Tutorial on the event-based B method

Dominique Cansell, Dominique Méry

Tutorial on the event-based B method

Concepts and Case Studies

FORTE 2006
26th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems
September 26-29 2006, Paris, France

Dominique Cansell, Université de Metz
and
Dominique Méry, Université Henri Poincaré Nancy 1

LORIA & Université Henri Poincaré Nancy 1
BP 239,Campus Scientifique
54506 Vandoœuvre-lès-Nancy Cedex, FRANCE
cansell,mery@loria.fr
Edited on September 11, 2006

On the lectures

B is a method for specifying, designing and coding software systems. It is based on Zermelo-Fraenkel set theory with the axiom of choice, the concept of generalized substitution and on structuring mechanisms (machine,refinement, implementation). The concept of refinement is the key notion for developing B models of (software) systems in an incremental way. B models are accompanied by mathematical proofs that justify them. Proofs of B models convince the user (designer or specifier) that the (software) system is effectively correct. We provide a survey of the underlying logic of the B method and the semantic concepts related to the B method; we detail the B development process partially supported by the mechanical engine of the prover. We can not talk or write on the B method without mentioning J.-R. Abrial, who has first invented the Z specification language and then has invented the B method; Both Dominiques have worked on joint works with Jean-Raymond and have been published; our lectures notes are produced from our lectures at the universities of Lorraine, joint works with Jean-Raymond. We encourage readers to get the B book, which is the seminal book on the B method and which contains a lot of *B things* and we advertise the forthcoming book on the event-based B method. Tools are available at www.b4free.com for B4free and at www.loria.fr/ cansell/cnp.html for Click'n'prove. Please, get and play.

Contents

Chapter 1

Introduction

1.1 Overview of B

Classical B is a state-based method developed by Abrial for specifying, designing and coding software systems. It is based on Zermelo-Fraenkel set theory with the axiom of choice. Sets are used for data modelling, *Generalised Substitutions* are used to describe state modifications, the refinement calculus is used to relate models at varying levels of abstraction, and there are a number of structuring mechanisms (machine, refinement, implementation) which are used in the organisation of a development. The first version of the B method is extensively described in The B-Book [21]. It is supported by the Atelier B tool [55] and by the B Toolkit [77].
Central to the classical B approach is the idea of a software operation which will perform according to a given specification if called within a given pre-condition. Subsequent to the formulation of the classical approach, Abrial and others have developed a more general approach in which the notion of *event* is fundamental. An event has a firing condition (a guard) as opposed to a pre-condition. It may fire when its guard is true. Event based models have proved useful in requirement analysis, modelling distributed systems and in the discovery/design of both distributed and sequential programming algorithms.
After extensive experience with B, current work by Abrial is proposing the formulation of a second version of the method [8]. This distills experience gained with the event based approach and provides a general framework for the development of *discrete systems*. Although this widens the scope of the method, the mathematical foundations of both versions of the method are the same.

1.2 Proof-based Development

Proof-based development methods [25, 21, 83] integrate formal proof techniques in the development of software systems. The main idea is to start with a very abstract model of the system under development. Details are gradually added to this first model by building a sequence of more concrete ones. The relationship between two successive models in this sequence is that of *refinement* [25, 21, 52, 24]. The essence of the refinement relationship is that it preserves already proved *system properties* including safety properties and termination.
A development gives rise to a number of, so-called, *proof obligations*, which guarantee its correctness. Such proof obligations are discharged by the proof tool using automatic and interactive proof procedures supported by a proof engine [55, 56].
At the most abstract level it is obligatory to describe the static properties of a model's data by means of an *invariant* predicate. This gives rise to proof obligations relating to the consistency of the model. They are required to ensure that data properties which are claimed to be invariant are preserved by the events or operations of the model. Each refinement step is associated with a further invariant which relates the data of the more concrete model to that of the abstract model and states any additional invariant properties of the (possibly richer) concrete data model. These invariants, so-called *gluing invariants* are used in the formulation of the refinement proof obligations.
The goal of a B development is to obtain a *proved model*. Since the development process leads to a large number of proof obligations, the mastering of proof complexity is a crucial issue. Even if a proof tool is

available, its effective power is limited by classical results over logical theories and we must distribute the complexity of proofs over the components of the current development, e.g. by refinement. Refinement has the potential to decrease the complexity of the proof process whilst allowing for traceability of requirements. B Models rarely need to make assumptions about the *size* of a system being modelled, e.g. the number of nodes in a network. This is in contrast to model checking approaches [54]. The price to pay is to face possibly complex mathematical theories and difficult proofs. The re-use of developed models and the structuring mechanisms available in B help in decreasing the complexity. Where B has been exercised on known difficult problems, the result has often been a simpler proof development than has been achieved by users of other more monolithic techniques [82].

1.3 Scope of the B modelling

The scope of the B method concerns the complete process of software and system development. Initially, the B method was mainly restricted to the development of software systems [27, 32, 68] but a wider scope for the method has emerged with the incorporation of the event based approach [1, 13, 7, 8, 38, 36, 93] and is related to the systematic derivation of reactive distributed systems. Events are simply expressed in the rich syntax of the B language. Abrial and Mussat [13] introduce elements to handle liveness properties. The refinement of the event-based B method does not deal with fairness constraints but introduces explicit counters to ensure the happening of abstract events, while new events are introduced in a refined model. Among case studies developed in B, we can mention the METEOR project [27] for controlling train traffic, the PCI protocol [41], the IEEE 1394 Tree Identify Protocol [12]. Finally, B has been combined with CSP for handling communications systems [36, 35] and with action systems [38, 93].

The proposal can be compared to action systems [23], UNITY programs [52] and TLA [71] specifications but there is no notion of abstract fairness like in TLA or in UNITY.

1.4 Related techniques

The B method is a state-based method integrating set theory, predicate calculus and generalized substitution language. We briefly compare it to related notations.
Like Z [94], B is based on the ZF set theory; both notations share the same roots, but we can point to a number of interesting differences. Z expresses state change by use of before and after predicates, whereas the predicate transformer semantics of B allows a notation which is closer to programming. Invariants in Z are incorporated into operation descriptions and alter their meaning, whereas the invariant in B is checked against the state changes described by operations and events to ensure consistency. Finally B makes a careful distinction between the logical properties of pre-conditions and guards, which are not clearly distinguished in Z.
The refinement calculus used in B for defining the refinement between models in the event-based B approach is very close to Back's action systems, but tool support for action systems appears to be less mechanized than B.
TLA$^+$ [72] can be compared to B, since it includes set theory with the ϵ operator of Hilbert. The semantics of TLA temporal operators is expressed over traces of states whereas the semantics of B actions is expressed in the weakest precondition calculus. Both semantics are equivalent with respect to safety properties, but the trace semantics of TLA$^+$ allows an expression of fairness and eventuality properties that is not directly available in B.
VDM [69] is a method with similar objectives to classical B. Like B it uses partial functions to model data, which can lead to meaningless terms and predicates e.g. when a function is a applied outside its domain. VDM uses a special three valued logic to deal with undefinedness. B retains classical two valued logic, which simplifies proof at the expense of requiring more care with undefinedness. Recent approaches to this problem will be mentioned later.
ASM [65, 39] and B share common objectives related to the design and the analysis of (software/hardware) systems. Both methods bridge the gap between human understanding and formulation of real-world problems and the deployment of their computer-based solutions. Each has a simple scientific foundation: B is

based on set theory and ASM is based on the algebraic framework with an abstract state change mechanism. An Abstract State Machine is defined by a signature, an abstract state, a finite collection of rules and a specific rule; rules provide an operational style very useful for modelling specification and programming mechanisms. Like B, ASM includes a refinement relation for the incremental design of systems; the tool support of ASM is under development but it allows one to verify and to analyse ASMs. In applications, B seems to be more mature than ASM, even if ASM has several real successes like the validation [95] of Java and the Java Virtual Machine.

1.5 Summary

Next sections provide a short description of event B:

- the B language and elements on the classical B method: syntax and semantics of operations, events, assertions, predicates, machines, models.
- the B modelling language and a simple introductory example: event B, refinement, proof-based development.
- other sections illustrate the event B modelling method by case studies:
  - Sequential algorithms.
  - Combining coordination and refinement for sorting.
  - Spanning trees algorithms.
  - A distributed leader election algorithm.
- Final section concludes the chapter on the B modelling techniques and on ongoing researches.

