation of the abstract model.

These two B development phases are now detailed in the next two sections.

4 The Abstract Model Phase

4.1 Informal Specifications

Informal software specification documents come as an output of the system development phase, performed by Siemens: system analysis. During system analysis, choices are made to design the system, so that it

[3] Wayside Control Unit - Safety Critical Software

should meet its functional requirements, and more important, that it should meet them safely. System design produces an equipment architecture broken down into hardware and software specification documents. Software is also broken down into a safe part, where safety is concerned, and a part where it is not. In the case of the WCU equipment, the safe software part commands and controls driverless trains on a given line section.

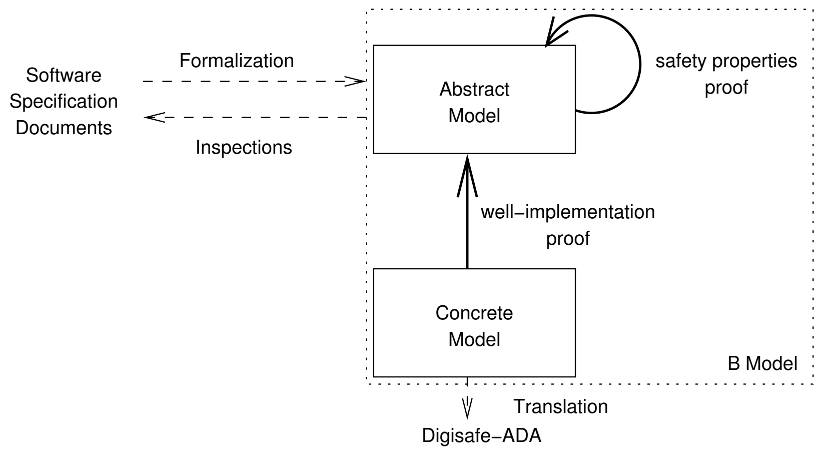

Fig. 2. Abstract and Concrete Models

The Safety Critical Software part (called WCU-SCS) is designed in B and will produce as an output Digisafe®-Ada code. In order to ensure safety for the whole system, detailed choices have to be made for the safe software during this early phase. Actually the safe software requirements produced at the end of system analysis are very detailed. They describe a functional breakdown of the software into elementary functions with data flow charts. Most elementary functions are precisely specified in terms of input, output and pseudo-code to specify output computation. Sequencing of elementary functions is also specified.

In the case of the Roissy WCU-SCS project, informal specifications come as a collection of various, more or less old, documents. Three documents were written more than ten years ago for the Chicago VAL system, before B was used at Siemens. They describe the specification of the three main modules of the system (Block Logic, Route Logic, Mode Logic). They use functional breakdown, data flow charts and treatments are specified by pseudo-code. The top document is more recent. It was written just before the beginning of the B development. It uses a functional breakdown. It describes data directly in B and makes references to treatments covered by the three main modules documents. It also defines new treatments with the same kind of pseudo-code than the other documents.

We can point here, that having a specification document as close as possible from the B model is a key issue. Since no formal proof can be done between the informal specifications and the Abstract B Model, the closest the two are, the less formalization errors or misunderstanding of informal specifications are likely to appear.

In this project data were already described in B, which minimize the risk of errors when specifying data into the Abstract Model. Siemens made all the abstract data modelling choices. Sometimes choices are just mathematical or methodological choices, for example, a relation $A \leftrightarrow B$ can be used instead of a function of sets $A \nrightarrow \mathbb{P}(B)$. But most of the time choices come from knowledge of the system. For example, to model a relation between sets A and B, should we use $A \leftrightarrow B$, $A \nrightarrow B$ or $A \rightarrow B$? It depends on the system properties.

Initializations and sequencing substitutions, which are quite simple, were also specified in B. Other operations were described by pseudo-code or in B. As they may lead to long B operations, their formalization requires to pay extra attention, to avoid any possible pitfall.

Formalizing software specifications into an abstract model is error prone since this cannot be covered by a formal proof. Informal specification are by nature liable to be incomplete, ambiguous and open to different understanding, especially when they are part of a 200-page document that has to be written in a reasonable time. Although they have flaws, informal specifications remain the best way to communicate specification. However, to handle this issue, a question/answer database on the software specification has been used to keep a trace of questions asked by ClearSy and answers provided by Siemens. This activity has been a key issue for ClearSy to understand every detail of the requirements. Sometimes questions led to explanations from Siemens and sometimes they led to precisions or adjustments of the specification documents. We can note here that updating regularly specification documents helped building a B model consistent with its informal specifications.

