Formal Methods: Theory Becoming Practice

Jean-Raymond Abrial
ETHZ, Zurich, Switzerland.
email: jabrial@inf.ethz.ch

I. Introduction

In this paper I will introduce the ideas behind the B formal method, especially as crystalised in its latest version, *Event-B*. I shall give some insights on *modeling* and *formal reasoning*, activities intended to be performed *before* undertaking the actual coding of a computer system, so that the software in question will be *correct by construction*. At the end, I shall present a number of *real projects* where the B formal method [1] has been used with great success.

II. Formal Methods

Formal methods are inescapably tied to mathematics of one sort or another. Consequently many claim that using formal methods is too challenging because of various intrinsic difficulties. Here are a few of these claimed difficulties: you have to be a mathematician; the proposed formalism is hard to master; it is not visual enough (boxes and arrows may well be missing); people will not be able to perform proofs.

I mostly disagree with the above point of view, but I recognize that there are some real difficulties, which, in my mind, are the following:

- When using formal methods, you have to think a lot before coding, which is not, as we know, the current practice.

- The usage of formal methods has to be incorporated within a development process, and this incorporation is not easy. In industry, people develop their products under precise guidelines, and usually, the introduction of such guidelines in an industry takes a significant time. In this context, managers are very reluctant to change the guidelines to incorporate the use of formal methods.

- Model building is not a simple activity, and one has to be careful not to confuse modeling and programming. More precisely, the initial model of a program describes the properties that the program must fulfil in order that we can judge that the final program is correct.

- Modeling has to be accompanied by reasoning. The model of a program is not just a piece of text, whatever the formalism being used; it should also contain proofs that are related to this text. Often, formal methods have just been used as a means of obtaining abstract descriptions of the program, but descriptions are not enough. We must justify what we write by proving some consistency properties. The problem is that software practitioners are not used to constructing proofs, unlike people in other engineering disciplines. A genuine obstacle here is the lack of good tool support, usable on a large scale.

- A final difficulty encountered in modeling is the frequent lack of good requirement documents associated with the project. Most of the time, industrial requirement documents are either almost nonexistent or far too verbose. Usually they have to be rewritten before serious modeling starts.

III. A Little Detour: Blueprints

It is my belief that people managing the development of large and complex computer systems should adopt a view shared by all mature engineering disciplines, namely *using an artifact to reason about the future system during its construction*. In these disciplines, people use *blueprints* in the wider sense of the term, which allows them to reason formally during the construction process. A blueprint is a *representation* of the future system, giving insight into some but not all of its aspects. You cannot drive the blueprint of a car!

The blueprints used in mature engineering disciplines have some very desirable properties that we seek to emulate in ours. They are typically organised into hierarchies in which: more detailed blueprints are related to less detailed views of the same thing; blueprints are decomposed to enhance readability; earlier blueprints do not completely specify all aspects, leaving later blueprints to refine options left open.

Most of the time in software construction, people do not use such artifacts. This results in a very heavy and inevitably incomplete testing phase on the final product, with its well known problems. The alternative, of using appropriate blueprint models of the future system, ultimately allows us to *prove* that the system will satisfy desirable properties, which can be stipulated early on.

IV. The Requirements Document

Before the blueprints we have just discussed can be created, comes the very important phase of creating the *Requirements Document*. As already noted, most of the time this document is either missing or very badly written. Usually this results in lots of difficulties in subsequent phases. In particular, the syndrome of unavoidable specification changes occurring during the design phases originates in the weakness of the Requirements Document. When this document is well written, these kinds of difficulties tend to disappear. For this reason we are going to dwell on requirements for a while to see how this phase can be improved.

Difficulties with the Requirements Document

Writing a good Requirements Document (RD) is a difficult task. We must remember that the readers of this document perform the phases that follow, namely Technical Specification and Design. Usually, it is difficult for them to exploit the RD because they cannot clearly identify what they have to take into account and in what order.

Often too, important points are missing from the RD. I have seen a huge RD for an aircraft alarm system where the simple fact that the system should not deliver false alarms was simply missing. When the authors of this document were interrogated on this point, the answer they gave was rather surprising: it was not necessary to put such detail in the RD "of course everybody knows that the system should not deliver false alarms." On other occasions, the RD is full of irrelevant detail.

