Chapter I. Introduction

1 Motivation

The intent of this book is to give you some insights on *modeling* and *formal reasoning*. These activities are supposed to be performed *before* undertaking the effective coding of a computer system, so that the system in question will be *correct by construction*.

In this book, you will thus learn how to build models of programs and, more generally, discrete systems. But this will be done with *practice in mind*. For this we shall study a large number of examples coming from various sources of computer system developments: sequential programs, concurrent programs, distributed programs, electronic circuits, reactive systems, etc.

You will understand that the model of a program is quite different from the program itself. And you will learn that it is far easier to *reason* about the model than about the program. You will be made aware of the very important notions of *abstraction* and *refinement*: the idea being that an executable program is only obtained at the final stage of a sometimes long sequence consisting of gradually building more and more accurate models of the future program (think of the various blue-prints made by an architect).

We shall make it very clear what we mean by *reasoning* about a model. This will be done by using some simple mathematical methods, which will be presented to you, first by means of some examples, then by reviewing classical Logic (Propositional and Predicate Calculus) and Set Theory. You will understand the necessity of performing proofs in a very rigorous fashion.

You will also understand how it is possible to detect the presence of inconsistencies in your models just by the fact that some proofs cannot be done. The failure of the proof will provide you with some helpful clues on what is wrong or insufficiently defined in your model.

The formalism which we use throughout the book is called Event-B. It is a simplification as well as an extension of the B formalism [1] which has been developed ten years ago and which has been used in a number of large industrial projects [4], [3]. The formal concepts used in Event-B are by no means new. They have been proposed a long time ago in a number of parent formalisms such as Action Systems [6], TLA$^+$ [2], VDM [15], and UNITY [5].

The book is organized around examples. Each chapter contains a new example (sometimes several) together with the necessary formalism allowing you to understand the mathematical concepts one is using. Of course, such concepts are not repeated from one chapter to the other although they are sometimes made more precise. As a matter of fact, each chapter is an almost independent essay. The proofs done in each chapter have all been performed using the tools of the open source Rodin Platform [7] (see also the web site "event-b.org").

The book can be used as a textbook by presenting each chapter in one or more lectures. After giving a small summary of the various chapters in the next section, a possible usage of the book for an introductory as well as an advanced course will be proposed.

2 Presentation of the Chapters

Let us now list the various chapters of the book and give a brief outline of each of them.

2.1 Chapter 1: "Introduction"

The intent of this first (non-technical) chapter is to introduce you to the notion of a *formal method*. It also intends to make clear what we mean by *modeling*. We shall see what kind of systematic *conventions* we

shall use for modeling. But we shall also notice that there is no point in embarking into the modeling of a system without knowing what the requirements of this system are. For this, we are going to study how a *requirements document* has to be written.

2.2 Chapter 2: "Controlling Cars on a Bridge"

The intent of this chapter is to introduce the complete example of a small system development. We develop the model of a system controlling cars on a one way bridge between an island and the mainland. As an additional constraint, the number of cars in the island is limited. The physical equipment is made of traffic lights and car sensors

During this development, you will be made aware of the systematic approach we are using: it consists in developing a series of more and more accurate models of the system we want to construct. Note that each model does not represent the programming of our system using a high level programming language, it rather formalizes what an *external observer* of this system could perceive of it.

Each model will be analyzed and proved, thus enabling us to establish that it is *correct* relative to a number of criteria. As a result, when the last model will be finished, we shall be able to say that this model is *correct by construction*. Moreover, this model will be so close to a final implementation that it will be very easy to transform it into a genuine program.

The correctness criteria alluded above will be made completely clear and systematic by giving a number of *proof obligation rules* which will be applied on our models. After applying such rules, we shall have to prove formally a number of statements. To this end, we shall also give a reminder of the classical *rules of inference of the sequent calculus*. Such rules concern propositional logic, equality, and basic arithmetic. The idea here is to give the reader the possibility to manually prove the statements as given by the proof obligation rules. Clearly, such proofs could easily be discharged by theorem provers (as the ones used in the Rodin Platform), but we feel important at this stage that the reader exercise himself before using an automatic theorem prover. Notice that we do not claim that a theorem prover would perform these proofs the way it is proposed here: quite often, a tool does not work like a human being does.

2.3 Chapter 3: "A Mechanical Press Controller"

In this chapter, we develop again the controller of a complete system: a mechanical press. The intention is to show how this can be done in a systematic fashion in order to obtain correct final code. We first present, as usual, the requirement document of this system. Then we develop two general *design patterns* that we shall subsequently use. The development of these patterns will be made by using the proofs as a mean of discovering the invariants and the guards of the events. Finally, the main development of the mechanical press will take place.

In this chapter, we illustrate how the usage of formal design patterns can help doing systematic correct developments.

2.4 Chapter 4: "A Simple File Transfer Protocol"

The example introduced in this chapter is quite different from the previous ones, where the program was supposed to control an external situation (cars on a bridge or a mechanical press). Here we present a, so-called, protocol to be used on a computer network by two agents. This is the very classical two-phase handshake protocol. A very nice presentation of this example can be found in the book by L. Lamport [2]

This example will allow us to extend our usage of the mathematical language with such constructs as partial and total functions, domain and range of functions, and function restrictions. We shall also extend our logical language by introducing *universally quantified formulas* and corresponding inference rules.