In order to check B models against the natural language specifications, we used inspections. Every B Language element of the model is read by a member of the modelling team who did not write it. Inspections should be as precise as possible. Every B symbol of the model should be checked. The reader has to be convinced that the related informal specification is correctly and completely formalized. Traceability between B models and specifications is achieved by giving explicit references, inside the quality-assurance comment at the beginning of each abstract operation. When a question arises during inspection, the B model may be changed, or the question might be forwarded to the project question/answer database.

4.2 Abstract Data

Data used to build the Abstract Model are abstract data. We are now going to detail those abstract data.

Basic Types The basic types used are: Boolean (the BOOL predefined set), enumerated sets or deferred sets (declared in the SETS clause) and implementable integers (predefined B sets INT and NAT).

The Boolean set is used every time a data has two possible values.

An enumerated set is used when a data has strictly more than two possible values and when we need to name explicitly all the values. For example, a section, managed by a wayside control unit, is divided into fixed blocks. A block can be a normal block, a switch block or a station block. So we define:

SETS
$t\_block\_type = \{c\_normal\_block, c\_switch\_block, c\_station\_block\}$

A deferred set is used when a variable can take a finite number of values and we do not want to name explicitly the values in order to be generic. Most data of the project are typed with deferred sets. For example, all the blocks accessed by a WCU are gathered into the type t_block. In the VAL system, beam sensors are used to detect the presence of a train at a certain point of a track. A train is detected when it cuts the beam. All the beam sensors are gathered into the type t_beam_sensor. So we define:

SETS
$t\_block;$
t_beam_sensor

The only integer variables needed for the abstract model are delays. We define a concrete constant t_time, as a renaming of NAT to declare delays regarded as decimal numbers. They express in milliseconds the time remaining before the delay expires. We define:

CONCRETE_CONSTANTS
t_time
PROPERTIES
$t\_time = \mathrm{NAT}$

Those types bring strong typing, just like in a programming language such as ADA. This makes an efficient use of the type checking constraints of the B language, since, for example, wherever a block is expected, we cannot use anything else instead, like a beam sensor.

Those scalar data types are either used alone or they are combined with type constructors as shown in the following examples.

Subsets of Scalar Types This type constructor was the most commonly used to type abstract data. For example, blocks accessed by a WCU are either occupied by a train or free. This information should be safely computed by the functional module "Block Logic". It is formalized by the abstract variable $occupied\_blocks$, which is a subset of t_block.

$$occupied\_blocks \subseteq t\_block$$

A block belongs to $occupied\_blocks$ if and only if it is considered to be occupied. We can point here, that another modelling choice could have been to use a total function from t_block to BOOL, associating to a block the value TRUE when the block is occupied and FALSE when it is not.

$$occupied\_blocks \in t\_block \rightarrow \mathrm{BOOL}$$

Actually this second choice suits less the proposed B Method, since it is more an encoding of the former (i.e., the use of a set characteristic function instead of the set directly). In this case using a subset of blocks is more abstract. Expressions, predicates and substitutions concerning this set are also more abstract and closest from informal specifications.

We also need to formalize block sensors, which are sensors detecting a train inside a block. We could have used a special deferred set to type the $occupied\_block\_sensors$ abstract variable. However, as there is a strict mapping from one block to one block sensor, we typed it as we did for $occupied\_blocks$.

$$occupied\_block\_sensors \subseteq t\_block$$

Relations and Functions Let t_a, t_b and t_c be scalar types. We also use the following relations, partial functions and total functions to type abstract data.

$$t\_a \leftrightarrow t\_b$$
$$t\_a \nrightarrow t\_b$$
$$t\_a \rightarrow t\_b$$
$$t\_a \rightarrow \mathbb{P}(t\_b)$$
$$t\_a \rightarrow (t\_b \nrightarrow t\_c)$$

For example, the abstract constant $ctx\_next\_block\_up$ associates to a block its next upward block. A block has at most one next block located in the upward direction. A terminal block in the upward direction has no next upward block. So $next\_block\_up$ is a partial function from t_block to t_block.

$ctx\_next\_block\_up \in t\_block \nrightarrow t\_block$

Sequences We use sequences of a scalar type when we need to formalize a sequence of elements. For example, the abstract constant $ctx\_block\_occ\_aut\_up$ gives in what order blocks have to be treated to compute block occupancy authorization in the upward direction. This abstract constant is formalized as a sequence of blocks.

$ctx\_block\_occ\_aut\_up \in seq(t\_block)$

Read Operations The data types described above are used to type abstract data. However, in the end, concrete code should be produced, so in order to interface abstract data with concrete code, read operations are provided with abstract data. For example, to read the abstract variable $occupied\_blocks$, we define the $read\_occupied\_blocks$, which returns as a Boolean value the fact that some block given as an input value is occupied.

```
$p\_bool \leftarrow read\_occupied\_blocks(p\_block) \;\widehat{=}$
PRE
  $p\_block \in t\_block$
THEN
  $p\_bool := bool(p\_block \in occupied\_blocks)$
END
```

Read operations are defined for every variable that has an abstract data type. Sometimes, more than one read operation is given for a given variable. This is the case with partial functions where, before reading the value associated to some input value, one needs to know if the input value belongs to the domain of the partial function. All these read operations are defined as generic pieces of B model. A tool was used to generate automatically read operations for a given abstract machine, by instantiating the generic operations.

