Formal Methods in Industry: Achievements, Problems, Future

Jean-Raymond Abrial
Swiss Federal Institute of Technology Zurich
jabrial@inf.ethz.ch

ABSTRACT

Two real projects using the B formal method are quickly presented. They show how some important parts of complex systems can be developed in such a way that the outcome is "correct by construction". A number of factors are then analyzed relating the pros, the cons, and the difficulties in applying this approach in Industry.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies, Tools*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness proofs, Formal methods, Validation*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Invariants, Mechanical verification, Specification techniques*

General Terms

Design, Management, Reliability, Verification

Keywords

Correctness, Formal method, B, Train system, Development process

1. INTRODUCTION

The purpose of this presentation[1] is to report on experiences of using formal methods in Industry. Although I will mainly present cases which I know of (all of them using B [1]), I will infer from these case studies some results which are applicable to the usage of other formal methods. I first present the cases succinctly and then I analyze the lessons one can learn from them. Before doing this however, I briefly summarize what I mean by the term "formal method".

[1]This work has been partly supported by IST FP6 Rigorous Open Development Environment for Complex Systems (RODIN, IST-511599) Project.

ICSE'06, May 20–28, 2006, Shanghai, China.

2. FORMAL METHOD

The term "formal method" used in this paper has a very precise, and maybe narrow, meaning. It is necessary to recall such a meaning here as this term is used nowadays with many different understandings. In order to keep this paper short, I only give the *most important concepts* at work in this usage of the B formal method.

2.1 Formal Development

The B formal method used in the examples of this presentation is an approach which the engineers use to transform a *Software Requirement Document* into some *Executable Code*. So far, this does not seem to be different from what the ordinary programmer does. What makes this approach significantly different however is that the engineers applying it are *not programming any more*. As a matter of fact, they are not using a classical programming language. They are rather working at a more abstract level *where execution is no longer possible*.

As a consequence, testing and debugging, cannot be made by executing some pieces of code. But as, clearly, we cannot assume that engineers will not make errors any more, they have to have other means to verify what they are doing. We shall see below in sections 2.2 and 2.4 what replaces executable testing.

This approach can be divided up into three distinct and successive phases:

(1) In phase 1, details of the problem are *gradually* extracted from the Software Requirement Document. This results in the eventual construction of an *Abstract Model* by stepwise refinement. The extraction and gradual construction of the Abstract Model requires a heavy human intervention.

(2) In phase 2, the Software Requirement Document is not used any more. In fact, the origin is now the Abstract Model. It is transformed, again gradually, into a *Concrete Model*. This phase also requires some human intervention, but, as we shall see in section 4.5, it is done in a less extensive way than in the previous phase.

(3) In phase 3, the Concrete Model is automatically translated into *Executable Code*. No human intervention is required any more in this phase.

The overall result of this approach is that the Executable Code will be *correct by construction* with regard to the Software Requirement Document. In the three following sections, these phases are briefly presented.

2.2 The Abstract Model

In this section, I succinctly describe the structure of the Abstract Model and I also make precise the way it can be verified by the engineers *while being constructed*.

The Abstract Model is first made of some *data* which are introduced by means of a number of *invariants*, which considerably enrich the simple notion of *data declarations* used in programming languages. Such invariants are predicates written by means of *first order logic* and *set-theoretic* constructs (set, relations, functions, etc.). As its name indicates, an invariant is a property of the data which remains untouched (although the data themselves are) during the evolution of the system. Note that an invariant might be a global system property, which clearly cannot be expressed in a programming language. Also note that the variable, hence the invariants, are not introduced at once. They are rather gradually incorporated by stepwise superposition as the construction of the Abstract Model proceeds.

The dynamics of the model is expressed by means of simple *transitions* which are, most of the time, *non-deterministic*. The definitions of such transitions also uses classical set-theoretic constructs. Again, as for the data, such transitions are enriched as the construction of the Abstract Model proceeds by stepwise superposition.