2.5 Chapter 5: "The Event-B Notation and Proof Obligation Rules"

In the previous chapters we used the Event-B notation and the various corresponding proof obligation rules without introducing them initially in a systematic fashion. We presented them instead in the examples when they were needed. This was sufficient for the simple examples we studied because we used part of the notation and part of the proof obligation rules only. But it might not be adequate to continue like this when presenting more complicated examples in subsequent chapters.

The purpose of this chapter is thus to correct this. First, we present the Event-B notation as a whole, in particular the bits we did not use so far, and then we present all the proof obligation rules. This will be illustrated with a simple running example. Note that the mathematical justifications of the proof obligation rules shall be covered in chapter 14.

2.6 Chapter 6: "Bounded Re-transmission Protocol"

In this chapter, we extend the file transfer protocol example of chapter 4. The added constraint with regards to the previous simple example is that we suppose now that the channels situated between the two sites are *unreliable*. As a consequence, the effect of the execution of the bounded re-transmission protocol is to only *partially* copy a sequential file from one site to another. The purpose of this example is precisely to study how we can cope with this kind of problems dealing with fault tolerance and how we can formally reason about them. This example has been studied in many papers among which is the following one: [8].

Notice that in this chapter, we do not develop proofs as much as we did in the previous chapters: we only give some hints and let the reader developing the formal proof by himself.

2.7 Chapter 7: "Development of a Concurrent Program"

In previous chapters we have seen examples of *sequential* program developments (note that we shall come back to sequential program developments in chapter 15) and *distributed* program developments. Here we show how we can develop *concurrent* program developments. Such concurrent programs are different from distributed programs where various processes are executed on different computers in such a way that they *cooperate* (by exchanging messages in a well defined manner) in order to achieve a well specified goal. This was typically the case in the examples presented in chapters 4 and 6. It will also be the case in chapters 10, 11, 12, and 13.

In the case of concurrent programs, we also have different processes but this time they are usually situated on the same computer and they *compete* rather than cooperate in order to gain access to some shared resources. The concurrent programs do not communicate by exchanging messages (they ignore each other) but they can interrupt each other in a rather random way. We illustrate this approach by developing the concurrent program known to be Simpson's "4-slot Fully Asynchronous Mechanism" [14].

2.8 Chapter 8: "Development of Electronic Circuits"

In this chapter, we present a methodology to develop electronic circuits in a systematic fashion. In doing so, we can see that the Event-B approach is general enough to be adapted to different execution paradigms. The approach used here is similar to the one we shall use for developing sequential programs in chapter 15: the circuit is first defined by means of a single event doing the job "in one shot", then the initial very abstract transition is refined into several transitions until it becomes possible to apply some syntactic rules able to merge the various transitions into a single circuit.

2.9 Chapter 9: "Mathematical Language"

This chapter does not contain any examples as previous chapters did (except chapter 5). It rather contains the formal definition of the mathematical language we use in this book. It is made of four parts introducing successively the Propositional Language, the Predicate Language, the Set-theoretic Language, and the Arithmetic Language. Each of these languages will be introduced as an extension of the previous one.

Before introducing these languages however, we shall also give a brief summary of the Sequent Calculus. Here we shall insist a lot on the concept of proof.

At the end of the chapter, we present the way various classical but "advanced" concepts are formalized: transitive closure, various graph properties (in particular strong connectivity), lists, trees, and well-founded relations. Such concepts will be used in subsequent chapters.

2.10 Chapter 10: "Leader Election Protocol on a Ring-shaped Network"

In this chapter we study another interesting problem in distributed computation. We have a possibly large (but finite) number of agents, just not two as in the examples of chapters 4 and 6 (file transmission protocols). These agents are disposed on different sites that are connected by means of unidirectional channels forming a ring. Each agent is executing the same piece of codes. The distributed execution of all these identical programs should result in a *unique agent* being “elected the leader”. This example comes from a paper written by G. Le Lann in the seventies [9].

The purpose of this chapter is to learn more about modeling, in particular in the area of non-determinism. We shall also use more mathematical conventions such as the image of a set under a relation, the relational overriding operator, and the relational composition operator, conventions which have all been introduced in the previous chapter. Finally, we are going to study some interesting data structures: ring and linear list, also introduced in the previous chapter.

2.11 Chapter 11: "Synchronizing Processes on a Tree-shaped Network"

In the example presented in this chapter, we have a network of nodes, which is slightly more complicated than in the previous case where we were dealing with a ring. Here we have a tree. At each node of the tree, we have a process performing a certain task, which is the same for all processes (the exact nature of this task is not important). The constraint we want these processes to observe is that they remain *synchronized*. An additional constraint of our distributed algorithm states that each process can only communicate with its immediate neighbors in the tree. This example has been treated by many researchers [10], [11].

In this chapter, we shall encounter another interesting mathematical object: a tree. We shall thus learn how to formalize such a data structure and see how we can fruitfully reason about it using an induction rule. We remind the reader that this data structure has been introduced already in chapter 9.