All the abstract data types presented here represent the basic elements of B regarded as a high-level language. During abstract modelling, only those types are used. Furthermore, after an abstract variable is defined in the abstract model, the variable will not be refined wherein. Abstract data will only be refined in the concrete model for implementation purpose. Actually in other B software projects we may refine some abstract variables with other abstract variables, but this was not used for Roissy WCU-SCS.

4.3 Abstract Model Architecture

The abstract model architecture is based on the functional breakdown provided by the informal specification documents. The B modules written at this stage are entirely part of the final B model. If we regard the final model as an importation tree (IMPORTS clause), then the abstract model is some higher part of this tree. The abstract model architecture looks like this.

The architecture is a tree of B modules. Two kinds of modules are used: modules with an abstract machine and an implementation and modules with only an abstract machine. Modules with an abstract machine and an implementation are called sequencing modules. They are used to specify sequencing of treatments inside operations implementation. An implementation imports modules defining the operations called by the implementation. Modules with only an abstract machine, named final modules, represent the leaves of the abstract model. They are used to specify data and to specify operations.

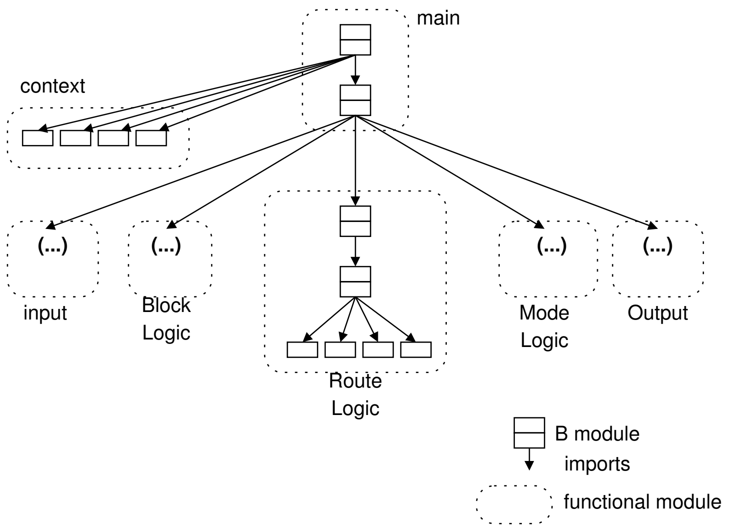

Fig. 3. Abstract Model Architecture

In the architecture, we recognize the main functional modules from the informal specifications:

Main This is the starting point of the project. The main B module, which is the root of the import tree, contains an entry operation called run_cycle which is called at a regular pace, and which, as a sequencing operation, calls indirectly every operation of the B project.

Input This functional module acquires input messages and filters them. Retrieving input data is carried on through basic machines, which have an abstract machine and handwritten ADA code to call low-level input procedures. Due to basic machines, the abstract model of this functional module differs slightly from the core functional modules. It will not be detailed any further in this paper.

Block Logic This functional module computes, in a safe way, the occupation of blocks by trains.

Route Logic This functional module establishes elementary routes, commands and controls switches positions on the tracks. It also contains a sub-function (command of route turnaround cycles) which is not classified as safety critical, but which is nevertheless specified in B as it is a high level function.

Mode Logic This functional module manages train anti-collision. It computes, commands and controls selection and powers discrete train speed programs.

Output This module gathers information computed by the previous modules to prepare the creation of output messages by basic machines.

Context There is also on the far left a subpart called context, specific to the B project. The context defines enumerated and deferred sets (from the SETS clause), concrete and abstract constants. All these constants data are gathered, since they are highly shared inside the B project.

Abstract constants are used to make the final program independent from configuration data. Let's consider again the example of the abstract constant $ctx\_next\_block\_up$ that associates to a block its next upward block. The value of this constant may differ from one WCU to the other. But the properties describing the abstract constant in the PROPERTIES clause remain the same for all WCU. As the B model represents the core of the WCU software, it remains the same for all WCU. The constant values are thus given in ADA. They are specific to each WCU. As their valuation is performed outside the scope of the B process, a specific mechanism has to be used to check that actual constant values fulfill their B properties.