Chapter 2

The B language

2.1 The B language for sets, predicates and logical structures

The development of a model starts by an analysis of the mathematical structure: sets, constants and properties over sets and constants and we produce the mathematical landscape by requirements elicitation. However, the statement of mathematical properties can be expressed using different assumed properties; for instance, a constant n is a natural number and is supposed to be greater than 3 - classically and formally written like $n \in \mathbb{N} \ \wedge \ n \geq 3$ - or a set of $persons$ is not empty - classically and formally written like $persons \neq \emptyset$. Abrial et al [9] develop a *structure language* which allows to one to encode mathematical structures and their accompanying theorems. Structures improve the possibility of mechanized proofs but they are not yet in the current version of the B tools; there is a close connection with the structuring mechanisms and the algebraic structures [62], but the main difference is in the use of sets rather than of abstract data types. B mathematical structures are built with notations of set theory and we list the main notations (and their meanings) used in further subsections; the complete notation is described in the B book of Abrial [21].

2.1.1 Sets and predicates

Constants can be defined using first order logic and set-theoretical notations of B. A set can be defined using either the comprehension schema $\{ \ x \mid x \ \in \ s \wedge P(x) \}$, or the Cartesian product schema $s \times t$ or using operators over sets like power $\mathbb{P}(s)$, intersection $\cap$ and union $\cup$. $y \ \in \ s$ is a predicate which can be sometimes simplified either from $y \ \in \ \{ \ x \mid x \ \in \ s \wedge P(x) \}$ into $y \ \in \ s \wedge P(y)$, or from $x \ \mapsto \ y \ \in \ s \times t$ into $x \ \in \ s \wedge y \ \in \ t$, or from $t \ \in \ \mathbb{P}(s)$ into $\forall \ x \ . \ (\ x \ \in \ t \ \Rightarrow \ x \ \in \ s)$ where x is a fresh variable. A pair is denoted either $(\ x \ , \ y \ )$ or $x \ \mapsto \ y$.
A relation over two sets s and t is an element of $\mathbb{P}(s \times t)$; a relation r has a domain $\mathsf{dom}(r)$ and a co-domain $\mathsf{ran}(r)$. A function f from the set s to the set t is a relation such that each element of $\mathsf{dom}(f)$ is related to at most one element of the set t.
A function f is either partial $f \ \in \ A \ \nrightarrow \ B$, or total $f \ \in A \rightarrow \ B$. Then, we can define the term $f(x)$ for every element x in $\mathsf{dom}(f)$ using the choice function ($f(x) = \mathsf{choice}(f[\{x\}])$ where $f[\{x\}]$ is the subset of t, whose elements are related to x by f. The choice function assumes that there exists at least one element in the set, which is not the case of the ϵ operator that can be applied to an empty set and returns some value. If $x \mapsto y \in f$ then $y = f(x)$ and $f(x)$ is well defined, only if f is a function and x is in $\mathsf{dom}(f)$.
We summarize in figure 2.1, set-theoretical notations that can be used in the writing of formal definitions related to constants. In fact, the modelling of data is oriented by sets, relations and functions; the task of the specifier is then to use effectively those notations.

2.1.2 A simple case study

Since we have a short space for explaining B concepts, we use a very simple case study, namely the development of models for computing the $factorial$ function; we can illustrate the expressivity of the B language of predicates. Other case studies can be found in complete work separately published (see for instance,[21, 1, 7, 5, 2, 11, 41, 12]). When considering the definition of a function, we can use different

Name	Syntax	Definition
Binary Relation	$s \leftrightarrow t$	$\mathcal{P}(s \times t)$
Composition of relations	$r_1 ; r_2$	$\{x, y \mid x \in a \ \wedge \ y \in b \ \wedge$ $\exists z.(z \in c \ \wedge \ x, z \in r_1 \ \wedge \ z, y \in r_2)\}$
Inverse relation	r^{-1}	$\{x, y \mid x \in \mathcal{P}(a) \ \wedge \ y \in \mathcal{P}(b) \ \wedge \ y, x \in r\}$
Domain	$\mathsf{dom}(r)$	$\{a \mid a \in s \ \wedge \ \exists b.(b \in t \ \wedge \ a \mapsto b \in r)\}$
Range	$\mathsf{ran}(r)$	$\mathsf{dom}(r^{-1})$
Identity	$\mathsf{id}(s)$	$\{x, y \mid x \in s \ \wedge \ y \in s \ \wedge \ x = y\}$
Restriction	$s \lhd r$	$\mathsf{id}(s) ; r$
Co-restriction	$r \rhd s$	$r ; \ \mathsf{id}(s)$
Anti-restriction	$s \lhd\!\!\!- r$	$(\mathsf{dom}(r) - s) \lhd r$
Anti-co-restriction	$r \rhd\!\!\!- s$	$r \rhd (\mathsf{ran}(r) - s)$
Image	$r[w]$	$\mathsf{ran}(w \lhd r)$
Overriding	$q \lhd\!\!+ r$	$(\mathsf{dom}(r) \lhd\!\!\!- q) \cup r$
Partial Function	$s \nrightarrow t$	$\{r \mid r \in s \leftrightarrow t \ \wedge \ (r^{-1} ; r) \subseteq \mathsf{id}(t)\}$

Figure 2.1: Set-theoretical notations

styles to characterize it. A function is mathematically defined as a (binary) relation over two sets, called source and target and it satisfies the *functionality property*. The set-theoretical framework of B invites us to follow this way for defining functions; however, a recursive definition of a given function is generally used. The recursive definition states that a given mathematical object exists and that it is the least solution of a fixed-point equation. Hence, a first step of the B development proves that the function defined by a relation is the least fixed-point of the given equation. Properties of the function might be assumed, but we prefer to advocate a style of *fully proved development* with respect to a minimal set of assumptions. The first step enumerates a list of basic properties considered as axioms and the final step reaches a point where both definitions are proved to be equivalent.
First, we define the mathematical function *factorial*, in a classical way; the first line states that *factorial* is a total function from $\mathbb{N}$ into $\mathbb{N}$ and the next lines state that *factorial* satisfies a fixed-point and, by default, it is supposed to be the least fixed-point. *factorial* is a B *constant* and has B *properties*:

$$\begin{array}{l} \mathit{factorial} \in \mathbb{N} \longrightarrow \mathbb{N} \wedge \\ \mathit{factorial}(0) = 1 \wedge \\ \forall n.(n \geq 0 \Rightarrow \mathit{factorial}(n{+}1) = (n{+}1) \times \mathit{factorial}(n)) \end{array}$$

In previous work on B [43], we use this definition and write it as a B property (a logical assumption or an axiom of the current theory) but nothing tells us that the definition is consistent and that it defines an *existing* function. A solution is to define the *factorial* function using a fixed-point schema such that the *factorial* function is the least fixed-point of the given equation over relations. The *factorial* function is the smallest relation satisfying some conditions and especially the functionality; the functionality is stated as a *logical consequence* of the B properties. The point is not new but we are able to introduce notions to students putting together fixed-point theory, set theory, theory of relations and functions and the process of validation by proof (mechanically done by the prover). The computation of the *factorial* function starts by a definition of the *factorial* function which is carefully and formally justified using the theorem prover. *factorial* is still a B constant but it is differently defined.
The $factorial$ function is a relation over natural numbers and it is defined by its graph over pairs of natural numbers:

(**axioms or B properties**)

$$\begin{array}{l} \mathit{factorial} \in \mathbb{N} \leftrightarrow \mathbb{N} \wedge \\ 0 \mapsto 1 \in \mathit{factorial} \wedge \\ \forall (n, \mathit{fn}) \cdot \left(\begin{array}{l} n \mapsto \mathit{fn} \in \mathit{factorial} \\ \Rightarrow \\ n{+}1 \mapsto (n{+}1) \times \mathit{fn} \in \mathit{factorial} \end{array} \right) \end{array}$$

The *factorial* function satisfies the fixed-point equation and is the least fixed-point:

(**axioms or B properties**)

$$\forall f \cdot \left(\begin{array}{l} f \in \mathbb{N} \leftrightarrow \mathbb{N} \wedge \\ 0 \mapsto 1 \in f \wedge \\ \forall (n, fn).(n \mapsto fn \in f \Rightarrow n{+}1 \mapsto (n{+}1){\times}fn \in f) \\ \Rightarrow \\ \mathit{factorial} \subseteq f \end{array} \right)$$

These last statements are B properties of the $factorial$ function and from these B properties, we should derive the functionality of the resulting least fixed-point: *factorial is a function* is a logical consequence of the new definition of *factorial*.