2.12 Chapter 12: "Routing Algorithm for Mobile Agent"

The purpose of the example developed in this chapter is to present an interesting routing algorithm for sending messages to a mobile phone. In this example, we shall again encounter a tree structure as in the previous chapter, but this time the tree structure will be dynamically modified. We shall also see another example (besides the "Bounded Re-transmission Protocol" of chapter 6) where the usage of clocks will play a fundamental role. This example is taken from [12].

2.13 Chapter 13: "The IEEE 1394 Protocol: Leader Election on a Connected Graph Network"

The example presented in this chapter resembles the one which was presented in chapter 10: it is again a leader election protocol, but here the network is more complicated than a simple ring. More precisely, the goal of the IEEE-1394 protocol, [13], is to elect in a finite time a specific node, called the leader, in a network made of a finite number of nodes linked by some communication channels. This election is done in a distributed and non-deterministic way.

The network has got some specific properties. As a mathematical structure, it is called a *free tree*: it is a finite graph which is symmetric, irreflexive, connected, and acyclic. In this chapter, we shall thus learn how to deal and reason with such a complex data structure, which was already presented in chapter 9.

2.14 Chapter 14: "Mathematical Models for Proof Obligation Rules"

In this chapter, some mathematical justifications are presented to the proof obligation rules introduced in chapter 5. This is done by constructing some set-theoretic mathematical models based on the trace semantics of Event-B developments. We show that the proof obligation rules used in this book are equivalent to those dictated by the mathematical models of Event-B developed in this chapter.

2.15 Chapter 15: "Development of Sequential Programs"

This chapter is devoted entirely to the development of sequential programs. We shall first study the structure of such programs. They are made up of a number of assignment statements glued together by means of a number of operators: sequential composition, conditional and loop. We shall see how this can be modelled by means of simple transitions which are the essence of the Event-B formalism. Once such transitions are developed gradually by means of a number of refinement steps, we shall see how they can be put together using a number of merging rules whose nature is completely syntactic.

All this will be illustrated with many examples ranging from simple array and numerical programs to more complex pointer programs.

2.16 Chapter 16: "A Location Access Controller"

The purpose of this chapter is to study another example dealing with a complete system like the one we studied in chapters 2 and 3, where we controlled cars on a bridge or a mechanical press. We shall construct a system which will be able to control the access of certain people to different locations of a “workplace” : for example, a university campus, an industrial site, a military compound, a shopping mall, etc.

The system we study now is a little more complicated than the previous one. In particular, the mathematical data structure we are going to use is more advanced. Our intention is also to show that during the reasoning on the model we shall discover a number of important missing points in the requirement document.

2.17 Chapter 17: "Train System"

The purpose of this chapter is to show the specification and construction of a complete computerized system. The example we are interested in is called a *train system*. By this, we mean a system that is practically managed by a *train agent*, whose role is to control the various trains crossing part of a certain *track network* situated under his supervision. The computerized system we want to construct is supposed to help the train agent in doing this task.

This example presents an interesting case of quite complex data structures (the track network) whose mathematical properties have to be defined with great care: we want to show that this is possible.

This example also shows a very interesting case where the reliability of the final product is absolutely fundamental: several trains have to be able to safely cross the network under the complete automatic guidance of the software product we want to construct. For this reason, it will be important to study the bad incidents that could happen and which we want to either completely avoid or safely manage.

The software must take account of the external environment which is to be carefully controlled. As a consequence, the formal modelling we propose here will contain not only a model of the future software we want to construct but also a detailed model of its environment. Our ultimate goal is to have the software working in perfect synchronization with the external equipment, namely the track circuits, the points (the "switch" in American English), the signals, and also the train drivers. We want to *prove* that trains obeying the signals, set by the software controller, and then (blindly) circulating on the tracks whose points (switches) have been positioned, again by the software controller, that these trains will do so in a completely safe manner.

2.18 Chapter 18: "Problems"

This last chapter contains only problems which readers might try to do. Rather than spreading exercises and projects through each chapter of the book, we preferred to put them all in a single chapter.

All problems have to be performed with the Rodin Platform which, again, you can download from the web site "event-b.org".

Besides exercises (supposed to be rather easy) and projects (supposed to be larger and more difficult than exercises), we propose some mathematical developments which you can prove also with the Rodin Platform.

2.19 How to Use this Book

The material presented in this book has been used to teach various courses: essentially either introductory courses or advanced courses. Here is what can be proposed for these two categories of courses.

Introductory Course. The danger with such an introductory course is to present too much material. The risk is to have the attendees being completely overwhelmed. What can be presented then is the following:

- chapter 1 (introduction),
- chapter 2 (cars on a bridge),
- chapter 3 (mechanical press),
- chapter 4 (simple file transfer),
- some parts of chapter 5 (Event-B notation),
- some parts of chapter 9 (mathematical language),
- some parts of chapter 15 (sequential program development),

The idea is to avoid encountering complex concepts, only simple mathematical constructs: propositional calculus, arithmetic, and simple set-theoretic constructs.