Architecture In the abstract Model architecture, we may have columns of sequencing modules (one sequencing module importing another sequencing module and so on), to reflect that in specifications a treatment calls another treatment and so on. To avoid too many sequencing modules, we use local operations (see [MRefB02] LOCAL_OPERATIONS clause).

The abstract model importation tree is presented such that B modules are allowed to see (SEES clause) other modules located on their left. This rule corresponds to a data flow from the left to the right. Thus, input data are seen by the block logic whose results are seen by the route logic and so on. Context modules may be seen by any other module. Applying this rule makes this architecture valid as regards the constraints of the SEES clause (see [MRefB02]).

An abstract module with only an abstract machine contains abstract variables, their initialization and operations that formalize pseudo-code specifications. As discussed before, operation specifications are deterministic and feasible. Here is an example of a substitution inside an operation body where set-theoretical notation is useful.

$$\begin{aligned}&occupied\_blocks := occupied\_blocks \cup \\ &\quad (ctx\_b2b\_up \cup ctx\_b2b\_down)^{-1}[obd] \cup \\ &\quad otd\end{aligned}$$

Actually due to the size factor, an operation specification may be much larger.

When a functional module is modelled into a too large B module, we try to break it down since the tools are much more efficient on treating many small modules than few large ones, especially when it comes to proof and automatic refinement. Such an abstract model breakdown should respect B architecture rules, especially those concerning variable modifications (see [MRefB02]). So we try to gather operations modifying the same abstract variables in the same modules. When this cannot be performed, them we can still replace a variable by two variables representing the same entity at different moments of the execution cycle. Synchronization of the two variables can be achieved by a parent module, since it has the rights to modify both variables.

Operations traceability is achieved by placing references to informal specification inside the quality-assurance header of operations. This is helpful for inspection and maintenance.

4.4 Abstract Model Properties

The method described so far, can be summarized into formalizing the software functional specification, element by element, into a B abstract model. Each element is more or less independent from the others, which means that a modelling error in one particular element is unlikely to be discovered after the element has been formalized.

During the last stage of the abstract model phase, we insert properties, tying elements together, into the abstract model that has to be proved by the end of the abstract model phase. Those properties strengthen the abstract model since we have to prove that they hold when all elements of the abstract model are put together.

These properties are part of the specification documents. They come from system analysis and they all are safety critical properties. So proving that the software built through this process complies with those safety critical properties is of tremendous importance.

Abstract model properties are written inside the higher-level B module (called the main module), the one with the run_cycle operation. Properties could be either static or dynamic. Static properties are properties that can be expressed with abstract variables (and of course with constants). They are part of the main module invariant. Dynamic properties are properties that link the old values and the new values of some variables, before and after calling the run_cycle operation. So in the main abstract machine, the run_cycle operation is formalized as a "becomes such that" substitution containing dynamic properties. The old value of an abstract variable x is expressed by $x\$0$ and the new value is expressed by x.

At runtime, the software developed in B is used by calling at a regular pace the run_cycle operation. As the B model should be entirely proved, thanks to invariant preservation and refinement consistency of the run_cycle operation, we are sure that static and dynamic properties hold at each call of this operation.

As an example, a property states that a block has to be regarded as occupied when its block detector is occupied or when a beam sensor located at one of the block borders is cut. A simplified predicate formalizing this property is given below.

$$\begin{array}{l} \forall block \cdot (block \in t\_block \wedge \\ \quad ((ctx\_block\_bs\_up[\{block\}] \cup ctx\_block\_bs\_down[\{block\}]) \cap cut\_beam\_sensors \neq \varnothing \vee \\ \quad ctx\_block\_detector[\{block\}] \subseteq occupied\_block\_detectors) \\ \quad \Rightarrow \\ \quad block \in occupied\_blocks) \end{array}$$

In order to prove entirely the abstract model, those properties have to be handled through sequencing modules until they prop on final modules. This top down approach for properties along the abstract model is only useful for explanation purpose. Actually, we used a bottom up approach. The process of formalizing a property is described as follows. After analyzing a property we figure out on which operations the property is propped on. The property may come directly from the postcondition of one final operation or it may come from several operations. Sometimes, a more complex reasoning has to be carried on to convince oneself that the property holds. Usually in those cases, the property is cut into several smaller properties.

The actual B properties come at first from the postcondition of the relevant final operations. They are then spread in upper sequencing modules from the bottom to the top. The abstract variables dealing with properties are redefined in the upper sequencing abstract machines and properties are inserted inside operation specification. After each step, before spreading the property to the next upper level, we check if the lemmas are provable. In fact, at the beginning we do not have a completely clear idea on how the property should be formalized in B, so we first start with an initial version of the property and then we finalize it through proving. In case of non-provable lemma we check our alleged property against the informal property and against the called operations. Through this process, we found errors in the way properties were expressed in B but also inside final operations. Some properties issued from the software specification documents were also made more precise. For instance, the property could be true only in nominal mode, which was not initially stated explicitly in the informal specification documents.