2.3 Proofs

The engineers are able to gradually verify their construction of the Abstract Model by means of *mathematical proofs*. Note that the statements to be proved are not defined by the engineers themselves, they are automatically generated by a tool called the *Proof Obligation Generator*. For doing so, the tool successively analyzes the various refinements of the Abstract Model. Note that it is very important to have a tool here to determine what is being proved (several thousands of such proof obligations are to be done). Otherwise, the engineers could do some errors in generating these verification statements: one would then just move the complexity from one area to another.

The proofs to be done have two distinct concerns:

(1) The preservation of the invariants by the transitions of the Abstract Model. This first concern is called an *invariant preservation proof*,

(2) The fact that each more accurate version of the Abstract Model does not invalidate the properties which were already proved in the previous versions. This second concern is called a *proof of correct refinement*. In the development of the Abstract Model, this second concern results in proof obligations which are less complex than the ones dealing with invariant preservations. The reason for this is that the steps we are doing at this stage are just *superposition steps* as we have said above in the previous section: at each step, we simply add new variables without destroying those that were introduced in previous steps, and we simply enrich the transitions.

The proofs are performed, most of the time *automatically*, by a tool called the *prover*. A very small proportion (we shall come to that below in section 3.3) of the proofs however are performed in an interactive way: the engineer has then to give some hints to the automatic prover.

2.4 The Concrete Model

The Concrete Model has exactly the same structure as the Abstract Model: it is made of data and transitions. The Concrete Model is also gradually constructed by stepwise refinement.

However, although the final version of it contains data that are still *genuine set-theoretic objects*, the Concrete Model has data which are in a one-to-one correspondence with computerizable objects: scalar, pointers, arrays, files, etc. Likewise, the final version of the Concrete Model will have transitions defining the dynamics of the system exactly as the Abstract Model had. But, this time, these transitions are completely deterministic and are reminiscent of what is found in classical programming languages: sequencings, conditional statements, loops, procedure calls, etc.

In the case of the Concrete Model, the proof statements generated by the Proof Obligation Generators are similar to those generated for the Abstract Model verification. But contrarily to what happened in the case of the Abstract Model construction, this time the more complex proofs are the refinement ones. This is because we now essentially perform two specific tasks:

(1) We *data-refine* the abstract set theoretic constructs used in the Abstract Model (set, relations, functions, etc.) into computerizable set-theoretic objects.

(2) We transform the set-theoretic non-deterministic transitions into classical deterministic programming statements.

2.5 The Executable Code

Besides being verified by mathematical proofs, the last version of the Concrete Model is also automatically verified by a tool checking that it only contains data and transitions which are mechanically translatable into executable code. Once this verification is positive, the translation is performed in two successive phases although it could very well be done in one phase only:

(1) The Concrete Model is automatically translated into its counterpart written by means of a classical programming language. In the cases I shall describe, the language is ADA.

(2) The programming language outcome of the first translation of the Concrete Model is then translated into executable code using a classical compiler.

Note that these automatic phases are *weak points* of this approach as we are not certain that the translators (in particular the second one) are doing correct translations. We shall see below in section 4.6 how this problem is (partially) handled. Also note that another weak point of this approach is the Software Requirement Document. We shall come back to this very important point in sections 4.1 and 4.10.

3. THE CASE STUDIES

The two case studies presented in this section are separated by an eight year period: the first one resulted in a system working since October 1998, whereas the second one will be operating in September 2006. Everyone can have the opportunity to experience the corresponding systems since the first one is the fully automatic driverless subway servicing line 14 in Paris intra-muros, whereas the second one will be the fully automatic driverless shuttle servicing the various terminals of Roissy Airport.

Note that, in both cases, the programs of these systems were not all developed using the formal method. The preliminary system studies determined which parts should be formally developed and proved: this corresponded to the *safety critical parts* representing one third of the overall program.