Chapter 2 (cars on a bridge) is important because the example is extremely easy to understand and the basic notions of Event-B and of classical logic are introduced by means of that simple example. However, one has to be careful to present this chapter very slowly, doing carefully the proofs with the students

because they are usually very disturbed when they encounter this kind of material for the first time. In this example the data structures are very simple: numbers and booleans.

Chapter 3 (mechanical press) shows again a complete development. It is simple and the usage of formal design patterns is helpful to construct the controller in a systematic fashion.

Chapter 4 (simple file transfer) allows to present a very simple distributed program. Students will learn how this can be specified and later refined in order to obtain a very well known distributed protocol. They have to understand that such a protocol can be constructed by starting from a very abstract (non-distributed) specification, which is gradually distributed among various (here two) processes. This example contains some more elaborated data structures than those used in the previous chapter: intervals, functions, restrictions.

Chapter 5 (Event-B notation) contains a summary of the Event-B notation and of the proof obligation rules. It is important that the students see that they use a well-defined although simple notation which is given a mathematical interpretation through the proof obligation rules. It is not necessary however to go too deeply into fine details in such an introductory course.

Chapter 9 (mathematical language) allows one to depart a bit from the examples. It is a refresher of mathematical concepts done in the middle of the course. The important aspect here is to have the students becoming more familiar with proofs done on set-theoretic concepts. Students have to be given a number of exercises for translating set-theoretic constructs into predicate calculus. It is not necessary to cover this chapter from beginning to end.

Chapter 15 (sequential program development) is to be done partly in an introductory course because people are used to write programs. They have to understand that programs can be constructed in a systematic fashion. They have to understand eventually the distinction between formal program construction (which we do here) versus program verification (where the program is "proved" once developed). Some of the example must be avoided in an introductory course, namely those dealing with pointers which are too difficult.

At the end of the course, students should be comfortable with the notions of abstraction and refinement. They should also be less afraid by doing formal proofs of simple mathematical statements. Finally, they should be convinced that it is possible to develop programs which work first time!

Students could be made aware of the Rodin Platform tool [7] which are devoted to Event-B. But we think that they must first do some proofs by hand in order to understand what the tool is doing.

Advanced Course. Here we suppose that the students have already attended the introductory course. In this case, it is not necessary to repeat the presentation of chapters 2 and 3. However, students will be encouraged to read them again. The course then consists in presenting all the other chapters.

It is important for the students to understand that the same Event-B approach can be used to model systems with very different execution paradigms: sequential, distributed, concurrent, and parallel.

Students should be comfortable in reasoning with complex data structures: list, trees, DAGs, arbitrary graphs. They must understand that set theory allows you to build very complex data structures. For these reasons, the examples presented in chapters 11 (synchronizing processes in a tree), 12 (mobile agent), 13 (IEEE protocol), and 17 (train system) are all important.

In this course, students should not do manual proofs any more as was the case in the previous introductory course. They must use a tool such as the Rodin Platform which is specially devoted to Event-B and associated plugins [7].

3 Formal Methods

The term "formal method" leads nowadays to a *great confusion* because its usage has been enlarged to many different activities. Some typical questions we can ask about such methods are the following: Why use formal methods? What are they used for? When do we need to use such methods? Is UML a formal method? Are they needed in object oriented programming? How can we define formal methods?

I will answer these questions gradually. Formal methods have to be used by people who have recognized that the (internal) *program development process* they use is inadequate. There may be several reasons for such inadequacies: e.g. failure, cost, risk.

The choice of a formal method is not an easy one. Partly because there are many formal method vendors. More precisely, the adjective "formal" does not mean anything. Here are some questions you may ask to a formal method vendor. Is there any theory behind your formal method? What kind of language is your formal method using? Does there exist any kind of refinement mechanism associated with your formal method? What is your way of reasoning with your formal method? Do you prove anything when using your formal method?

People might claim that using formal methods is impossible because there are some intrinsic difficulties in doing so. 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, arrows are missing). People will not be able to perform proofs.

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

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

2. The usage of formal methods has to be incorporated within a certain development process, and this incorporation is not easy. In industry, people develop their products under very precise guidelines, which they have to follow very carefully. Usually, the introduction of such guidelines in an industry takes a significant time before being accepted and fully observed by engineers. Now, changing such guidelines to incorporate the usage of formal methods in them is something that managers are very reluctant to do because they are afraid of the time and cost this process modification will take.

3. Model building is not a simple activity: remember that this is what you will learn in this book. One has to be careful not to confuse modeling and programming. Sometimes people do some kind of pseudo-programming instead of modeling. More precisely, the initial model of a program describes the properties that the program must fulfil. It does not describe the algorithm contained in the program but rather the way by which we can eventually judge that the final program is correct. For example, the initial model of a file sorting program does not explain how to sort. It rather explains what the properties of a sorted file are and which relationship exists between the initial non-sorted file we want to sort and the final sorted one.