(**consequences or B assertions**)

$$\begin{array}{l} \mathit{factorial} \in \mathbb{N} \longrightarrow \mathbb{N} \wedge \\ \mathit{factorial}(0) = 1 \wedge \\ \forall n.(n \in \mathbb{N} \Rightarrow \mathit{factorial}(n{+}1) = (n{+}1){\times}\mathit{factorial}(n)) \end{array}$$

Now, *factorial* is proved to be a function and no assumption concerning the functionality is left unspecified or simply an assumption. Proofs are carried out using the first order predicate calculus together with set theory and arithmetic. When we have proved that *factorial* is a function, it means that every derived property is effectively obtained by a mechanical process of proof; the proof can be reused in another case study, if necessary. The proof is an application of the induction principle; every inductive property mentions a property over values of the underlying structure namely $\mathcal{P}(n)$; hence we should quantify over predicates and derive theorems in higher order logic [9]. Using a quantification over subsets of a set, we can get higher order theorems. For instance, $\mathcal{P}(n)$ is represented by the following set $\{n | n \in NATURAL \wedge \mathcal{P}(n)\}$ and the inductive property is stated as follows; the first expression is given in the B language and the second expression (equivalent to the first one) in classical mathematical notation (`succ` denotes the successor function defined over natural numbers):

B statement

$$\forall P \cdot \left(\begin{array}{l} P \subseteq \mathbb{N} \wedge \\ 0 \in P \wedge \\ \mathsf{succ}[P] \subseteq P \\ \Rightarrow \\ \mathbb{N} \subseteq P) \end{array} \right)$$

classical logical statement

$$\forall \mathcal{P} \cdot \left(\begin{array}{l} \mathcal{P}(n) \text{ a property on } \mathbb{N} \wedge \\ \mathcal{P}(0) \wedge \\ \forall n \geq 0 \cdot (\mathcal{P}(n) \Rightarrow \mathcal{P}(n{+}1)) \\ \Rightarrow \\ \forall n \geq 0 \cdot \mathcal{P}(n) \end{array} \right)$$

The higher-order aspect is achieved by the use of set theory, which offers the possibility to *quantify over all the subsets of a set*. Such quantification give indeed the possibility to climb up to *higher-order* in a way that is always framed.

The structure language introduced by Abrial et al [9] can be useful to provide the reuse of already formally validated properties. It is then clear that the first step of our modelling process is an analysis of the mathematical landscape. The analysis of properties is essential, when dealing with the undefinedness of expressions and the work of Abrial et al [9] or the doctoral thesis of Burdy [34] propose different ways to deal with this question. For instance, the existence of a function like *factorial* may appear obvious but the technique of modelling might lead to silly models, if no proof of definedness is done. The proof of the functionality of *factorial* necessitates to instantiate the variable P in the inductive property by the following set:

$$\{n | n \in \mathbb{N} \wedge 0..n \lhd \mathit{factorial} \in 0..n \longrightarrow \mathbb{N}\}$$

Now, we consider the structures in B used for organizing axioms, definitions, theorems and theories.

2.1.3 Logical structures in B

The B language of predicates denoted $\mathcal{BP}$ for expressing data and properties combine set theory and first order predicate calculus with a simple arithmetic theory. The B environment can be used to derive theorems from axioms; B provides a simple way to express axioms and theorems using abstract machines without variables. It is a way to use the underlying B prover and to implement the proof process that we have already described.

machine
m
sets
s
constants
c
properties
$P(s, c)$
assertions
$A(x)$
end

An abstract machine has a name m; the clause **sets** contains definitions of sets in the problem; the clause **constants** allows one to introduce information related to the mathematical structure of the problem to solve and the clause **properties** contains the effective definitions of constants: it is very important to list carefully properties of constants in a way that can be easily used by the tool. The clause **assertions** contains the list of theorems to be discharged by the proof engine. The proof process is based on the sequent calculus and the prover provides (semi-)decision procedures [55] for proving the validity of a given logical fact called a sequent and allows one to build interactively the proof by applying possible rules of sequent calculus.

For instance, the machine *FACTORIAL_DEF* introduces a new constant called *factorial* satisfying given properties in the previous lines. The functionality of *factorial* is derived from the assumptions in the clause **assertions**.

machine
FACTORIAL_DEF
constants
$factorial$
properties
$$
\begin{array}{l}
factorial \in \mathbb{N} \leftrightarrow \mathbb{N} \wedge \\
0 \mapsto 1 \in factorial \wedge \\
\forall (n, fn).(n \mapsto fn \in factorial \Rightarrow n{+}1 \mapsto (n{+}1) times fn \in factorial) \wedge \\
\forall f \cdot \left(\begin{array}{l} f \in \mathbb{N} \leftrightarrow \mathbb{N} \wedge \\ 0 \mapsto 1 \in f \wedge \\ \forall (n, fn).(n \mapsto fn \in f \Rightarrow n{+}1 \mapsto (n{+}1) \times fn \in f) \\ \Rightarrow \\ factorial \subseteq f \end{array} \right)
\end{array}
$$
assertions
$factorial \in \mathbb{N} \longrightarrow \mathbb{N}$;
$factorial(0) = 1$;
$\forall n.(n \in \mathbb{N} \Rightarrow factorial(n{+}1) = (n{+}1) \times factorial(n))$
end

The interactive prover breaks a sequent into simpler-to-prove sequents but the user must know the global structure of the final proof. $\mathcal{BP}$ allows us to define underlying mathematical structures required for a given problem; now we should introduce how to specify states and how to describe *transitions* over *states*.

2.2 The B language of transitions

The B language is not restricted to classical set-theoretical notations and the sequent calculus; it includes notations for defining *transitions* over *states* of the model, called *generalized substitutions*. In its simple form, $x := E(x)$, a generalized substitution looks like an assignment; the B language of generalized substitutions called GSL (Generalized Substitution Language) (see figure 2.2) contains syntactical structures for expressing different kinds of (states) transitions. Generalized substitutions of GSL allow us to write

operations in the classical B approach [21]; a restriction over GSL leads to *events* in the so called event-based B approach [13, 8]. In the following sub-subsections, we address the semantical issues of generalized substitutions and the differences between *operations* and *events*.

2.2.1 Generalized Substitutions

Generalized substitutions provide a way to express transformations of state variables of a given model. In the construct $x := E(x)$, x denotes a vector of state variables of the model, and $E(x)$ a vector of expressions of the same size as the vector x. The interpretation we shall give here to this statement is *not* however that of an assignment statement. The class of generalized substitutions contains the following possible forms of generalized substitutions:

- $x := E$ (assignment).
- **skip** (stuttering).
- $P \mid S$ (precondition) (or **PRE** P **THEN** S **END**).
- $S \parallel T$ (bounded choice) (or **CHOICE** S_1 **OR** S_2 **END**).
- $P \Rightarrow S$ (guard) (or **SELECT**(or**WHEN**) P **THEN** S **END**).
- $@z.S$ (unbounded choice).
- $x :\in S$ (set choice), $x : R(x_0, x)$ (generalized assignment).
- $S_1; S_2$ (sequencing).
- **WHILE** B **DO** S **INVARIANT** J **VARIANT** V **END**.

The meaning of a generalized substitution S is defined in the weakest-precondition calculus [60, 61] by the predicate transformer $\lambda P \in \mathcal{BP}.[S]P$ where $[S]P$ means that S *establishes* P. Intuitively, it means that every *accepted* execution of S starting from a state s satisfying $[S]P$ terminates in a state satisfying P; certain substitutions can be *feasibly* executed (or accepted for execution) by any physical computational device; it means also that S terminates for every state of $[S]P$. The weakest-precondition operator has properties related to implication over predicates: $\lambda P \in \mathcal{BP}.[S]P$ is monotonic with respect to the implication, it is distributive with respect to the conjunction of predicates. The properties of the weakest-precondition operator are known, since the work of Dijkstra [60, 61] on the semantics defined by predicate transformers. The definition of $\lambda P \in \mathcal{BP}.[S]P$ is inductively expressed over the syntax of B predicates and the syntax of generalized substitutions. $[S]P$ can be reduced to a B predicate, which is used by the proof-obligations generator. Figure 2.2 contains the inductive definition of $[S]P$.
We say that two substitutions S_1 and S_2 are equivalent, denoted $S_1 = S_2$, if for any predicate P of the B language, $[S_1]P \equiv [S_2]P$. The relation defines a way to compare substitutions. Abrial [21] proves a theorem for normalized form related to any substitution and it proves that a substitution is characterized by a precondition and a computation relation over variables.

Théorème 1 *[21]*
For any substitution S, there exists two predicates P and Q where x' is not free in P such that: $S = P|@x'.(Q \Longrightarrow x := x')$.