Line length	8.5 km
Number of Stops	8
Time interval between two trains	115 s
Speed	40 km/h
Number of trains	17
Passengers per day	350,000

Table 1: Parameters of Line 14 of Paris Subway

Line length	3.3 km
Number of Stops	5
Time interval between two trains	105 s
Speed	26 km/h
Number of trains	14
Passengers per hour	2,000

Table 2: Parameters of Roissy Airport Shuttle

3.1 1st Case Study: Line 14 of Paris Subway

Table 1 gives the main figures of this subway line [11]. The formally developed part of this system has been described in a very well documented article [5], which is highly recommended to the interested reader. The following rough description is extracted from this article.

Since this heavy subway system is completely automatic, the safety critical part concerns the running and stopping of the trains and the opening and closing of the doors in the train and in the platforms. The overall program is distributed into three different kinds of sub-systems: the wayside equipment (several such equipment installed along the tracks), the on-board equipment (one equipment in each train), and the line equipment (one equipment). These sub-systems are heavily connected. In each sub-system, the safety parts which are developed using the formal method have the following characteristics: they are sequential and cyclic (350 ms), and they constitute a single non-interruptible task.

3.2 2nd Case Study: Shuttle at Roissy Airport

Table 2 gives the main figures of this shuttle line. It is also extracted from [11]. This system has been described in a very well documented article [4], which is also highly recommended to the interested reader.

The Roissy Airport shuttle system is derived from the light shuttle system of Chicago O'Hare Airport. The difference between the former and the later is that the former has a significant computerized part located along the tracks and called the Wayside Control Unit. There are several such units disposed on the tracks. They are linked by means of an Ethernet network.

The Wayside Control Units are driving the trains by sending them the predefined speed programs they have to follow. This is done in response to the actual situation detected by means of some sensors situated on the track and connected to the Wayside Control Units.

3.3 Comparing the Two Case Studies

In table 3, a number of information is recorded making it possible to compare the two case studies. The most important data are the first and the last. The first information contains the number of lines of the two programs, whereas the last one contains the time needed to perform the corresponding interactive proofs.

The number of ADA lines represents the size of that part of the software system which has been developed using the formal method. These lines were *not modified* by the engineers.

The time used for doing the interactive proofs is calculated by taking an average of 15 interactive proofs per Man.Day, and 21 days in a month. As can be seen, the gain from the first to the second case study is quite significant. Roughly speaking, twice as many lines of code were automatically generated for half of the proving time. The manufacturer also said that in the second case study a significant time was saved in the building of the Concrete Model. In section 4.5, we explain where such differences are coming from. Note that in both cases *no unit tests* were performed. We shall see below in section 4.3 what kind of tests were still performed.

One important difference between the two case studies is that the Software Requirement Document of the first one was done specially for it, whereas the one of the second case study was derived from an existing Requirement Document (that of the Chicago O'Hare Airport shuttle). In that second case, this document had to be modified and extended in order to deal with the new requirements and functionality of the Roissy Airport shuttle. This has caused a number of problems which were only discovered during the development of the Abstract Model.

3.4 Other Similar Case Studies

Similar train control systems are presently developed in the same way for the New York City subway, the Barcelona subway, the Prague subway, and line 1 of the Paris subway.

Items	1st Case Study	2nd Case Study
Number of ADA lines	86,000	158,000
Number of proofs	27,800	43,610
Percentage of interactive proofs	8.1	3.3
Interactive proofs time in Man.Month	7.1	4.6

Table 3: Comparison of the Cases Studies

4. LESSONS TO BE LEARNED

In this section, I analyze a number of important points related to the usage of this approach in Industry.

4.1 Importance of the Requirement Document

As the point of departure in these case studies (as well as in many others) is the Software Requirement Document, it is then clear that the quality and completeness of such a document is of utmost importance.