The software specification documents contain 16 properties which had to be formalized in the abstract model. Although it seems to be a small number, some properties had to be cut into many actual properties. At the end, the size of the static and dynamic properties in the main abstract machine was more than 1,000 line long.

5 Concrete Model

5.1 Principles

The concrete Model phase consists in completing the Abstract Model to get to a completely implementable B project. As the operations of the final modules of the abstract model are deterministic, we do not need the specification documents any more. The only input of this phase is the abstract model and the goal is to implement it completely through refinement and importation breakdown. When the concrete model is fully proved, thanks to refinement proof, then we are sure that the concrete model complies with the abstract model.

Actually developing the concrete model is just a technical phase. Ideally, we could use a tool that would translate automatically the deterministic abstract model into ADA code in order to reduce drastically the development cost. However, as abstract model deals with abstract data types (see 4.2) such a tool is not obvious to implement and does not exist yet.

The concrete model was originally handwritten by Siemens (see [Behm99]). The process has been enhanced since. As the abstract model uses a limited number of abstract data types (see 4.2) and as terminal modules use set-theoretical substitutions working with these same data types, the refinement process is similar for all software developed with this method. So this process has been rationalized by developing semi-automatic refinement tools based on the application of refinement rules. These tools are called EDiTh B and Bertille (see [Burd99]).

During the automatic refinement process, abstract variables are either kept through refinement (see [MRefB02] homonymous abstract variable data refinement) or they are refined by a concrete variable with a complete gluing invariant. With this precaution, we are sure that proof refinement guaranties that the concrete model complies with the abstract model. So refinement tools do not require to be validated, since the process is fully covered by proof. In other words, refinement tools are just here to guess the refinement (hopefully a correct refinement), but if their guess is incorrect, proof will detect it.

The activity of automatic refinement by applying refinement rules is very similar to proving by applying proving rules. In both cases it is usually easier to treat a general case, than to treat a special case since it breaks down an initial, and possibly large problem, step by step into smaller problems described by rules with a limited complexity.

5.2 Semi-automatic Refinement

We are now going to describe the semi-automatic refinement. In practice, building the concrete model consists in refining independently every final abstract machine of the abstract model. We build an importation (clause IMPORTS) subtree for every such abstract machine.

Manual Refinement Preparation In a first step we prepare manually the automatic refinement. This step aims to reduce the complexity of the automatic refinement by breaking down the starting point abstract machine into smaller abstract machines and by writing intermediate refinement levels to ease refinement of operations.

To break abstract machine m down, we write an implementation that imports several abstract machines named m_a, m_b, m_c,. The refinement of an operation is made by calling imported operations specified as a subpart of the original operation. Every abstract variable of m is redeclared inside one of the imported abstract machine. Breaking down an abstract machine is not always easy since we have to comply with architecture rules, especially concerning abstract variables modification. However it was always at least possible to split an abstract machine into an abstract machine computing the conditions (of SELECT or IF substitutions) and an abstract machine computing the bodies of those substitutions, since a condition computation only returns a Boolean value and does not involve global variable modification.

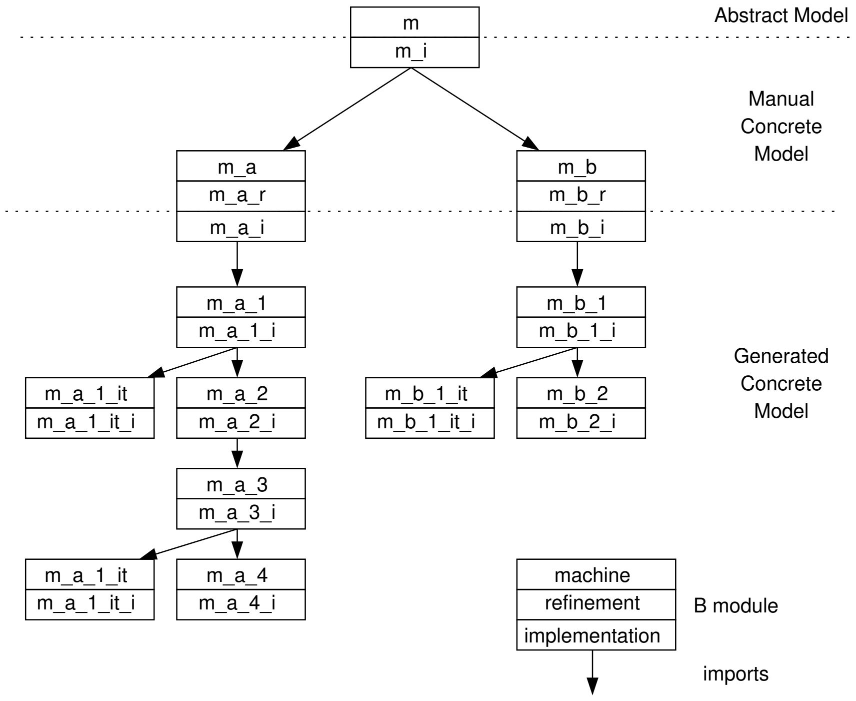

Fig. 4. Concrete Model of a Terminal Abstract Machine

This first step is not mandatory, however it was widely used in the project as we were dealing with very large terminal abstract machines.