4. Modeling has to be accompanied by reasoning. In other words, the model of a program is not just a piece of text, whatever the formalism being used. It also contains proofs that are related to this text. For many years, formal methods have just been used as a means of obtaining abstract descriptions of the program we wanted to construct. Again, descriptions are not enough. We must justify what we write by proving some consistency properties. Now the problem is that software practitioners are not used to constructing such proofs, whereas people in other engineering disciplines are far more familiar with doing so. And one of the difficulties to make this become part of the daily practice of software engineers is the lack of good proving tool support for proofs, which could be used on a large scale.

5. Finally, one important difficulty encountered in modeling is the very frequent lack of good requirement documents associated with the programming job we have to perform. Most of the time, the requirement document which can be found in industry are either almost inexistent or far too verbose. In my opinion, it is vital, most of the time, to completely re-write such documents before starting any modeling. We shall come back to this point in what follows.

4 A Little Detour: Blue-prints

It is my belief that the people in charge of the development of large and complex computer systems should adopt a point of view shared by all mature engineering disciplines, namely that of *using an artifact to reason about their future system during its construction*. In these disciplines, people use *blue-prints* in the wider sense of the term, which allows them to reason formally during the very construction process. Here are a number of mature engineering disciplines: Avionics, civil engineering, mechanical engineering, train control systems, ship building, etc. In these disciplines, people use blue-prints and they consider these as very important parts of their engineering activity.

Let us analyze for a while what a blue-print is. A blue-print is a certain representation of the future system. It is not a mock-up however because the basis is lacking: you cannot drive the blue-print of a car! The blue-print allows you to reason about the future system you want to construct during its very construction process.

Reasoning about a future system means defining and calculating its behavior and its constraints. It also allows you to construct an architecture gradually. It is based on some dedicated underlying theories: strength of material, fluid mechanics, gravitation, etc.

It is possible to use a number of "blue-printing" techniques which we are going to review now. While blue-printing, one is using a number of pre-defined conventions, which helps reasoning but also allows to share blue-prints among large communities. Blue-prints are usually organized as sequences of more and more accurate versions (again think of the blue-prints made by architects) where each more recent version is adding details which could not be visible in previous ones. Likewise, blue-prints can be decomposed into smaller ones in order to enhance readability. It is also possible for some early blue-prints to be not completely determined, thus leaving open options which will be later refined (in further blue-prints). Finally, it is very interesting to have libraries of old blue-prints where the engineer can browse in order to re-use some work that has already been done. All this (refinement, decomposition, re-use) clearly requires that blue-prints are used with care so that the entire blue-print development of a system is coherent. For example, one has to be sure that a more accurate blue-print does not contradict a previous less precise one.

Most of the time, in our engineering discipline of software construction, people do not use such blue-printing artifacts. This results in a very heavy testing phase on the final product, which is well known to happen quite often too late. The blue-print drawing of our discipline consists of *building models* of our future systems. In no way is the model of a program the program itself. But the model of a program and more generally of a complex computer system, although not executable, allows you to clearly identify the properties of the future system and to prove that they will be present in it.

5 The Requirement Document

The blue-print we have quickly described in the previous section is not however the initial phase of the development process. It is preceded by a very important one which consists of writing a, so-called, *requirement document*. Most of the time such a document is either missing or very badly written. This is the reason why we are going to dwell for a while on this question and try to give an adequate answer to it.

5.1 Life Cycle

First, we are going to recall what is the place of this activity, namely that of the requirement document writing, within the life cycle of a program development. Here is a rough list of the various phases of the life cycle: system analysis, *requirement document*, technical specification, design, implementation, tests, maintenance.

Let us briefly summarize what the contents of these phases are. The system analysis phase contains the preliminary feasibility studies of the system we want to construct. The requirement document phase clearly states what the functions and constraints of the system are. It is mostly written in natural language. The technical specification contains the structured formalization of the previous document using some modeling techniques. The design phase develops the previous one by taking and justifying the decisions which implement the previous specification and also defines the architecture of the future system. The implementation phase contains the translation of the outcome of the previous phase into hardware and software components. The test phase consists of the experimental verifications of the final system. The maintenance phase contains the system upgrading.

As noticed above, the requirement document phase is quite often a *very weak point* in this life cycle. This results in lots of difficulties in subsequent phases. In particular, the famous syndrome of the inescapable specification changes occurring during the design phases originates in the weakness of the requirement document. When such a document is well written, these kinds of difficulties tend to disappear. This is the reason why it is so important to see how this phase can be improved.

5.2 Difficulties with the Requirement Document

Writing a good requirement document is a difficult task. We have to remember that the readers of such a document are the people who are conducting the next phases, namely technical specification and design. It is usually very difficult for them to exploit the requirement document because they cannot clearly identify what they have to take into account and in which order.

Quite often too, some important points are missing in the requirement document. I have seen a huge requirement document for the alarm system of an aircraft where the simple fact that this system should not deliver false alarms was simply missing. When the authors of this document were interrogated on this missing point, the answer they gave was rather surprising: it was not necessary to put such a detail in the requirement document because "of course everybody knows that the system should not deliver any false alarm." Sometimes, on the contrary, the requirement document is over-specified with a number of irrelevant details.