What is difficult for the reader of the RD is to make a clear distinction between the part of the text that is devoted to *explanations* and the part that is devoted to genuine *requirements*. Explanations are initially needed for the reader to understand the future system. But when the reader is better acquainted with the system, explanations become less important. What counts then, is to recall what the real requirements are, in order to know exactly what has to be taken into account during system construction.

Structuring the Requirements Document

The idea is to have our RD organized around two texts embedded in each other: the *explanatory text* and the *reference text*. These two texts should be immediately separable, so that it is possible to summarize the reference text independently.

Usually, the reference text takes the form of self contained *labeled and numbered short statements* written using natural language, which must be very easy to read independently from the explanatory text. The explanatory text is there to comment on the formal requirements in a way that could help a reader on first becoming acquainted with the document. However, after the initial acclimatisation period, the reference text is the only one that counts. It is only the the reference text whose labels and numbers act as cross references in later development phases, providing *traceability*, so that it can be checked whether all the requirements have in fact been taken into account properly.

V. The Context of the B Method

Every methodology has its own appropriate sphere of applicability, and the B methodology we are going to describe is no exception. In this section we focus on the context of the B method. The kind of systems we are interested in developing are *complex* and *discrete*. Let us develop these two ideas for a while.

Complex Systems

What is common among, say, an electronic circuit, a file transfer protocol, an airline seat booking system, a sorting program, a PC operating system, a network routing program, a nuclear plant control system, a SmartCard electronic purse, a launch vehicle flight controller, etc.? What is common to the requirements, specification, design and implementation of *systems* that are so different in size and purpose?

Well, almost all such systems are *complex* in that they are made of many parts interacting with a highly evolving and sometimes hostile environment. They can also involve concurrent executing agents. They require a high degree of correctness. Finally, most of them result from a long construction process requiring a large and talented team.

Discrete Systems

Although at the physical level their behavior is ultimately continuous, the systems listed above mostly operate in a *discrete fashion*. They fall under the generic name of *transition systems*. Having said this does not give us a method, but it gives us at least *a common point of departure*.

Some of the examples listed are pure programs. In other words, their transitions are essentially concentrated in *one medium* only, e.g. the electronic circuit and the sorting program. Most of the other examples however are more complex than that because they involve many different executing agents, and also heavy interaction with the environment. This means that the transitions are executed by different entities acting concurrently. Fortunately this does not change the discrete nature of the problem.

Test Reasoning versus Model Reasoning

Thus the technology we consider in this short paper is concerned with the construction of *complex discrete systems*. A very important factor here is that they should operate in a *correct* fashion. And as long as the main validation method used is testing, we consider that the technology remains in an underdeveloped state. Testing, or 'laboratory execution,' does not involve any kind of sophisticated reasoning. It rather consists of *always postponing any serious thinking* during the specification and design phase; the system is always re-adapted and re-shaped according to the test results. But this, as is well known, is often far too late.

In other technologies, say avionics, it is certainly true that people do eventually test what they are constructing, but the testing is just the *routine confirmation* of a sophisticated design process rather than a guiding principle within it. In fact, most of the reasoning is done *before* the final object is built. It is performed on various blueprints, by analysing them using appropriate theories. We aim to emulate this for the construction of complex discrete systems.

VI. Overview of the B Method

It is time to outline the essential features of our blueprint based methodology, as it is understood in Event-B [2]. As with conventional blueprints, we shall use well defined notational conventions to write our models. We use the language of *classical logic* and *set theory*, familiar to most people having some mathematical background. There is a price for this choice: just as one cannot *drive* a blueprint of a car, one cannot in general *execute* a system model.

States and Transitions

Roughly speaking, a discrete model consists of a *state*, represented by some constants and variables at a particular level of abstraction with respect to the real system under study, and some *transitions*. The variables are like those used in conventional sciences (physics, biology, operational research) for studying natural systems.

The *transitions*, here called 'events,' have two pieces, the *guard* and the *action*. The guard, which is a predicate built on the state constants and variables, represents the conditions *necessary* for the event to occur. The action describes the way certain state variables are modified when the event occurs. This framework has been strongly influenced by the development of *Action Systems* [3].