Semi-automatic Refinement In a second step we use automatic tools on the last B component of every leaf of the new import tree, which could be either an abstract machine or a refinement. In this case it operates on m_a and m_b. The tool is made up of two passes. It calls once EDiTh B and then it calls Bertille till the concrete model is completed. In the project, automatic refinement may lead to a maximum of 7 levels of generated modules.

EDiTh B is used to implement an abstract machine or a refinement with an implementation and a new sub-abstract machine (suffixed with _1) in which high level abstract substitutions (ANY, SELECT,"becomes such that",) are replace by low-level abstract substitutions (IF, CASE) that can be handled by Bertille.

Bertille implements an abstract machine that contains only low-level substitutions, and it may also produce a new sub-abstract machine (suffixed with _2, _3, _4,) and an abstract machine containing iterators (suffixed with $\_it$). To refine abstract variables and substitutions, Bertille uses a refinement rule base. When Bertille produced a new abstract machine, then it is called again on this abstract machine, until the concrete model is complete or until it fails during refinement.

Abstract variables of an abstract machine are either kept in the next sub-abstract machine (see [MRefB02] homonymous refinement) when they are still needed in the sub-abstract machine or they are implemented by using a variable refinement rule. All the abstract data types described in 4.2 have a corresponding variable refinement. So variable refinement is an easy and complete process.

For example, the $occupied\_blocks$ abstract variable:

$$occupied\_blocks \subseteq t\_block$$

is refined by the $occupied\_blocks\_i$ concrete variable:

$$occupied\_blocks\_i \in t\_block \rightarrow \mathrm{BOOL} \wedge$$
$$occupied\_blocks = occupied\_blocks\_i^{-1}[\{\mathrm{TRUE}\}]$$

Substitution refinement, however, is more complex. As a general principle, substitutions dealing with sets can be refined with the help of set iterators, as shown in the example below.

```
MACHINE $m\_a\_1$
$op\_1 \ \widehat{=}$
    $a := bool(S \neq \varnothing)$
...
```

```
IMPLEMENTATION $m\_a\_1\_i$
$op\_1 \ \widehat{=}$
    ...
    $a :=$ FALSE;
    WHILE $continue = \mathrm{TRUE}$ DO
        $continue, x \leftarrow iterate\_t\_a$;
        $y \leftarrow op\_1\_1(x)$;
        IF $y = \mathrm{TRUE}$ THEN
            $a := \mathrm{TRUE}$
        END
        ...
```

```
MACHINE $m\_a\_1\_it$
$p\_bool, p\_elt \leftarrow iterate\_t\_a \ \widehat{=}$
...
```

```
MACHINE $m\_a\_2$
$p\_y \leftarrow op\_1\_1(p\_x) \ \widehat{=}$
PRE
    $p\_x \in t\_a$
THEN
    $p\_y := bool(p\_x \in S)$
END
...
```

We have to implement here the set-theoretical substitution $a := bool(S \neq \varnothing)$ where S is a subset of the deferred set t_a. The idea is to initialize a to FALSE and then to loop on each element y of t_a. If y belongs to S then S is not empty, so a is set to TRUE, otherwise S is the empty set and a remains equal to FALSE.

To do so, we use an abstract machine containing an iterator on type t_a. This machine is available as a generic machine and offers services to perform efficiently a loop on a deferred set. The knowledge of this kind of iterators is integrated into Bertille, so that it generates automatically the loop variant and the part of the invariant related to the loop indexes. Bertille automatically generates the iterator abstract machine in which it instantiates generic parameters; in this case the t_a type. Bertille also generates the implementation of the iterator abstract machine.

We also use a new sub-machine containing an operation to test if some value x belongs or not to S. As we can see, we have now another abstract machine to implement. However, this operation is this time quite easy to implement, since we just need to implement the abstract variable S with the array S_i and them we can get the value of y from $S\_i(x)$.