This document is written using a mixture of semi-formal approaches. But, on the whole, it is mainly written in natural language. Generally speaking, this document may have two kinds of related weaknesses:

(1) It might be of a poor quality, either too short, or on the contrary, far too verbose. But, in both cases, the *precise requirements* are difficult to extract. It is important to make clear here that the purpose of using a formal method is not to correct the Software Requirement Document. In other words, if there exists an error or an omission in this document it might very well be the case that this error or omission be also present in the Abstract Model. Notice however, that the proofs done with the Abstract Model could reveal some inconsistencies in the Software Requirement Document.

(2) Its comprehension by the developer of the Abstract Model could be erroneous so that the Abstract Model would not reflect the intention of the authors of the Software Requirement Document. This is partially addressed by having a special *validation team* (independent from the development team) whose work is precisely to inspect the Abstract Model in order to be sure that it correctly reflects the Software Requirement Document. Such a team was put in place with great profit in the two case studies.

Nevertheless, these points are so important that we might below (section 4.10) propose ways to improve the present situation.

4.2 Difficulties with the Abstract Model

The most difficult part in using a formal method is clearly encountered in the construction of the Abstract Model. In general, engineers (especially Software engineers) are not well educated in the discipline of *modeling*, and, as pointed in the previous section, they are not well formed either in the discipline of *Software Requirement Document writing*.

Usually, the *gradual construction* of the Abstract Model is not mastered by Software engineer encountering this approach for the first time. People have the tendency either to make no step at all, or else to make a very few steps. This results in a heavy proof effort (we shall come back to the proof effort in section 3.3), and in difficulties encountered in the validation of the Abstract Model against the Software Requirement Document. In fact, the concept of refinement (independent of its technical aspect) is *poorly understood* by Software engineers. One should notice that it is not the case in other more mature engineering disciplines where modeling, hence refinement, is completely natural.

One of the reasons for this poor usage of refinement is that it is not easy to decide how to organize the construction steps: what is the best order to be used to incorporate the various constituents of the Software Requirement Document in the Abstract Model. Is it better to start with the functional requirements, then the safety requirements, then the failure requirements? Or should we take another order? At the moment, there is no definite answers to such questions. Experience still plays an important role.

In spite of these difficulties, one of the great merit of the formal Abstract Model, which will become the *sole model* of the future design, is to force the engineers to have an in-depth understanding of the Software Requirement Document. In the two case studies we mentioned earlier, there has been a lot of interactions of various forms (mails, meetings, inspections) between the developers and the authors of the Software Requirement Document. During such interactions, a large number of clarifications and extensions were added to this document.

An important aspect of this relationship is that the Software Requirement Document becomes the starting point of the *traceability* of the requirements. Such a traceability will be found first into the modeling, then into the future design, and finally into the executable code.

4.3 Comparing Testing and Proving

In both cases, the *unit tests* as well as the *integration tests*, which are very heavy in safety critical systems, hence very expensive, have been completely abandoned. The proofs are considered a *far better verification process* than the test. Notice that this removing of the unit tests was proposed by RATP, the Parisian Subway Authority, to the manufacturer.

It does not mean, of course, that all tests are cancelled. The testing effort could now be concentrated on a far more important point than the software modules and their correct integration, namely the Software Requirement Document itself, which was considered a weak point in the previous section. The idea is to have the validation team, alluded above in section 4.1, defining a number of *global functional tests* (as well as catastrophic scenarios) able to detect errors or omissions in the Software Requirement Document.

Since the proofs have partially replaced the tests, it is certainly useful to compare the two and see what each of them may provide to the developer.

4.3.1 Tests.

A well-prepared program test comprises four phases which are the following:

(1) The definition of the *precise property* to be tested in the program at hand.

(2) The elaboration of the *expected result* of the test: this has to be done *before* executing the test. Note that this elaboration is sometimes difficult to obtain as it has to come from a source which, clearly, must be independent of the program to be tested. Sometimes, however, the source of the elaboration of the result of the test is just in the head of the tester himself as he is looking at what the program does!