A state together with some events on it thus constitutes a state transition machine, and we can give such a machine a simple *operational interpretation*. This interpretation should not be considered as providing the semantics of our models (which is given by means of suitable proof obligations), it is just given here to support their *informal understanding*.

First of all, the execution of an event, which describes a certain observable transition of the state variables, is considered to take *no time*, and no two events can occur simultaneously. The execution of an event is then the following:

- When no event guards are true, then the model execution stops: *it is said to have deadlocked*.

- When some event guards are true, exactly one of the corresponding events occurs and the state is modified accordingly. Subsequently the guards are checked again, and so on.

Evidently the above exhibits potential non-determinism (external non-determinism) as several guards may be true simultaneously. We make *no assumption* concerning the event which is executed among those whose guards are true. When only one guard is ever true, the model is said to be deterministic.

Notice that the termination of a model is *not at all mandatory*. As a matter of fact, most of the systems we study never deadlock: they run for ever.

Formal Reasoning

The simple transition machine we have described, although primitive, is nevertheless sufficiently powerful to allow us to do interesting formal reasoning. We focus on two kinds of model property.

The first kind of property is an *invariant property*. An invariant is a condition on the state variables that must hold permanently. In order to have an invariant, it is enough to *prove* that for each event, assuming the invariant and the event's guard, the invariant still holds after the event's action has been performed.

A second kind of property, a *modality*, need not hold permanently. We only consider a very special form of modality called *reachability*. What we would like to prove is that an event whose guard is not necessarily true now will nevertheless certainly occur within a certain finite time.

Managing the Complexity of Closed Models

Note that our models cannot just describe the control part of the intended system. They must also contain a representation of the environment with which our system will interact. Thus we must construct *closed models* which capture the actions and reactions between system models and environment models.

The number of transitions of such complex structures will be huge. How are we going to manage this complexity? The answer to this question lies in three linked concepts: *refinement*, *decomposition*, and *generic instantiation*.

Refinement, Decomposition, Generic Instantiation

Refinement allows us to build up a model *gradually* by making it more and more precise, that is, closer to reality. We typically construct an ordered sequence of embedded models, where each is a refinement of its predecessor. Usually, a refined, more concrete, model will have more variables than its abstraction. The new variables correspond to fresh detail made visible by seeing the system from closer up, or by viewing the system under a microscope and turning up the magnification.

Along with the *spatial extension*, there is a *temporal extension*, because there will be new transitions modifying the new variables. These are expressed using *new events* which involve the new variables alone. These new events refine implicit events doing nothing in the abstraction. This clean separation of new and old is vital in keeping under control the complexity of the reasoning needed to ensure correctness, as the complexity of the system builds up, layer by layer. Once the refinement process has captured all important properties, further *data-refinements* map the model onto executable data types.

By itself, refinement does not overcome the complexity problem, since the model can become too cumbersome to manage monolithically. At this point, it is necessary to cut up the single model into several (almost) independent pieces, via decomposition. The decomposition mechanism is designed so that it is always possible to reassemble the component models (which may themselves undergo further stages of independent refinement), to form a single model that is guaranteed to be a refinement of the original one.

Any development in the above style is parameterized by some carrier sets and constants enjoying a number of properties. Such a static model may be instantiated within a different development, by ensuring all the axioms of the static model are mere theorems in the instantiation. This in turn can save us redoing proofs already done in the more abstract context.

VII. Formal Methods in Industry: Two Cases

In this final section, we present two real projects where the B formal method has been used in a systematic fashion [4]. The two projects are separated by a nine year period: the first one resulted in a system (Line 14 of the Paris Métro) that has been working since October 1998, whereas the second one (the driverless shuttle at Paris Roissy Airport) will be operational in 2007. In both cases, not all the software for these systems was developed using B, only the *safety critical parts* were, representing one third of the overall program. Next is a table showing the main characteristics of these systems:

	Paris Métro Line 14	**Roissy Shuttle**
Line length	8.5 km	3.3 km
No. of stops	8	5
Inter-train time	115 s	105 s
Speed	40 km/h	26 km/h
No. of trains	17	14
Passengers / day	350,000	40,000

The information in this table has been taken from [5]. The formally developed part of the first system has been described in [6], while that of the second system is described in [7].

