1 Introduction

1.1 Motivation

The intent of this book is to give 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, we 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 development: sequential programs, concurrent programs, distributed programs, electronic circuits, reactive systems, etc.

We will understand that the model of a program is quite different from the program itself. And we will learn that it is far easier to *reason* about the model than about the program. We 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 blueprints 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 first by means of some examples then by reviewing classical logic (propositional and predicate calculus) and set theory. We will understand the necessity of performing proofs in a very rigorous fashion.

We will also understand how it is possible to detect the presence of inconsistencies in our models just by the fact that some proofs cannot be done. The failure of the proof will provide us with some helpful clues about what is wrong or insufficiently defined in our model. We will use such tools and see how easy it is to perform proofs with a computer.

The formalism we use throughout the book is called Event-B. It is a simplification as well as an extension of the B formalism [1] which was 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 were proposed a long time ago in a number of parent formalisms, such as Action Systems [6], TLA$^+$ [2], and UNITY [5].

The book is organized around examples. Each chapter contains a new example (sometimes several) together with the necessary formalism allowing the mathematical concepts being used to be understood. 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 website "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 use for the book in an introductory as well as an advanced course will be proposed.

1.2 Overview of the chapters

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

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 on 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.

Chapter 2: Controlling cars on a bridge

The intent of this chapter is to introduce a 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 on the island is limited. The physical equipment is made of traffic lights and car sensors

During this development, we 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 is 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 to above will be made completely clear and systematic by giving a number of *proof obligation rules*, which will be applied to 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 opportunity 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 it important at this stage that the reader takles these proofs 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.

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 the correct final code. We first present, as usual, the requirement document of this system. Then we develop two general *design patterns* which we shall subsequently use. The development of these patterns will be made by using the proofs as a means 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 tackling systematic correct developments.

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.

Chapter 5: The Event-B Modeling 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 in a systematic fashion. We presented them instead in the examples when they were needed. This was sufficient for the simple examples studied so far 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 parts we have not used so far, 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 will be covered in Chapter 14.

Chapter 6: Bounded re-transmission protocol

In this chapter, we extend the file transfer protocol example of Chapter 4. The added constraint with regard to the previous simple example is that we now suppose 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 problem, i.e. dealing with fault tolerances and how we can formally reason about them. This example has been studied in many papers among which is [8].

Notice that, in this chapter, we do not develop proofs to the extent we did in the previous chapters, we only give some hints and let the reader develop the formal proof.

Chapter 7: Development of a concurrent program

In previous chapters, we saw 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 co-operate 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].

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.

Chapter 9: Mathematical language

This chapter does not contain any examples as in previous chapters (except Chapter 5). It rather contains the formal definition of the mathematical language we use in this book. It is made up of four sections 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 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.

Chapter 10: Leader election 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, not just 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 coding. 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 1970s [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.

Chapter 11: Synchronizing 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 already been introduced in Chapter 9.

Chapter 12: Routing algorithm for a 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].

Chapter 13: Leader election on a connected graph network

The example presented in this chapter resembles the one 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.

Chapter 14: Mathematical models for proof obligations

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.

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 modeled 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, the nature of which is completely syntactic.

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

Chapter 16: A location access controller

The purpose of this chapter is to study another example dealing with a complete system such as the one we studied in Chapters 2 and 3, where we controlled cars on a bridge and 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 now study is a little more complicated than the previous ones. In particular, the mathematical data structure we are going to use is more advanced. Our intention is also to show that during the reasoning of the model, we shall discover a number of important missing points in the requirements document.

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) where 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 modeling 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 (or "switch"), 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 where the points (switches) have been positioned, again by the software controller, will do so in a completely safe manner.

Chapter 18: Problems

This last chapter contains only problems which readers might try to tackle. 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, can be downloaded 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 can also be proved with the Rodin Platform.

1.3 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, we have to be careful to present this chapter very slowly, doing carefully the proofs with the students because they are usually very confused 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 us 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 us to depart a bit from the examples. It is a refresher of the mathematical concepts in the middle of the course. The important aspect here is to have the students becoming more familiar with proofs undertaken in 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 partly an introductory course because students are used to writing programs. It is important to understand that programs can be constructed in a systematic fashion; 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 examples must be avoided in an introductory course, namely those dealing with pointers that 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 of tackling formal proofs of simple mathematical statements. Finally, they should be convinced that it is possible to develop programs that work first time!

Students could be made aware of the Rodin Platform tool [7], which is 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 presentations 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 reasoning with complex data structures: list, trees, DAGs, arbitrary graphs. They must understand that set theory allows us 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].

1.4 Formal methods

The term "formal method" leads nowadays to *great confusion* because its usage has been enlarged to cover 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?

We will look at these questions gradually. Formal methods have to be used by people who have recognized that the (internal) *program development process* they use