(3) The test itself by running the program being tested.

(4) The *comparison* of the result of the test with the expected result elaborated before doing the test. The fact that the result of the test is the same as the expected result does not mean that the program is correct: it only means that the program has passed the test. When the comparison is negative, one could think that the program is not correct, but it can also be that the expected result was wrong!

4.3.2 Proofs.

In the case of a proof done on a model, the situation is completely different from the one we have just seen with testing. In fact, phase (1) above of the testing (choosing a property to be tested) does not exist, for the simple reason that the abstract model of a future program is nothing else but the list of properties which are necessary to qualify that program. In other words, *the properties are part of the model*, they are not chosen a posteriori as is the case for program testing. This shows the very important distinction to be made between programming and abstract modeling. In programming, you make precise what a computer should do. Modeling has nothing to do with instructing a computer, it simply denotes the static and dynamic properties of the future program, and it allows the engineers to *reason* about them.

The result of the proof could be one of the following:

(1) The prover succeeds to do the proof (either automatically or interactively).

(2) The prover succeeds in proving the negation of what was supposed to be proved.

(3) The prover fails but the engineer has the strong impression that what is to be proved is nevertheless true.

(4) The prover fails and the engineer feels that it would also fail to prove the negation of what is to be proved.

Case (1) corresponds to what we are aiming at for all proofs by the end of the proving phase of a complete model. Case (2) is interesting because it points to what has to be modified in the model. In case (3), the failure of the prover is not to be taken, most of the time, to the prover itself: it is an indication that the model is too complicated and has to be reorganized. It turns out that this indication is extremely useful. When the prover has some difficulties then it might be because the structure of the model is poor: it came as a complete surprise for us. This information became then a systematic quality criteria. As a matter of fact, when the prover could not automatically discharge 90% of the proofs, then the engineer had to reorganize his model. Case (4) means that the model is not rich enough, it has thus to be extended.

These cases are all extremely interesting because their outcome are fully integrated in the development process itself. To summarize, modeling and proving is not a goal per se. It is rather an excellent basis for asking oneself questions about the system we want to construct.

4.4 Methodological Document

In both case studies, the engineers were using a proprietary document called the "B Development Guide". This document is comparable, although used in a completely different context, to the "Design Pattern" book [9] of Gamma et. al. It is a "methodological repository" that the engineers can use in order to do their mathematical modeling. People found that such a document is very important. It allows the engineers to work in a very systematic fashion. We shall see below in the next section that this "B Development Guide" can be partly transformed into a specific tool.

Among other things, the "B Development Guide" presents a number of rules to be followed in order to model various kind of automata which can be found in the Software Requirement Document. It also indicates how to interface B developments with software parts which are not developed with the formal method. Finally, it proposes lots of techniques to be used in the data refinement: it shows how to formulate the formal model so that the automatic proofs are easily handled by the prover.

4.5 Constructing the Concrete Model

The main difference between both presented case studies is to be found in the construction of the Concrete Model. In the second case studies, the concrete model was almost entirely built with a *refinement tool* named "EdithB". This tool performs semi-automatic data-refinements. It has been succinctly described in various articles [7, 8].

The tool EdithB interprets the sub-models of the Abstract Model. It detects where each abstract set-theoretic data is defined and used. EdithB uses then a database of predefined data-refinement schemes which it tries to apply systematically. When EdithB fails, the engineer can provide his own refinement scheme and add it to the database of EdithB.

In order to be sure that EdithB does not perform wrong refinements (since EdithB might have bugs in it), the outcome of it is nevertheless proved as if it had been produced manually by an engineer. This prevents doing a very complex proof of EdithB itself.

The net result of using EdithB is quite spectacular since almost 2/3 of the B models could then be generated automatically. Comparisons have been made with a completely manually produced models. It turns out that the final code, although slightly less efficient, is nevertheless satisfactory.