What is difficult for the reader of the requirement document is to make a clear distinction between which part of the text is devoted to *explanations* and which part is devoted to genuine *requirements*. Explanations are needed initially for the reader to understand the future system. But when the reader is more acquainted with the purpose of the system, explanations are less important. At that time, what counts is to remember what the real requirements are in order to know exactly what has to be taken into account in the system to be constructed.

5.3 A Useful Comparison

There exists other documents (rather books) which also contain explanations and, in a sense, requirements. These are books of mathematics. The "requirements" are Definitions and Theorems. Such items are usually easily recognizable because they are labeled by their function (definition, lemma, theorem), numbered in a systematic fashion, and usually written with a font which differs from that used elsewhere in the book. Here is an example:

2.8 The Cantor-Bernstein Theorem. *If* $a \preceq b$ *and* $b \preceq a$ *then* a *and* b *are equinumerous.*

This theorem was first conjectured by Cantor in 1895, and proved by Bernstein in 1898.

Proof. Since $b \preceq a$, then a has a subset c such that $b \approx c$. ...
...
□

In this quotation extracted from a book of mathematics, we can clearly see the "requirement" as indicated on the first line: the theorem number, the theorem name, and the theorem statement (written in italic). Next are the associated "explanations": historical comments and proof.

This distinction is extremely interesting and useful for the reader. If it is our first contact with this material, then the explanation is fundamental. Later, we might only be interested to just have a look at the precise statement of the theorem but we are not interested any more by the historical comments and even by the proof. There are some books of mathematics where the "requirements", that is the definitions and the theorems, are summarized at the end of the book in an appendix, which can be conveniently consulted.

5.4 Structuring the requirement document

Following this analogy of a book of mathematics, the idea is to have our requirement document 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 *labeled and numbered short statements* written using natural language, which must be very easy to read independently from the explanatory text. For this, we shall use a special font for the reference text. These fragments must be self contained without the explanations. They together form the requirements. The explanations are just there to give some comments which could help a first reader. But after an initial period, the reference text is the only one that counts.

The labels of the requirement fragments are very important. It may vary from one system to the other. Common labels are the following:

FUN: for functional requirements,

ENV: for environment requirements,

SAF: for safety properties,

DEG: for degraded mode requirements,

DEL: for requirements concerned with delays,

etc.

An important activity to be undertaken before writing the requirement document is that of defining with care the various labels we are going to use. Numbering these requirements is also very important as they will be referenced in later stages of the development. This is called *traceability*. The idea is to have these labeled numbers appearing in later stages (technical specification, design, even implementation) so that it will be easy to recognize how each requirement has indeed been taken into account during the construction of our system and in its final operational version.

Most of the time, the requirement fragments are made of short statements. But we might also have other styles: date description tables, transition diagrams, mathematical formulae, physical unit tables, figures, etc.

The order and more generally, the structure of the entire requirement document is not so important at this stage. This will be taken care of in later development phases.

6 Definition of the Term "Formal Method" as Used in this Book

Formal methods are techniques used to build blue-prints adapted to our discipline. Such blue-prints are called *formal models*.

As for real blue-prints, we shall use some *pre-defined conventions* to write our models. There is no point in inventing a new language: we are going to use the language of *classical logic* and *set theory*. These conventions will allow us to easily communicate our models to others as these languages are known by everyone having some mathematical backgrounds. The usage of such a mathematical language will allow us to do some reasoning in the form of mathematical proofs, which we shall conduct as usual.

Note again that, as with blue-prints, the basis is lacking: *our model will thus not in general be executable*.

The kind of systems we are interested in developing are *complex* and *discrete*. Let us develop these two ideas for a while.

6.1 Complex Systems

Here is the kind of questions we might ask to begin with. 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.? Does there exist any kind of unified approach to in depth study and formally prove the requirements, the specification, the design and the implementation of *systems* that are so different in size and purpose?

We shall only give for the moment a very general answer. Almost all such systems are *complex* in that they are made of many parts interacting with a highly evolving and sometimes hostile environment. They also quite often involve several concurrent executing agents. They require a high degree of correctness. Finally, most of them are the result of a construction process which is spread over several years and which requires a large and talented team of engineers and technicians.

6.2 Discrete Systems

Although their behavior is certainly ultimately continuous, the systems listed in the previous section operate most of the time operating in a *discrete fashion*. This means that their behavior can be faithfully *abstracted* by a succession of steady states intermixed with jumps which cause sudden state changes. Of course, the number of such possible changes is enormous, and they are occurring in a concurrent fashion at an unthinkable frequency. But this number and this high frequency do not change the very nature of the problem: such systems are intrinsically discrete. 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 envisaged above are pure programs. In other words, their transitions are essentially concentrated in *one medium* only. The electronic circuit and the sorting program clearly fall into this category. Most of the other examples however are far more complex than just pure programs because they involve many different executing agents and also a heavy interaction with their environment. This means that the transitions are executed by different kinds of entities acting concurrently. But, again, this does not change the very discrete nature of the problem, it only complicates matters.

6.3 Test Reasoning versus Model (Blue Print) Reasoning

A very important activity, at least in terms of time and money, concerned with the construction of such complex discrete systems is certainly that of verifying that the final implementations are operating in a,

so called, *correct* fashion. Most of the time nowadays, this activity is realized during a very heavy testing phase, which we shall call a "laboratory execution".