Since the Line 14 subway system is completely automatic, the safety critical part concerns the running and stopping of the trains, and the opening and closing of the train doors and the platform doors. The program is distributed into three different kinds of sub-system: the wayside equipment (several of these along the tracks), the on-board equipment (one instance in each train), and the line equipment (one instance), which are all heavily interconnected. In each sub-system, the safety parts which are developed using B, are sequential and cyclic (350 ms), and constitute a single non-interruptible task.

The Roissy Airport shuttle system is derived from the light shuttle system of Chicago's O'Hare Airport. The difference between the former and the latter, is that the former has a significant computerized part located along the tracks, called the Wayside Control Unit. There are several such units located along the tracks. They are linked by means of an Ethernet network. The Wayside Control Units drive the trains by sending them predefined speed programs which they have to follow. The programs are sent in response to the current state of the train which is detected by means of sensors situated on the track and connected to the Wayside Control Units.

The following table records a number of facts comparing the two projects. The most important data in are the first and the last rows. The first contains the number of lines in the two programs, whereas the last contains the time needed to perform the corresponding interactive proofs.

	Line 14	**Shuttle**
No. of ADA lines	86,000	158,000
No. of B lines	115,000	183,000
No. of proofs	27,800	43,610
Interactive proof percentage	8.1	3.3
Interactive proof effort (M-Mths)	7.1	4.6

The number of ADA lines represents the size of that part of the software system which was developed using the B 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 stated that in the second case study, a significant amount time was saved in the building of the Concrete Model. Such differences came from the use of a semi-automatic refining tool [8], [9]. Note that in both cases *no unit tests* were performed. They were replaced by some global tests which were all successful.

One important difference between the two case studies is that the RD of the first was created specially for it, whereas for the second, it was derived from an existing RD (that of the Chicago O'Hare Airport shuttle). In that second case, the RD had to be modified and extended in order to deal with the new requirements and functionality of the Roissy Airport shuttle. This caused a number of problems which were only discovered during the development of the model. Similar train control systems are presently being developed with these techniques for the New York City subway, the Barcelona subway, the Prague subway, and Line 1 of the Paris Métro.

VIII. Conclusions

In this paper, we have presented a brief panorama of formal methods in the B style, which relies on refinement and proofs to rigorously transform an abstract high level model into a correct implementation. It must be emphasised that the practicality of this approach is entirely dependent on the quality and power of tool support. As the case studies suggest, it would be impossible to complete such a large project in the time given, were it necessary to do everything manually. Worse, even if that were not so, there would be no confidence that the final result was correct — human beings are simply not capable of handling such quantities of low level (and ultimately rather featureless) mathematical detail without making mistakes. Work on tools for this methodology is still active. The European Project Rodin [10], which runs till September 2007, will create a new platform, implemented on Eclipse, for embedding the Event-B techniques described above.

Acknowledgments: This research was carried out as part of the EU research project IST 511599 RODIN (Rigorous Open Development for Complex Systems) [10]. I would like to thank Richard Banach for his help in preparing this paper.

References

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

[2] ——, *Event-B*, to be published.

[3] R. Back and R. Kurki-Suonio, "Decentralization of process nets with centralized control," in *Proc. 2nd. ACM SIGACT-SIGOPS Symp. on Distributed Computing*, 1989, pp. 131–142.

[4] J.-R. Abrial, "Formal methods in industry: Achievements, problems future," in *Proc. ACM/IEEE ICSE 2006 International Conference on Software Engineering*, 2006, pp. 761–768.

[5] Siemens, Siemens Transportation Systems http://www.siemens.fr/transportation/.

[6] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier, "Météor: A successful application of B in a large project," in *Proc. FM 1999, LNCS 1708*, 1999, pp. 369–387.

[7] F. Badeau and A. Amelot, "Using B as a high level programming language in an industrial project: Roissy val," in *Proc. ZB 2005, LNCS 3455*, 2005, pp. 334–354.

[8] L. Burdy, "Automatic refinement," in *Proc. BUGM at FM 1999*, 1999.

[9] D. Dollé, "Vital software: Formal method and coded processor," in *Proc. ERTS 2006* http://www.erts2006.org/, 2006.

[10] Rodin, European Project Rodin (Rigorous Open Development for Complex Systems) http://rodin.cs.ncl.ac.uk/.