4.6 The Vital Coded Processor

As pointed out in section 2.5, the translation of the last Concrete Model refinements to ADA and then the translation of ADA to object code are weak points in the development processes. We briefly explain here how this can be make safer.

First, there are two independently developed translators from B to ADA. These translators are both executed on the last refinement stages of the Concrete Model. The way these codes are compared is explained in [4], and [8].

In the presented examples, the usage of the B formal method is complemented by having the automatically generated software being executed on an extended processor, the *Vital Coded Processor*, which is described in [6]. Roughly speaking, each data is encoded in two parts: a normal part, which contains the value of the data, and a redundant part. During execution, the Vital Coded Processor checks for any inconsistencies that could be detected between the two parts. In case of inconsistencies, then the system is put in a safety state (i.e. the train is stopped).

The usage of the *Vital Coded Processor* makes possible to detect errors caused by some "spontaneous" changing of the value of the memory (due to the heavy electronic noises in the subway tunnels).

The usage of the formal method and the Vital Coded Processor are complementary in that the former guarantees that the software is correct (respects its requirements), whereas the later guarantees that the execution of the software is correct.

4.7 Integration in the Development Process

One of the *most difficult obstacles* encountered in Industry for using formal methods is the integration of such approaches within the *software development process*, which everybody nowadays adopt.

People are quite reluctant to use such methods mostly because it necessitates to modify the development process in a significant fashion. As it is well known, such development processes are hard to develop and even harder to put in place so that the working engineers are using them. This is one of the the main reason why managers do not want to modify them.

In fact, the initial and middle phases (that is, Abstract Model construction and Concrete Model construction) are far more important than similar phases (that is, technical specification and design) in more classical software developments. They are then clearly more costly. On the other hand, the last phases (programming, integration, and testing) are far less important (less costly then). But managers do not like to have such changes in the financial figures because they do not believe that spending more time and money at the beginning of a project will save a lot of time and money at the end of it!

Besides these changes in the sizes of the phases, the main changes in the development process is the incorporation of a proof activity. Managers are afraid that engineers will not be able to perform the interactive proofs.

Most of the manufacturers in the studied field (train systems) do not use formal methods. Although there are some standard requirements corresponding to Safety Integrity Level 4 in this field, the usage of formal methods (with proofs, as presented here) is not well spread. Asked why it is the case, the manufacturers give the classical answer that there is no point for them to use formal methods since potential clients do not ask for them. In other words, they do not want to especially invest in this area because it is not enough used. They claim that the investing cost will not be paid off by more clients.

4.8 Instructing Software Engineers

Experience has shown that engineers can easily learn the mathematical concepts and notations used in the formal method. What is more difficult to get however is the ability to develop formal models that are understandable and easy to prove. At the beginning, engineers have the tendency to pseudo-program rather than to really build models.

It is my opinion that the two disciplines of modeling and writing Software Requirement Document should be made far more important in Computer Science curriculum. I do not think that courses in UML cover these problems. Such new courses should be given with practice in mind. The students have to exercise themselves in these disciplines which are not presently well understood. But for doing so, it is clear that these disciplines must be taught together with corresponding tools, which the students can use. Otherwise, it might remain too abstract. In Zurich, at the Swiss Federal Institute of Technology, I have developed two such courses: one at the Undergraduate level and the other one at the master level.

4.9 Less Usage of Programming Languages

An important lesson to be learned from using formal methods is that the usage of a High Level Programming Language as the basic tool of the developers is significantly decreased. In the example we have seen in section 3, the engineers have not used the ADA language for the development of the safety critical parts (remember the ADA code was not touched by the engineers as explained in section 3.3). Although, for practical reasons due, in particular, to the remaining part of the Software which was not developed using the formal method, the last refinement of the Concrete Model was translated into ADA.