The validation of a discrete system by means of such "laboratory executions" is certainly far more complicated to realize, if not impossible in practice, on the multiple medium case than in the single medium one. And we already know that program testing, used as a validation process in almost all programming projects, is by far an incomplete process. Not so much, in fact, because of the impossibility to achieve a total cover of all executing cases. The incompleteness is rather, for us, the consequence of the very often *lack of oracles* which would give, *beforehand* and independently of the tested objects, the expected results of a future testing session.

It is nevertheless the case that today the basic ingredients for complex system construction still are a very small design team of smart people, managing an army of implementers, eventually concluding the construction process with a long and heavy testing phase. And it is a well known fact that the testing cost is at least twice that of the pure development effort. Is this a reasonable attitude nowadays? Our opinion is that a technology using such an approach is still in its infancy. This was the case at the beginning of last century for some technologies, which have now reached a more mature status (e.g. avionics).

The technology we consider in this short presentation is concerned with the construction of *complex discrete systems*. As long as the main validation method used is testing, we consider that this technology will remain in an underdeveloped state. Testing does not involve any kind of sophisticated reasoning. It rather consists of *always postponing any serious thinking* during the specification and design phase. The construction of the system will always be re-adapted and re-shaped according to the testing results (trial and error). But, as one knows, it is quite often too late.

In conclusion, testing always gives a shortsighted operational view of the system under construction: that of execution. In other technologies, say again avionics, it is certainly the case that people eventually do test what they are constructing, but the testing is just the *routine confirmation* of a sophisticated design process rather than a fundamental phase in it. As a matter of fact, most of the reasoning is done *before* the very construction of the final object. It is performed on various blue prints, in the broad sense of the term, by applying on them some well defined practical theories.

The purpose of this book is to incorporate such a "blue print" approach in the design of complex discrete systems. It also aims at presenting a theory which is able to facilitate the elaboration of some *proved reasoning* on such blue prints. Such reasoning will thus take place far before the final construction. In the present context, the "blue prints" are called *discrete models*. We shall now give a brief informal overview of the notion of discrete models.

7 Informal Overview of Discrete Models

In this section, we give an informal description of discrete models. A discrete model is made of a state and a number of transitions. For the sake of understanding, we then give an operational interpretation of discrete models. We then present the kind of formal reasoning we want to express. Finally, we shortly address the problem of mastering the complexity of models by means of three concepts: refinement, decomposition, and generic development.

7.1 State and Transitions

Roughly speaking, a discrete model is made of a *state* represented by some constants and variables at a certain level of abstraction with regards to the real system under study. Such variables are very much the same as those used in applied sciences (physics, biology, operational research) for studying natural systems. In such sciences, people also build models. It helps them inferring some laws in reality by means of some reasoning, which they undertake on these models.

Besides the state, the model also contains a number of *transitions* that can occur under certain circumstances. Such transitions are called here "events". Each event is first made of a *guard*, which is a predicate built on the state constants and variables. It represents the *necessary* conditions for the event to occur. Each event is also made of an *action*, which describes the way certain state variables are modified as a consequence of the event occurrence.

7.2 Operational Interpretation

As can be seen, a discrete dynamical model thus indeed constitutes a kind of state transition machine. We can give such a machine an extremely simple *operational interpretation*. Notice that such an interpretation should not be considered as providing any operational semantics to our models (this will be given later by means of a proof system), 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*. Moreover, no two events can occur simultaneously. The execution 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, then one of the corresponding events necessarily occurs and the state is modified accordingly, subsequently the guards are checked again, and so on.

This behavior clearly shows some possible non-determinism (called external non-determinism) as several guards may be true simultaneously. We make *no assumption* concerning the specific event which is indeed executed among those whose guards are true. When only one guard is true at all times, the model is said to be deterministic.

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

7.3 Formal Reasoning

The very elementary transition machine we have described in the previous section, although primitive, is nevertheless sufficiently elaborate to allow us to undertake some interesting formal reasoning. In the following, we envisage two kinds of discrete model properties.

The first kind of properties that we want to prove about our models, and hence ultimately about our real systems, are, so called, *invariant properties*. An invariant is a condition on the state variables that must hold permanently. In order to achieve this, it is just required to *prove* that, under the invariant in question and under the guard of each event, the invariant still holds after being modified according to the action associated with that event.

We might also consider more complicated forms of reasoning involving conditions which, in contrast with the invariants, do not hold permanently. The corresponding statements are called *modalities*. In our approach, 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 necessary true now will nevertheless certainly occur within a certain finite time.

7.4 Managing the Complexity of Closed Models

Note that the models we are going to construct will not just describe the control part of our intended system. It will also contain a certain representation of the environment within which the system we build

is supposed to behave. In fact, we shall quite often essentially construct *closed models* which are able to exhibit the actions and reactions taking place between a certain environment and a corresponding, possibly distributed, controller.

In doing so, we shall be able to insert the model of the controller within an abstraction of its environment, which is formalized as yet another model. The state of such a closed system thus contains physical variables, describing the environment state, as well as logical variables, describing the controller state. And, in the same way, the transitions will fall into two groups: those concerned by the environment and those concerned by the controller. We shall also have to put into the model the way these two entities communicate.