The theorem tells us the importance of the precondition of a substitution, which should be true, when the substitution is applied to the current state, else the resulting state is not consistent with the transformation. Q is a relation between the initial state x and the next state x'. In fact, a substitution should be applied to a state satisfying the invariant and should preserve it. Intuitively, it means that, when one applies the substitution, one has to check that the initial state is correct. The weakest-precondition operator allows to define specific conditions over substitutions:

- Aborted computations: $\mathsf{abt}(S) \stackrel{def}{=}$ *for any predicate R,* $\neg[S]R$ and *it defines the set of states that can not establish any predicate R and that are the non-terminating states.*

Name	Generalized substitution : S	$[S]P$
Assignment	$x := E$	$P(E/x)$
Skip	$skip$	P
Parallel Composition	$x := E \|\| y := F$	$[x, y := E, F]P$
Non-deterministic Choice in a Set	$x :\in S$	$\forall v.(v \in S \Rightarrow P(v/x))$
Relational Assignment	$x : R(x_0, x)$	$\forall v.(R(x_0, v) \Rightarrow P(v/x))$
Unbounded Choice	$@x.S$	$\forall x.[S]P$
Bounded Choice	**choice** S_1 **or** S_2 **end** (*or equivalently* $S_1 [] S_2$)	$[S_1]P \ \wedge \ [S_2]P$
Guard	**select** G **then** T **end** (*or equivalently* $G \Longrightarrow S_2$)	$G \ \Rightarrow \ [T]P$
Precondition	**pre** G **then** T **end** (*or equivalently* $G\|T$)	$G \ \wedge \ [T]P$
Generalized Guard	**any** t **where** G **then** T **end**	$\forall t \cdot (G \ \Rightarrow \ [T]P)$
Sequential Composition	$S; T$	$[S][T]P$

Figure 2.2: Definition of GSL and $[S]P$

Generalized substitution : S	trm(S)
$x := E$	$TRUE$
$skip$	$TRUE$
$x :\in S$	$TRUE$
$x : R(x_0, x)$	$TRUE$
$@x.S$	$\forall x.\mathsf{trm}(S)$
choice S_1 **or** S_2 **end** (*or equivalently* $S_1 [\!] S_2$)	$\mathsf{trm}(S_1) \ \wedge \ \mathsf{trm}(S_2)$
select G **then** T **end** (*or equivalently* $G \Longrightarrow S_2$)	$G \ \Rightarrow \ \mathsf{trm}(T)$
pre G **then** T **end** (*or equivalently* $G\|T$)	$G \ \wedge \ \mathsf{trm}(T)$
any t **where** G **then** T **end**	$\forall\, t \cdot (\, G \ \Rightarrow \ \mathsf{trm}(T)\,)$

Figure 2.3: Examples of definitions for $\mathsf{trm}(S)$

- Terminating computations: $\mathsf{trm}(S) \stackrel{def}{=} \neg\mathsf{abt}(S)$ and *it defines the termination condition for the substitution* S.
- Miraculous computations: $\mathsf{mir}(S) \stackrel{def}{=}$ *for any predicate R,* $[S]R$ and means that *among states, some states may establish every predicate R, for instance* $FALSE$*, and they are called miraculous states, since they establish a miracle.*
- Feasible computations: $\mathsf{fis}(S) \stackrel{def}{=} \neg\mathsf{mir}(S)$ *Miraculous states correspond to non-feasible computations and the feasibility condition ensures that the computation is realistic.*

Terminating computations and feasible computations play a central role in the analysis of generalized substitutions, whose the expressivity if very important. The figures 2.3 and 2.4 provide two lists of rules for simplifying $\mathsf{trm}(S)$ and $\mathsf{fis}(S)$ into the B predicates language; both lists are not complete (see Abrial [21] for complete lists).
For instance, fis(**select** FALSE **then** $x := 0$ **end**) is $FALSE$ and mir(**select** FALSE **then** $x := 0$ **end**) is $TRUE$; the substitution **select** FALSE **then** $x := 0$ **end** establishes any predicate and is not feasible. We can not implement such a substitution in a programming language.

A relational predicate can be defined using the weakest-precondition semantics, namely $\mathsf{prd}_x(S)$, by the expression $\neg[S](x \neq x')$ which is the relation characterizing the computations of S. The figure 2.5 contains a list of definitions of the predicate with respect to the syntax.

Generalized substitution : S	fis(S)
$x := E$	$TRUE$
$skip$	$TRUE$
$x :\in S$	$S \neq \emptyset$
$x : R(x_0, x)$	$\exists v.(R(x_0, v)$
$@x.S$	$\exists x.\mathsf{fis}(S)$
choice S_1 **or** S_2 **end** (*or equivalently* $S_1 [\!] S_2$)	$\mathsf{fis}(S_1) \ \vee \ \mathsf{fis}(S_2)$
select G **then** T **end** (*or equivalently* $G \Longrightarrow S_2$)	$G \ \wedge \ \mathsf{fis}(T)$
pre G **then** T **end** (*or equivalently* $G \mid T$)	$G \ \Rightarrow \ \mathsf{fis}(T)$
any t **where** G **then** T **end**	$\exists\, t \cdot (\, G \ \wedge \ \mathsf{fis}(T)\,)$

Figure 2.4: Examples of definitions for $\mathsf{fis}(S)$

Generalized substitution : S	$\mathsf{prd}_x(S)$
$x := E$	$x' = E$
$skip$	$x' = x$
$x :\in S$	$x' \in S$
$x : R(x_0, x)$	$R(x, x')$
$@z.S$	$\exists z.\mathsf{prd}_x(S)$ *if* $z \neq x'$
choice S_1 **or** S_2 **end** (*or equivalently* $S_1 [\!] S_2$)	$\mathsf{prd}_x(S_1) \ \vee \ \mathsf{prd}_x(S_2)$
select G **then** T **end** (*or equivalently* $G \Longrightarrow S_2$)	$G \ \wedge \ \mathsf{prd}_x(T)$
pre G **then** T **end** (*or equivalently* $G \mid T$)	$G \ \Rightarrow \ \mathsf{prd}_x(T)$
any t **where** G **then** T **end**	$\exists t \cdot (G \ \wedge \ \mathsf{prd}_x(T))$

Figure 2.5: Examples of definitions for $\mathsf{prd}_x(S)$

The next property is proved by Abrial and shows the relationship between weakest-precondition and relational semantics. Predicates $\mathsf{trm}(S)$ and $\mathsf{prd}_x(S)$ are respectively defined in figure 2.3 and figure 2.5.

Théorème 2 *[21]*
For any substitution S, we have: $S = \mathit{trm}(S)|@x'.(\mathit{prd}_x(S) \Longrightarrow x := x')$

Both theorems emphasize the role of the precondition and the relation in the semantical definition of a substitution. The refinement of two substitutions is simply defined using the weakest-precondition calculus as follows: S is refined by T (written $S \sqsubseteq T$), if for any predicate P, $[S]P \Rightarrow [T]P$. We can give an equivalent version of the refinement that shows that it decreases the non-determinism. Let us define the following sets: $\mathsf{pre}(S) = \{x|x \in s \wedge \mathsf{trm}(S)\}$, $\mathsf{rel}(S) = \{x, x'|x \in s \wedge x' \in s \wedge \mathsf{prd}_x(S)\}$ and $\mathsf{dom}(S) = \{x|x \in s \wedge \mathsf{fis}(S)\}$ where s is supposed to be the global set of states. The refinement can be defined equivalently using the set-theoretical versions: S is refined by T, if, and only if, $\mathsf{pre}(S) \subseteq \mathsf{pre}(T)$ and $\mathsf{rel}(T) \subseteq \mathsf{rel}(S)$. We can also use previous notations and define equivalently the refinement of two substitutions by the expression: $\mathsf{trm}(S) \Rightarrow \mathsf{trm}(T)$ and $\mathsf{prd}_x(T) \Rightarrow \mathsf{prd}_x(S)$. The predicate $\mathsf{prd}_x(S)$ relates S to a relation over x and x'; it means that a substitution can be seen like a relation over pairs of states. The weakest-precondition semantics over generalized substitutions provides the semantical foundation of the generator of proof obligations; in the next sub-subsections we introduce operations and events, which are two ways to use the B method.

2.2.2 Operations and events

Generalized substitutions are used to construct *operations* of *abstract machines* or *events* of *abstract models*. Both notions will be detailed in the next subsection. However, we should explain the difference between those two notions. A (abstract) machine is a structure with a part defining data (**sets, constants, properties**), a part defining state (**variables,invariant**) and a part defining operations (**operations, initialisation**); it only gives its potential user the ability to activate the operations, not to access its state directly and this aspect is very important for refining the machine by making changes of variables and of operations, while keeping their names. An operation has a precondition and the precondition should be true, when one calls the operation. Operations are characterized by generalized substitutions and their semantics is based on the semantics of generalized substitutions (either in the weakest-precondition-based style, or in the relational style). It means that the condition of preservation of the invariant is simply written as follows:

$$I \wedge \mathsf{trm}(O) \Rightarrow [O]I \tag{2.1}$$

If one calls the operation, when the precondition is false, any state can be reached and the invariant is not ensured. The style of programming is called *generous* but it assumes that an operation is always called when the precondition is true. An operation can have input and output parameters and it is called in a state satisfying the invariant and it is a passive object, since it requires to be called to have an effect.
On the other hand, an event has a guard and is triggered in a state validating the guard. Both operation and event have a name, but an event has no input and output parameters. An event is observed or not observed. and possible changes of variables should maintain the invariant of the current model: the style is called *defensive*. Like an operation, an event is characterized by a generalized substitution and it can be defined by a relation over variables and primed variables: a before-after predicate denoted $BA(e)(x, x')$. An event is essentially a reactive object and reacts with respect to its guard $\mathsf{grd}(e)(x)$. However, there is a restriction over the language GSL used for defining events and we authorize only three kinds of generalized substitutions (see the figure 2.6). In the definition of an event, three basic substitutions are used to write an event ($x := E(x),\ x :\in S(x),\ x : P(x_0, x)$) and the last substitution is the normal form of the three ones. An event should be *feasible* and the feasibility is related to the feasibility of the generalized substitution of the event: some next state must be reachable from a given state. Since events are reactive objects, related proof obligations should guarantee that the current state satisfying the invariant should be feasible. The figure 2.7 contains the definition of guards of events. We leave the classical abstract machines of the B classical approach and we illustrate the system modelling through events and models.
When using the relational style for defining the semantics of events, we use the style advocated by Lamport [71] in TLA; an event is seen as a transformation between states before the transformation and states after the transformation. Lamport uses the priming of variables to separate before values from after values. Using this notation and supposing that x_0 denotes the value of x before the transition of the event,