The refinement with those tools is only semi-automatic, since Bertille may fail during the refinement of a substitution. In this case, we have to examine the cause of the failure. Most of the time, we add a new rule, or we adapt an existing rule, in Bertille rule base, and that solves the problem. Them, Bertille should be rerun. This action may be repeated several times. At the end of the process, when everything is refined, the concrete B modules automatically produced should be type checked and one should check if the lemmas generated are provable. This is required to check that the code of new rules is correct. As proving easily is also a main issue, we should also check that the lemmas may be easily proved. That is why new rules may sometimes be completed by ASSERT substitutions.

5.3 ADA Code Produced

When the B model is complete and fully proved, it should be translated into ADA. Both the abstract and concrete models are translated. Actually the ADA code produced uses the Digisafe®-ADA technology.

Automatic refinement lowers the cost of a software development but what about the efficiency of the code generated this way? According to Siemens studies, code produced through automatic refinement is 10% slower than hand written code, which is fully acceptable. However automatic refinement may lead to suboptimal algorithms which are not acceptable. To estimate code efficiency at runtime, a static tool was developed by Siemens. It requires, as an input, the dimensioning of types (for example the number of blocks of the t_block type) and gives as an output the estimated runtime of every operation. We spotted a half-dozen exponential-time algorithms where linear-time algorithms could be used instead. This was due to the automatic application of refinement rules. We corrected these points by writing new dedicated refinement rules applying appropriate linear-time algorithms.

As a final remark on the generated ADA code, it is interesting to note that when a member of the project development team took a look at some ADA code corresponding to an automatic refinement level, this person was lost and could hardly make the link with the software specification. Actually, it would be the same for a person who would compare the ADA code he wrote to the corresponding assembly code generated by a compiler. However, proof makes all the difference in our case. Although automatically generated B0 code may look meaningless, we can still be sure that it implements correctly its part of the abstract model.

6 Project Statistics

We shall now present and comment some project statistics.

6.1 B Model Size

B Model	Lines	Rate
Grand total	183,987	100%
Abstract Model without read operations and iterators (324 operations)	28,163	15%
Abstract Model: read operations and iterators	10,503	6%
Manual Concrete model without read operations	27,756	15%
Automatic Concrete Model	117,565	64%

Modules and Components	Number
Number of B modules	532
Number of basic modules	28
Number of intermediate refinements	59
Number of B componants	1,093

The core of the abstract model is 28,000 line long. This is quite a large number that can be explained by the different kinds of sections (normal line sections, garage sections, switch section between the line and the garage). Though they share common principles, they have different software specification, since they have been specified at different times by different people. So they make the specification documents, as well as the abstract model, larger. The rest of the abstract model (10,000 lines) is generated automatically by instantiating abstract iterator machines or read operations.

Although the manual part of the concrete model is also 28,000 long, it does not cost much to produce since it is mostly written by applying simple transformations on a copy of the final abstract machines.

The rest of the concrete model is huge (118,000 lines) but it is automatically generated by tools.

6.2 ADA Code Size

Translated ADA Code	
Lines number (without empty lines nor comments)	158,612
Number of procedures	4,809

The number of ADA code lines is also huge. This is for three reasons:

- The use of Digisafe®-ADA: data and instruction dedicated to Digisafe®-ADA are automatically inserted during translation. We can also point out that Digisafe®-ADA forbids complex expressions or conditions, which leads to an extra use of local variables and "becomes equal" substitutions.
- Every part of the concrete model is usually broken down into many intermediate levels, which produce a lot of code.
- Code is not shared through this process. For example, generic elements share a similar code, however the code is duplicated at each use.

The estimated size of the complete software without Digisafe®-ADA and without so many intermediate levels would be 60,000 lines of ADA code.

6.3 Proof

Proof of Lemmas	Lem. Nb	Rate	Lem. Nb	Rate
Grand total	43,610	100%		
Force 0	38,822	89%		
Force 1	1,397	3%		
Generic demonstrations (61 user pass)	1,950	5%		
- based on predicate prover			1,272	3%
Total of automatic demonstrations	42,169	97%		
Interactive demonstrations (745)	1,441	3%		

Number of interactive demonstrations/day	15

Proof Rules	Nb	Rate
total	290	100%
validated by the predicate prover	243	84%
validated semi-automatiquely	27	9%
validated manually	20	7%

The number of lemmas automatically generated is also high (43,000 lemmas). Thus performance of the tool concerning proof, and especially concerning automatic proof, is a major issue for the project cost. Force 0 of the automatic prover of Atelier B did most of the job, since it proved 89% of all lemmas. Force 1 proved 3%. Around 60 generic demonstrations, mainly based on the predicate prover, proved another 5%. This makes the automatic proof rate of 97%. The remaining 3% lemmas were demonstrated interactively with an average rate of 15 lemmas by man.day.

The concrete model was easier to prove than the abstract model; since everything which is done in the concrete model, is broken down into small steps repeating the same patterns optimized for proof. In the abstract model, the proof of safety properties leads to complex and long interactive demonstrations that cannot be easily reused.