In fact, the classical notions of source and object codes, and of a compiler situated between the two and used to translate one into the other, has almost disappeared. As pointed out in section 2.5, we certainly have a final object code, but we might say that we have not any more a single source code but *many layers* of them (some of them very abstract) corresponding to the various refinement steps we construct (often more than twenty all together in the Abstract and Concrete Models!).

It appears then that using a language (reminiscent to a programming language) in order to define the various layers of our models is probably not any more the right approach. It seems that a far better one would be that of using a database within which the various layers of the models could be stored and connected together by the refinement and decomposition relationships. This database will contain a number of basic *modeling elements* which are the constituents we briefly presented in section 2: data, invariants, transitions, refinement layers, proof obligations, proofs, etc. The engineer can then navigate through this database which is in *constant evolution* during the development phase of the Abstract and Concrete Models. Then, the proof obligation generator, the prover, and the refiner could partially take the place of the engineer to do their more clerical and auto-

matic jobs directly on the database, and they will, of course, store their results into it. Notice that many other tools could be installed around this database.

The European Project Rodin involving various European academic institutions (Newcastle, Southampton, Turku, Zurich) as well as major industries is aiming at developing such an open development database [10].

4.10 More Usage of Formal Methods

In section 4.1, we pointed to the weaknesses of the Software Requirement Document. Most of the time, this document is very concrete in that it contains a completely defined architecture of the future software as well as some very detailed descriptions of the various modules this architecture is made of. There is nothing wrong with that, of course. The only problem is that it has been defined (indeed usually by very competent people) in an informal way. As a consequence, it might contain some *early bugs* which, as we have said in section 4.1, cannot generally be detected by the usage of formal methods made in further phases. And, as we have seen in section 4.3, such bugs could very well be only discovered *at the very end of the process* when performing the global tests: this situation is potentially dangerous.

One possibility to improve the situation is to use a formal method in order to construct the Software Requirement Document itself. For doing so, one would use a similar approach to that advocated earlier, namely that of building models. We have to start, of course, from some initial informal document, which could be called the System Requirement Document. Such a document would contain the very important requirements of our future system (comprising software and equipment) without mentioning at all any architecture or even any distinction yet between the future software parts and the future physical equipment parts.

At this level, the purpose of the models, which is, as usual, gradually constructed by successive refinement steps, is to obtain an harmonious architecture taking account of the system requirements which might sometimes be slightly in opposition (i.e. safety and functionality). It is outside the scope of this paper to explain how this can be done. Let us just say that a basic idea is to construct closed models as advocated by [3]. A book presenting this approach is in preparation and will be published soon [2].

5. CONCLUSION

In this presentation, I briefly introduced two case studies of existing industrial and widely used systems where the B formal method has been used to develop some safety critical parts. I also proposed a number of comments concerning the advantage and difficulties of using such methods. I tried to explain why the introduction of such methods is rejected by some industrial managers.

6. REFERENCES

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. CUP, 1996.

[2] J.-R. Abrial. *Event-B*. To be published, 2006.

[3] R. Back. Decentralization of process nets with centralized control. *Distributed Computing*, 1989.

[4] F. Badeau. Using B as a high level programming language in an industrial project: Roissy val. In *Proceedings of ZB'05*, 2005.

[5] P. Behm. Meteor: A successful application of B in a large project. In *Proceedings of FM'99*, 1999.

[6] L. Burdy. Vital coded microprocessor: Principles and application for various transit systems. In *Proceedings of IFAC-GCCT 1989*, 1989.

[7] L. Burdy. Automatic refinement. In *Proceedings of BUGM at FM'99*, 1999.

[8] D. Dolle. Vital software: Formal method and coded processor. In *Proceedings of ERTS 2006*, 2006.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[10] Rodin. European Project Rodin http://rodin.cs.ncl.ac.uk

[11] Siemens. Siemens transportation systems, 2006. http://www.siemens.fr/transportation.