Event : E	Before-After Predicate
begin $x : P(x_0, x)$ **end**	$P(x, x')$
when $G(x)$ **then** $x : P(x_0, x)$ **end**	$G(x) \ \wedge \ P(x, x')$
any t **where** $G(t, x)$ **then** $x : P(x_0, x, t)$ **end**	$\exists t \cdot (G(t, x) \ \wedge \ P(x, x', t))$

Figure 2.6: Definition of events and before-after predicates of events

Event : E	Guard: grd(E)
begin S **end**	$TRUE$
when $G(x)$ **then** T **end**	$G(x)$
any t **where** $G(t, x)$ **then** T **end**	$\exists t \cdot G(t, x)$

Figure 2.7: Definition of events and guards of events

events can get a semantics defined over primed and unprimed variables in figure 2.6. The before-after predicate is already defined in the B book as the predicate $\mathsf{prd}_x(S)$ defined for every substitution S (see sub-subsection 2.2.1).
Any event e 2.7 has a guard defining the enabledness condition over the current state and it expresses the existence of a next state. For instance, the disjunction of all guards is used for strengthening the invariant of a B system of events to include the deadlock freedom of the current model. The new syntax of event B accepts both equivalent expressions: $x : E(x_0, x)$ or $e : |E(x_0, x)$. Before to introduce B models, we give the expression stating the preservation of a property by a given event e:

$$I(x) \ \Rightarrow \ [e] \, I(x) \tag{2.2}$$

or equivalently in a relational style

$$I(x) \ \wedge \ BA(e)(x, x') \ \Rightarrow \ I(x') \tag{2.3}$$

$BA(e)(x, x')$ is the before-after relation of the event e and $I(x)$ is a state predicate over variables x. The equation 2.1 states the proof obligation of the operation O using the weakest-precondition operator and the equation 2.3 defines the proof obligation for the preservation of $I(x)$, while e is observed. Since the two approaches are semantically equivalent, the proof-obligations generator of the Atelier B can be reused for generating those assertions in the B environment. The **SELECT** event is the previous notation for the WHEN event; both are equivalent; however, the **WHEN** notation captures the idea of reactivity of guarded events; B$^\sharp$ [14] will provide other notations for combining events. In the next subsection, we detail abstract machines and abstract models, which are using operations and events.

Chapter 3

B models

3.1 Modelling Systems

Systems under consideration are software systems, control systems, protocols, sequential and distributed algorithms, operating systems, circuits; they are generally very complex and have parts interacting with an environment. A discrete abstraction of such systems constitutes an adequate framework: such an abstraction is called a *discrete model*. A discrete model is more generally known as a *discrete transition system* and provides a view of the current system; the development of a model in B follows an incremental process validated by the refinement. A system is modelled by a sequence of models related by the refinement and managed in a project.

A project [21, 8] in B contains information for editing, proving, analysing, mapping and exporting models or components. A B component has two separate forms: a first form concerns the development of software models and B components are *abstract machine, refinement, implementation*; a second form is related to modelling reactive systems using the event-based B approach and B components are simply called *models*. Each form corresponds to a specific approach for developing B components; the first form is fully supported by the B tools [55, 77] and the second one is partly supported by tools [55]. In the next sub-subsections, we overview each approach based on the same logical and mathematical concepts.

3.1.1 Modelling Systems in the B classical approach

The B method [21] is historically applied to software systems and has helped in developing safe software controlling trains [27]. The scope of the method is not restricted to the specification step but includes facilities for designing larger models or machines gathered in a project. The basic model is called an *abstract machine* and is defined in the A(bstract) M(achine) N(otation) language. We describe an abstract machine in the next figure. An abstract machine encapsulates variables defining the state of the system; the state should conform to the invariant and each operation should be called, when the current state satisfies the invariant. Each operation should preserve the invariant, when it is called.

An operation may have input/output parameters and only operations can change state variables. An abstract machine looks like a desk calculator and each time a user presses the button of an operation, he should check that the precondition of the operation is true, else no preservation of invariant can be ensured (for instance, division by zero). Structuring mechanisms will be reviewed in the sub-subsection 3.1.3. An abstract machine has a name m; the clause **sets** contains definitions of sets; the clause **constants** allows one to introduce information related to the mathematical structure of the problem to solve and the clause **properties** contains the effective definitions of constants: it is very important to list carefully properties of constants in a way that can be easily used by the tool. We do not mention structuring mechanisms like *sees, includes, extends, promotes, uses, imports* but they can help in the management of proof obligations.

machine
FACTORIAL_MAC
constants
$factorial, m$
constants
$factorial$
properties
$m \in \mathbb{N} \wedge$
$factorial \in \mathbb{N} \leftrightarrow \mathbb{N} \wedge$
$$\forall f \cdot \left(\begin{array}{l} f \in \mathbb{N} \leftrightarrow \mathbb{N} \wedge \\ 0 \mapsto 1 \in f \wedge \\ \forall (n, fn).(n \mapsto fn \in f \Rightarrow n{+}1 \mapsto (n{+}1) \times fn \in f) \\ \Rightarrow \\ factorial \subseteq f \end{array} \right)$$
variables
$result$
invariant
$result \in \mathbb{N}$
assertions
$factorial \in \mathbb{N} \longrightarrow \mathbb{N}$;
$factorial(0) = 1$;
$\forall n.(n \in \mathbb{N} \Rightarrow factorial(n{+}1) = (n{+}1) \times factorial(n))$
initialisation
$result :\in \mathbb{N}$
operations
$computation =$ **begin** $result := factorial(m)$ **end**
end

Figure 3.1: An example of an abstract machine for the factorial computation

machine
m
sets
s
constants
c
properties
$P(s, c)$
variables
x
invariant
$I(x)$
assertions
$A(x)$
initialisation
<substitution>
operations
<list of operations>
end

The second part of the abstract machine defines dynamic aspects of state variables and properties over variables using the *invariant* generally called *inductive invariant* and using *assertions* generally called *safety properties*. The invariant $I(x)$ types the variable x, which is assumed to be initialized with respect to the initial conditions and which is supposed to be preserved by operations (or transitions) of the list of operations. Conditions of verification called *proof obligations* are generated from the text of the model using the first part for defining the mathematical theory and the second part is used to generate proof obligations for the preservation (when calling the operation) of the invariant and proof obligations stating the correctness of safety properties with respect to the invariant. The figure 3.1 contains an example of an abstract machine with only one operation setting the variable $result$ to the value of the $factorial(m)$, with m a constant.

3.1.2 Modelling systems in the event-based B approach

Abstract machines are based on classical mechanisms like the call of operation or the input/output mechanisms. On the other hand, reactive systems reacts to the environment with respect to external stimuli; abstract models of the event-based B approach intend to integrate the reactivity to stimuli by promoting events rather than operations. Contrary to operations, events have no parameters and there is no access to state variables. At most one event is observed at any time of the system.

A (abstract) model is made up of a part defining mathematical structures related to the problem to solve and a part containing elements on state variables, transitions and (safety and invariance) properties of the model. Proof obligations are generated from the model to ensure that properties are effectively holding: it is called *internal consistency* of the model. A model is assumed to be closed and it means that every possible change over state variables is defined by transitions; transitions correspond to events observed by the specifier. A model m is defined by the following structure. A model has a name m; the clause **sets** contains definitions of sets of the problem; the clause **constants** allows one to introduce information related to the mathematical structure of the problem to solve and the clause **properties** contains the effective definitions of constants: it is very important to list carefully properties of constants in a way that can be easily used by the tool. Another point is the fact that sets and constants can be considered like parameters and extensions of the B method exploit this aspect to introduce parametrization techniques in the development process of B models. The second part of the model defines dynamic aspects of state variables and properties over variables using the invariant called generally inductive invariant and using assertions called generally safety properties. The invariant $I(x)$ types the variable x, which is assumed to be initialized with respect to the initial conditions and which is preserved by events (or transitions) of the list of events.