But as we mentioned earlier, the number of transitions in the real systems under study is certainly enormous. And, needless to say, the number of variables describing the state of such systems is also extremely large. How are we going to practically manage such a complexity? The answer to this question lies in three concepts: *refinement* (section 7.5), *decomposition* (section 7.6), and *generic instantiation* (section 7.7). It is important to notice here that these concepts are linked together. As a matter of fact, one refines a model to later decompose it, and, more importantly, one decomposes it to further refine it more freely. And finally, a generic model development can be later instantiated, thus saving the user of redoing similar proofs.

7.5 Refinement

Refinement allows us to build a model *gradually* by making it more and more precise, that is closer to the reality. In other words, we are not going to build a single model representing once and for all our reality in a flat manner: this is clearly impossible due to the size of the state and the number of its transitions. It would also make the resulting model very difficult to master, if not just to read. We are rather going to construct an ordered sequence of embedded models, where each of them is supposed to be a refinement of the one preceding it in that sequence. This means that a refined, more concrete, model will have more variables than its abstraction: such new variables are visible consequence of a look at our system from closer range.

A useful analogy here is that of the scientist looking through a microscope. In doing so, the reality is the same, the microscope does not change it, *our look at it is only more accurate*: some previously invisible parts of the reality are now revealed by the microscope. An even more powerful microscope will reveal more parts, etc. A refined model is thus one which is spatially larger than its previous abstractions.

And analogously to this *spatial extension*, there is a corresponding *temporal extension*: this is because the new variables are now able to be modified by some transitions, which could not have been present in the previous abstractions simply because the concerned variables did not exist in them. Practically, this is realized by means of *new events* involving the new variables only. Such new events refine some implicit events doing nothing on the abstraction. Refinement will thus result in a discrete observation of our reality, which is now performed using a *finer time granularity*.

Refinement is also used in order to modify the state so that it can be implemented on a computer by means of some programming language. This second usage of refinement is called *data-refinement*. It is used as a second technique, once all important properties have been modelled.

7.6 Decomposition

Refinement does not solve completely the problem of the complexity. As a model is more and more refined, the number of its state variables and that of its transitions may augment in such a way that it becomes impossible to manage them as a whole. At this point, it is necessary to cut our single refined model into several almost independent pieces.

Decomposition is precisely the process by which a single model can be split into various component models in a systematic fashion. In doing so, we reduce the complexity of the whole by studying, and thus refining, each component model independently of the others. The very definition of such a decomposition implies that independent refinements of the component models could always be put together again to form a single model that is guaranteed to be a refinement of the original one. This decomposition process can be further applied on the components, and so on. Note that the component model could already exist and be developed, thus allowing to mix a top down and a bottom up approach.

7.7 Generic Development

Any model development done by applying refinement and decomposition, is parameterized by some carrier sets and constants defined by means of a number of properties.

Such a generic model may be instantiated within another development in the same way as a mathematical theory like, say, group theory, can be instantiated in a more specific mathematical theory. This can be done providing that one has been able to prove that the axioms of the abstract theory are mere theorems in the second one.

The interest of this approach of generic instantiation is that it saves us redoing the proofs already done in the abstract development.

References

1. J.R. Abrial. *The B-book: Assigning Programs to Meanings.* CUP 1996
2. L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley 1999.
3. F. Badeau. *Using B as a high level programming language in an industrial project: Roissy val.* In *Proceedings of ZB'05*, 2005.
4. P. Behm. *Meteor: A successful application of B in a large project.* In *Proceedings of FM'99*, 1999.
5. K.M. Chandy and J. Misra. *Parallel Program Design, a Foundation.* Addison-Wesley, 1988.
6. R.J. Back and R. Kurki-Suonio. *Distributed Cooperation with Action Systems. ACM Transaction on Programming Languages and Systems.* 10(4): 513-554, 1988.
7. Rodin. *European Project Rodin.* http://rodin.cs.ncl.ac.uk.
8. J.F. Groote and J.C. Van de Pol *A bounded retransmission protocol for large data packets - a case study in computer checked algebraic verification.* Algebraic Methodology and Software Technology, 5th International Conference AMAST '96, Munich. Lecture Notes in Computer Science 1101.
9. G. Le Lann. *Distributed systems - towards a formal approach*. In B Gilchrist, editor *Information Processing 77* North-Holland 1977.
10. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers 1996.
11. W.H.J. Feijen and A.J.M. van Gasteren. *On a Method of Multi-programming* Springer 1999.
12. L. Moreau. *Distributed Directory Service and Message Routers for Mobile Agent*. Science of Computer Programming 39(2-3):249-272, 2001.
13. IEEE *Standard for a High Performance Serial Bus.* Std 1394-1995, August 1995.
14. H.R. Simpson. *Four-slot Fully Asynchronous Communication Mechanism*. Computer and Digital Techniques. IEE Proceedings. Vol 137 (1) (Jan 1990).
15. C.B. Jones. *Systematic Software Development using VDM*. Prentice Hall International (1990).