The predicate prover was also very helpful for validating new proof rules.

6.4 Manpower Breakdown

Manpower Cost Rate		
Grand total	100%	
Warmup	5%	
Project management	8%	
Abstract Model	55%	
- questions/answers and documents analysis (267 questions, 4 questions/day)		18%
- inspections		5%
- proof		16%
Concrete Model	24%	
- proof		11%
Finalization (configuration mngt,replay, doc, rules validation)	8%	

The cost ratio between abstract and concrete models is 2/3 for the abstract model and 1/3 for the concrete model.

Most of the time spent on building the abstract model was actually spent on analyzing the specification documents and tracing issues in the question/answer database.

Proof represents a high cost for abstract model (30%) but it is worthwhile, since this is the way to assure that the final code indeed fulfills the safety-critical properties.

The 10% of abstract model time spent in detailed inspections was also of interest, since it led to some corrections and raised new questions/answers.

6.5 Specification Documents Size

Size of input documents (in pages)	
WCU Software Specification Document (84 functional modules)	228
Block Logic Software Specification Document (30 functional algo)	51
Route Logic Software Specification Document (37 functional algo)	80
Mode Logic Software Specification Document (50 functional algo)	98

The size of the main input specification documents gives elements to measure the size of the project.

7 Maintenance

Whenever modifications or evolutions are performed on a piece of software developed through the method presented in this article, what are the consequences on the new software release?

First, we are sure that every release is still consistent, as long as it is again completely proved.

Then, modification cost is limited, if the software structure is not questioned and if the impact on the previous proof work, especially the proof of the abstract model properties, is limited because all the development environment is set up. In this case, refinement rules and proof rules are likely to be efficiently reused. However a major modification of either the specification or the properties may require to develop and to prove again most of the software, but such a situation would also occur with a non formal development.

8 Conclusion

We have presented in this article, a straightforward process, summarized below, and based on reusable generic elements (read operations and iterators components) and on state of the art tools, especially automatic refinement tools.

The process is split into two phases, first software document specifications are formalized into an abstract model and then the abstract model is implemented into the concrete model.

The abstract B model manipulates high-level software data. Those data have a high-level aspect, since they use abstract data types such as sets of scalar types, relations, partial and total functions from an abstract data type into another abstract data type. However, they also have a concrete software aspect since every abstract data type is directly and systematically implemented by concrete data. This is the only data refinement used: an abstract variable defined in the B model is kept unchanged in lower imported modules (it is refined by an homonym abstract variable) until it is finally refined by its associated concrete variable(s) in the concrete model.

Sequencing treatments are formalized as sequencing calls of operations in the implementation of sequencing modules.

Core logical treatments are formalized in operation specifications by high-level software substitutions. They are high-level, since they use set-theoretical expressions and parallel substitutions. However these substitutions are deterministic and they can be implemented through automatic refinement rules. The refinement process to generate the concrete model starting from the abstract model is semi-automatic. Refinement rules implement step by step such a set-theoretical substitution by generating software loops going through every element of the set.

All these characteristics make the B Method used through this process, a high-level programming language. However, a key feature of this so-called language lies in its proof capabilities. The process offers the possibility of proving that some properties are indeed preserved by each call of the main software procedure. Obviously, in the case of safety critical software those properties shall be safety critical properties. The process also proves that the final code correctly implements its formal specification.

The process described here is suitable for any industrial domains, not only for railways command/control software. Actually this process deals with designing procedural software based on logical treatments, not based on real or floating-point numbers. It is all the more suitable that software specification can be easily formalized into set-theoretical expressions.

From the management point of view, the project went off according to the initial schedule, although the software produced is quite large, thanks to a straightforward process and efficient tools.

Every verification stage throughout the process was useful and led to early error detection: analysis of software document specification, type checking, inspections, proof of abstract model safety properties, refinement proof of correct implementation. The WCU-SCS is currently being integrated with the rest of the software.

Acknowledgments: We would like to thank Laurent Voisin for his very interesting comments.

References

[Abr96] Abrial, J.-R., Extending B Without Changing it (for Developing Distributed Systems), 1996
[BBook96] Abrial, J.-R., The B-Book: Assigning Programs to Meanings, 1996
[Behm99] Behm P., Benoit P., Faivre A. and Meynadier J.-M., Météor: A Successful Application of B in a Large Project, 1999
[Burd99] Burdy L., Meynadier J.-M., Automatic Refinement; BUGM at FM'99, 1999
[Dol03] Dollé D., Essamé D., Falampin J., B dans le transport ferroviaire, l'expérience de Siemens, Technique et science informatiques - volume 22, 2003
[MRefB02] Badeau F., B Language Reference Manual v1.8.5, 2002