Conditions of verification called proof obligations are generated from the text of the model using the first part for defining the mathematical theory and the second part is used to generate proof obligations for the preservation of the invariant and proof obligations stating the correctness of safety properties with respect to the invariant. The predicate $A(x)$ states properties derivable from the model invariant. A model states that state variables are always in a given set of possible values defined by the invariant and it contains the only possible transitions operating over state variables.

A model is not a program and no control flow is related to it; however, it requires a validation but we first define the mathematics for stating sets, properties over sets, invariants, safety properties. Conditions of consistency of the model are called *proof obligations* and they express the preservation of invariant properties and avoidance of deadlock.

model
m
sets
s
constants
c
properties
$P(s, c)$
variables
x
invariant
$I(x)$
assertions
$A(x)$
initialisation
<substitution>
events
<list of events>
end

	Proof obligation
(INV1)	$Init(x) \Rightarrow I(x)$
(INV2)	$I(x) \wedge BA(e)(x, x') \Rightarrow I(x')$
(DEAD)	$I(x) \Rightarrow (\mathsf{grd}(e_1) \vee \ldots \mathsf{grd}(e_n))$

$e_1, \ldots, e_n$ is the list of events of the model m. (INV1) states the initial condition which should establish the invariant. (INV2) should be checked for every event e of the model, where $BA(e)(x, x')$ is the before-after predicate of e. (DEAD) is the condition of deadlock-freedom: at least one event is enabled.

Finally, predicates in the clause **assertions** should be implied by the predicates of the clause **invariant**; the condition is simply formalized as follows:

$$P(s,c) \ \wedge \ I(x) \ \Rightarrow \ A(x)$$

Finally, the substitution of an event must be feasible; an event is feasible with respect to its guard and the invariant $I(x)$, if there is always a possible transition of this event or equivalently, there exists a next value x' satisfying the before-after predicate of the event. The feasibility of the initialisation event requires that at least one value exists for the predicate defining the initial conditions. The feasibility of an event leads to a readability of the form of the event; the recognition of the guard in the text of the event simplifies the semantical reading of the event and it simplifies the translation process of the tool: no guard is hidden inside the event. We summarize the feasibility conditions in the next table.

Event : E	Feasibility : $fis(E)$
$x : Init(x)$	$\exists x \cdot Init(x)$
begin $x : P(x_0, x)$ **end**	$I(x) \ \Rightarrow \ \exists x' \cdot P(x, x')$
when $G(x)$ **then** $x : P(x_0, x)$ **end**	$I(x) \ \wedge \ G(x) \ \Rightarrow \ \exists x' \cdot P(x, x')$
any l **where** $G(l, x)$ **then** $x : P(x_0, x, l)$ **end**	$I(x) \ \wedge \ G(l, x) \ \Rightarrow \ \exists x' \cdot P(x, x', l)$

Proof obligations for a model are generated by the proof-obligations generator of the B environment; the sequent calculus is used to state the validity of the proof obligations in the current mathematical environment defined by constants, properties. Several proof techniques are available but the proof tool is not able to prove automatically every proof obligation and interactions with the prover should lead to prove every generated proof obligation. We say that the model is *internally consistent* when every proof obligation is proved. A model uses only three kinds of events, while the generalized substitutions are richer; but the objectives are to provide a simple and powerful framework for modelling reactive systems. Since the consistency of a model is defined, we should introduce the refinement of models using the refinement of events defined like the substitution refinement. We reconsider the example of the $factorial$ function and its computation and we propose the model of the figure 3.2. As you notice, the abstract machine fac and the abstract model fac are very close and the main difference is in the use of events rather than operations: the event $computation$ eventually appears or is executed, because of the properties of the mathematical function called $factorial$. The operation $computation$ of the machine in the figure 3.1 is passive, but the event $computation$ of the model in the figure 3.2 is reactive, when it is possible. Moreover, events may hide other ones and the refinement of models will play a central role in the development process. We present in the next sub-subsection classical mechanisms for structuring developed components of specification.

3.1.3 Structuring mechanisms of the B method

In the last two sub-subsections, we have introduced B models following the classification into two main categories *abstract machines* and *models*; both are called *components* but they are not dealing with the same approach. We detail structuring mechanisms of both approaches to be complete on references of work on B.

Sharing B components

The AMN notation provides clauses related to structuring mechanisms in components like *abstract machines* but also like *refinements* or *implementations*. The B development process starts from basic compo-

model
FACTORIAL_EVENTS
constants
$factorial, m$
constants
$factorial$
properties
$m \in \mathbb{N} \wedge$
$factorial \in \mathbb{N} \leftrightarrow \mathbb{N} \wedge$
$0 \mapsto 1 \in factorial \wedge$
$\forall (n, fn).(n \mapsto fn \in factorial \Rightarrow n{+}1 \mapsto (n{+}1) \times fn \in factorial) \wedge$
$\forall f \cdot \left(\begin{array}{l} f \in \mathbb{N} \leftrightarrow \mathbb{N} \wedge \\ 0 \mapsto 1 \in f \wedge \\ \forall (n,fn).(n \mapsto fn \in f \Rightarrow n{+}1 \mapsto (n{+}1) \times fn \in f) \\ \Rightarrow \\ factorial \subseteq f \end{array} \right)$
variables
$result$
invariant
$result \in \mathbb{N}$
assertions
$factorial \in \mathbb{N} \longrightarrow \mathbb{N}$;
$factorial(0) = 1$;
$\forall n.(n \in \mathbb{N} \Rightarrow factorial(n{+}1) = (n{+}1) \times factorial(n))$
initialisation
$result :\in \mathbb{N}$
events
$computation =$ **begin** $result := factorial(m)$ **end**
end

Figure 3.2: An example of an abstract model for the *factorial* computation

nents mainly *abstract machines* and is layered development; the goal is to obtain implementation components through structuring mechanisms like *includes*, *sees*, *uses*, *extends*, *promotes*, *imports*, *refines*. The clauses *includes*, *sees*, *uses*, *extends*, *promotes*, *imports*, *refines* allow one to compose B components in the classical B approach and every clause leads to specific conditions for use. Several authors [87, 40] analyse the limits of existing B primitives to share data, while refining and composing B components; it is clear that the B primitives for structuring B components can be used following strong conditions on the sharing of data and operations. The limits are mainly due to the reuse of already proved B components; reuse of variables, invariants, constants, properties, operations. In fact, the problem to solve is the management of *interferences* among components and the seminal solution of Owicki and Gries [85] faces the combinatorial explosion of the number of proof obligations. The problem is to compose components according to given constraints of correctness. The new event-based B approach considers a different way to cope with structuring mechanisms and considers only two primitives: the *refines* primitive and the *decomposition* primitive.

B classical primitives for combining components

We focus on the meaning and the use of five primitives for sharing data and operations among B components, namely *includes*,*sees*, *uses*, *extends*, *promotes*. Each primitive is related to a clause of the AMN notation and allows access to data or operations of already developed components; specific proof obligations state conditions to ensure a sound composition. A structuring primitive makes accessed components visible under various degrees from the accessing component.
The *includes* primitive can be used in an abstract machine or in a refinement; the included component allows the including component to modify included variables by included operations; the included invariant is preserved by the including component and is really used by the tool for deriving proofs of proof obligations of the including component. The including component can not modify included variables but it can use them in read access. No interference is possible under those constraints. The *uses* primitives can only appear in abstract machines and using machines have a read-only access to the used machine, which can be shared by other machines. Using machines can refer to shared variables in their invariants and data of the used machine are shared among using machines. When a machine uses another machine, the current project must contain another machine including the using and the used machines. The refinement is related to the including machine and the using machine can not be refined. The *sees* primitive refers to an abstract machine imported in another branch of the tree structure of the project and sets, constants and variables can be consulted without change. Several machines can see the same machine. Finally, the *extends* primitive can only be applied to abstract machines and only one machine can extend a given machine; the *extends* primitive is equivalent to the *includes* primitive followed by the *promotes* primitive for every operation of the included machine. For instance, we can illustrate the implementation and we can show that the implementation of the figure 3.3 implements (refines) the machine of the figure 3.1. The operation $computation$ is refined or implemented by a while statement; proof obligations should take into account the termination of the operation in the implementation: the variant establishes the termination. Specific proof obligations are produced to check the absence of overflow of variables.

Organizing components in a project

The B development process is based on a structure defined by a collection of components which are either abstract machines, refinements or implementations. An implementation corresponds to a stage of development leading to the production of codes when the language of substitutions is restricted to the B0 language. The B0 language is a subset of the language of substitutions and translation to C, C++ or ADA is possible in tools. The links between components are defined by the B primitives previously mentioned and by the refinement.
When building a software system, the development starts from a document which may be written in a semi-formal specification language; the system is decomposed into subsystems and a model is progressively built using B primitives for composing B components. We emphasize the role of structuring primitives, since they allow to distribute the global proof complexity. The B development process covers the classical life cycle: requirements analysis, specification development, (formal) design and validation through the proof process and animation. K. Lano [73] illustrates an object-oriented approach of the B development and it identifies the layered development paradigm that we have already mentioned through B primitives.

implementation
FACTORIAL_IMP
refines
FACTORIAL_MAC
values
$m = 5$
concrete_variables
$result, x$
invariant
$x \in 0..n \wedge$
$result = factorial(x)$
assertions
factorial$(5) = 120 \wedge$
result ≤ 120
initialisation
$result := 1; x := 0$
operations
$computation =$
while $x < m$ **do**
$x := x{+}1; fn := x{\times}fn$
invariant
$x \in 0..m$
$result = factorial(x)$
$result \leq factorial(m)$
variant
$m{-}x$
end
end

Figure 3.3: An example of an implementation for the factorial computation

Finally, implementations are B components that are close to real code; in an implementation component, an operation can be refined by a while loop and the checking should prove that the while loop is terminating.

Structures for the event-based B approach

While the B classical approach is based on the B components and B structuring primitives, the event-based B approach promotes two concepts: the refinement of models and the decomposition of models [7, 8]. As we have already mentioned, the classical B primitives have limits in the scope of their use; we need mainly to manage sharing data but without generating too many proof obligations. So the main idea of Abrial is not to compose, but to decompose a first model and to refine models obtained after decomposition step. The new proposed approach simplifies the B method and focuses on the refinement. It means that previous development in the B classical approach can be replayed in the event-based B one. Moreover, the foundations of B remain useful and usable in the current environment of the Atelier B. In the next subsection, we describe the mathematical foundations of B and we illustrate B concepts in the event-based B approach.

Summary on structuring mechanisms

We have reviewed structuring mechanisms of the classical B approach and the new ones proposed for the event-based B approach. While the classical approach provides several mechanisms for structuring machines, only two mechanisms supports the event-based approach. In fact, the crucial point is to compose abstract models or abstract machines; the limit of composition is related upon the production of a too high number of proof obligations. The specifier wants to share state variables in read and write mode; the structuring mechanisms of classical B do not allow the sharing of variable, but in read mode. Our work on the feature interaction problem [42] illustrates the use of refinement for composing features and other approaches based on the detection of interaction by using a model checker on finite models, do not cope the global problem because of finite models. Finally, we think that the choice of events with the refinement provides a simple way to integrate proof into the development of complex systems and conforms to the view of systems through different abstractions, thanks to the stuttering [71].

3.2 Proof-based development in B

3.2.1 Refinement of B models

The refinement of a formal model allows one to enrich a model in a *step by step* approach. Refinement provides a way to construct stronger invariants and also to add details in a model. It is also used to transform an abstract model in a more concrete version by modifying the state description. This is essentially done by extending the list of state variables (possibly suppressing some of them), by refining each abstract event into a corresponding concrete version, and by adding new events. The abstract state variables, x, and the concrete ones, y, are linked together by means of a, so-called, *gluing invariant* $J(x, y)$. A number of proof obligations ensure that (1) each abstract event is correctly refined by its corresponding concrete version, (2) each new event refines $skip$, (3) no new event take control for ever, and (4) relative deadlockfreeness is preserved. We detail proof obligations of a refinement after the introduction of the syntax of a refinement:

refinement
r
refines
m
sets
t
constants
d
properties
$Q(t, d)$
variables
y
invariant
$J(x, y)$
variant
$V(y)$
assertions
$B(y)$
initialisation
$y : INIT(y)$
events
<list of events>
end

A *refinement* has a name r; it is a model refining a model m in the clause **refines** and m can be a refinement. New sets, new constants and new properties can be declared in the clauses **sets**, **constants** or **properties**. New variables y are declared in the clause **variables** and are the concrete variables; variables x of the refined model m are called the abstract variables. The gluing invariant defines a mapping between abstract variables and concrete ones; when a concrete event occurs, there must be a corresponding one in the abstract model: the concrete model *simulates* the abstract model. The clause **variant** *controls* new events, which can not take the control over others events of the system. In a refinement, new events may appear and are refining an event SKIP; events of the refined model can be strengthened and one should prove that the new model does not contain more deadlock configurations than the refined one: if a guard is strengthened too much, it can lead to a dead refined event.

The refinement r of a model m is a system; its trace semantics is based on traces of states over variables x and y and the projection of concrete traces on abstract traces is a stuttering-free traces semantics of the abstract model. The mapping between abstract and concrete traces is called a refinement mapping by Lamport [71] and the stuttering is the key concept for refining events systems. When an event e of m is triggered, it modifies variables y and the abstract event refining e modifies x. Proof obligations make precise the relationship between abstract model and concrete model.

The abstract system is m and the concrete system is r; $INIT(y)$ denotes the initial condition of the concrete model; $I(x)$ is the invariant of the refined model m; $BAC(y, y')$ is the concrete before-after relation of an event of the concrete system r and $BAA(x, x')$ is the abstract before-after relation of the event of the abstract system m; $G_1(x)$, $\ldots G_n(x)$ are the guards of the n abstract events of m; $H_1(y)$, $\ldots$, $H_k(y)$ are the guards of k concrete events of r. Formally, the refinement of a model is defined as follows:

- (REF1) $INIT(y) \Rightarrow \exists x \cdot (Init(x) \wedge J(x, y))$:

  The initial condition of the refinement model imply that there exists an abstract value in the abstract model such that that value satisfies the initial conditions of the abstract one and implies the new invariant of the refinement model.

- (REF2) $I(x) \wedge J(x, y) \wedge BAC(y, y') \Rightarrow \exists x' . (BAA(x, x') \wedge J(x', y'))$:

  The invariant in the refinement model is preserved by the refined event and the activation of the refined event triggers the corresponding abstract event.

- (REF3) $I(x) \wedge J(x, y) \wedge BAC(y, y') \Rightarrow J(x, y')$:

  The invariant in the refinement model is preserved by the refined event but the event of the refinement model is a new event which was not visible in the abstract model; the new event refines $skip$.

- (REF4) $I(x) \ \wedge \ J(x,y) \ \wedge \ (G_1(x) \vee \ldots \vee G_n(x)) \ \Rightarrow \ H_1(y) \vee \ldots \vee H_k(y)$:

  The guards of events in the refinement model are strengthened and we have to prove that the refinement model is not more blocked than the abstract.

- (REF5) $I(x) \ \wedge \ J(x,y)) \ \Rightarrow \ V(y) \in \mathbb{N}$ and
- (REF6) $I(x) \ \wedge \ J(x,y) \ \wedge \ BAC(y,y') \ \Rightarrow \ V(y') < V(y)$:

  New events should not block forever abstract ones.

The refinement of models by refining events is close to the refinement of action systems [25], the refinement of UNITY and the TLA refinement, even if there is no explicit semantics based on traces but one can consider the refinement of events like a relation between abstract traces and concrete traces. The stuttering plays a central role in the global process of development where new events can be added into the refinement model. When one refines a model, one can either refine an existing event by strengthening the guard or/and the before-after predicate (removing non-determinism), or add a new event which is supposed to refine the skip event. When one refines a model by another one, it means that the set of traces of the refined model contains the traces of the resulting model with respect to the stuttering relationship. Models and refined models are defined and can be validated through the proofs of proof obligations; the refinement supports the proof-based development and we illustrate it by a case study on the development of a program for computing the $factorial$ function.

3.2.2 Proof-based development in action

The B language of predicates, the B language of events, the B language of models and the B refinement constitute the B method; however, the objectives of the B method are to provide a framework for developing models and finally programs. The development is based on proofs and should be validated by a tool. The current version of Atelier B groups B models into projects; a project is a set of B models related to a given problem. The statement of the problem is expressed in a mathematical framework defined by constants, properties, structures and the development of a problem starts from a very high level model which is simply stating the problem in an event-based style. The proof tool is central in the B method, since it allows us to write models and to validate step-by-step each decision of development; it is an assistant used by the user to integrate decisions into the models, especially by refining them. The proof process is fundamental and the interaction of a user in the proof process is a very critical point. We examine the different aspects of the development by an example. The problem is to compute the value of the $factorial$ function for a given data n. We have already proved that the (mathematical) $factorial$ function exists and we can reuse its definition and its properties. Three successive models are provided by development, namely $Fac1$ (the initial model stating in one-shot the computation of $factorial(n)$), $Fac2$ (refinement of the model $Fac1$ computing step by step $factorial(n)$), $Fac3$ (completing the development of an algorithm for $factorial(n)$).
We begin by writing a first model which is re-phrasing the problem and we simply state that an event is calculating the value $factorial(n)$ where n is a natural number. The model has only one event and is the one-shot model:

```
computation =
  begin
    fn := factorial(n)
  end
```

fn is the variable containing the value computed by the program; the expression *one-shot* means that we show a solution just by assigning the value of mathematical function to fn. It is clear that the one-shot event is not satisfactory, since it does not describe the algorithmic process for computing the result. Proofs are not difficult, since they are based on the properties stated in the preliminary part. Our next model will be a refinement of *Fac1*. It will introduce an iterative process of computation based on the mathematical definition of $factorial$. We therefore add a new event $prog$ which is extending the partial function under

construction called fac that contains a partial definition of the $factorial$ function. The initialisation is simply to set fac to the value for 0.

$$fac := \{0 \mapsto 1\}$$

and there is a new event progress which simulates the progress by adding the next pair in the function *fac*.

```
prog =
  when n ∉ DOM(fac) then
    any x where
      x ∈ ℕ ∧
      x ∈ DOM(fac) ∧
      x+1 ∉ DOM(fac)
    then
      fac(x+1) := (x+1)×fac(x)
    end
  end
```

Secondly, the event computation is refined by the following event stating that the process stops when the fac variable is defined for n.

```
computation =
  when n ∈ DOM(fac) then
    fn := fac(n)
  end;
```

The computation is based on the calculation of the fixed-point of the equation defining *factorial* and the ordering is the set inclusion over domains of functions; *fac* is a variable satisfying the following invariant property:

$$fac \in \mathbb{N} \nrightarrow \mathbb{N} \wedge fac \subseteq factorial \wedge \mathsf{dom}(fac) \subseteq 0..n \wedge \mathsf{dom}(fac) \neq \emptyset$$

fac is a relation over natural numbers and it contains a partial definition of the $factorial$ function; as long as n is not defined for fac, the computing process adds a new pair in fac. The system is deadlock-free, since the disjunction of the guards $n \in \mathsf{dom}(fac)$, or $n \notin \mathsf{dom}(fac)$ is trivially true. The event *progress* increases the domain of *fac*: $\mathsf{dom}(fac) \subseteq 0..n$. The proof obligations for the refinement are effectively proved by the proof tool:

$$n \in \mathsf{dom}(fac) \vee (n \notin \mathsf{dom}(fac) \wedge \exists x.(x \in \mathbb{N} \wedge x \in \mathsf{dom}(fac) \wedge x{+}1 \notin \mathsf{dom}(fac)))$$

The model is more algorithmic than the first one and it can be refined into a third one called *Fac3* closer to the classical algorithmic solution. Two new variables are introduced: a variable *i* plays the role of index and a variable *fq* is an accumulator. A gluing invariant define relations between old and new variables:

$$i \in \mathbb{N} \wedge 0..i = \mathsf{dom}(fac) \wedge fq = fac(i)$$

The two events of the second model are refined into the two next events.

```
computation =
  when i = n then
    fn := fq
  end;
prog =
  when i ≠ n then
    i := i+1 || fq := (i+1)×fq
  end
```

Proof obligations are completely discharged with the proof tool and we derive easily the algorithm by analysing guards of the last model.

$i := 0 \parallel fq := 1$
while $i \neq n$ **do**
$i := i+1 \parallel fq := (i+1) \times fq$
end
$fn := fq$

We can simplify the algorithm by removing the parallel operator and we transform it as follows:

$i := 0;$
$fq := 1;$
while $i \neq n$ **do**
$i := i+1;$
$fq := i \times fq$
end

Case studies provide information over the development process; different domains have been considered for illustrating the event-based B approach: sequential programs [5, 11], distributed systems [2, 12, 41, 45, 18], circuits [4], information systems [46]. In the next sections, we illustrate the event B modelling method by case studies:

- Sequential algorithms
- Combining coordination and refinement for sorting
- Spanning trees algorithms
- A distributed leader election algorithm

Chapter 4

Sequential algorithms

4.1 Primitive recursive functions

4.1.1 The class of primitive recursive functions

In the computability theory [90], the primitive recursive functions contitute a strict sub-class of general recursive functions also called the class or computable functions. Many computable functions are primitive recursive as the addition, the multiplication, the exponetitaion, the sign, ...; in fact, a primitive function corresponds to a bounded (for) loop and we show how to derive the (for) algorithm from the definition of the primitive recursive function.
The primitive recursive functions are defined by initial functions (the 0-place zero function ζ, the k-place projection function ${\pi_i}^k$, the successor function σ) and by two combining rules, namely the composition rule and the primitive recursive rule. More precisely, we give the definition of functions and rules:

- $\zeta() = 0$
- $\forall i \in \{1, \ldots, k\} : \forall x_1, \ldots, x_k \in \mathbb{N} : {\pi_i}^k(x_1, \ldots, x_k) = x_i$
- $\forall x \in \mathbb{N} : \sigma(n) = n{+}1$
- If g is a l-place function, if $h_1, \ldots, h_l$ are n-place functions and if the function f is defined by:

  $\forall x_1, \ldots, x_n \in \mathbb{N} : f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_l(x_l, \ldots, x_n))$,

  then f is obtained from g and $h_1, \ldots, h_l$ by composition.
- If g is a l-place function, if h is a $(l{+}2)$-place function and if the function f is defined by: $\forall x_1, \ldots, x_l, x \in \mathbb{N}$,

  $$\begin{cases} f(x_1, \ldots, x_l, 0) & = & g(x_1, \ldots, x_l) \\ f(x_1, \ldots, x_l, x{+}1) & = & h(x_1, \ldots, x_l, x, f(x_1, \ldots, x_l, x)) \end{cases},$$

  then f is obtained from g and h by primitive recursion.

A function f is primitive recursive, if it is an initial function or can be generated from initial functions by some finite sequence of the operations of composition and primitive recursion. A primitive recursive function is computed by an iteration and we define a general framework for stating the development of functions defined by primitive recursion using predicate diagrams.

4.1.2 Modeling the computation of a primitive recursive function

The first step is to define the mathematical function to compute the value of $f(u, v)$ where u and v are two natural numbers; the primitive recursive rule is stated as follows:

CONSTANTS
u, v, g, h, f

- u, v, g, h, f are constants corresponding to values and functions.
- u, v, g, h are supposed to be given.
- g, h are total and two primitive recursive functions.
- f is defined by a fixed-point-based rule.

PROPERTIES
$u \in \mathbb{N} \wedge$
$v \in \mathbb{N} \wedge$
$g \in \mathbb{N} \longrightarrow \mathbb{N} \wedge$
$h \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N} \wedge$
$f \in \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N} \wedge$
$\forall (a, b).\ (\ (a \in \mathbb{N} \wedge b \in \mathbb{N}) \Rightarrow (f(a, 0) = g(a))) \wedge$
$\forall (a, b).\ (\ (a \in \mathbb{N} \wedge b \in \mathbb{N}) \Rightarrow (f(a, b{+}1) = h(a, b, f(a, b)))) \wedge$

From the characterization of the constants, the totality of f is derived, since both g and h are total. The reader should be very careful on the functional notation $f(a, 0)$ which intends to mean the functional application but also the membership $(a, 0) \in f$, when f is not yet proved to be functional. The system uses three variables: two variables are the input values and the third one is the output value: VARIABLES $result$.

The required properties are the invariance of the **INVARIANT** clause and the partial correctness of the system with respect to the pre and postconditions of the computation of the function defined by the primitive recursion rule. The invariant property is very simple to establish:

INVARIANT
result $\in \mathbb{N}$
INITIALIZATION
result $:\in \mathbb{N}$

The INVARIANT clause is very simple for the first model and is in fact a typing invariant.

computation $=$
begin
result $:=$ $f(u, v)$
end

The first model has only one visible event and others events are hidden by the stuttering step; the $computation$ event models or simulates the computation of the resulting value and it simulates the end of a hidden loop. The loop will appear in the further model which is a refinement of the current one.

4.1.3 Iterative Computations from Primitive Recursion

The next model $primrec1$ is a refinement of $primrec0$; it introduces a new event called $step$ and $step$ is simulating the progression of an iterative process satisfying a loop invariant.

REFINEMENT $primrec1$
REFINES $primrec0$
VARIABLES $cx, cy, cresult, result$
INVARIANT
$cx \in \mathbb{N} \wedge$
$cy \in \mathbb{N} \wedge$
$cresult \in \mathbb{N} \wedge$
$cresult = f(cx, cy) \wedge$
$cx = u \wedge$
$0 \leq cy \wedge$
$cy \leq v$
INITIALISATION
$cx := u \parallel cy := 0 \parallel cresult := g(u) \parallel$
$result :\in \mathbb{N}$
EVENTS
computation =
when
$v - cy = 0$
then
$result := cresult$
end;
step =
when
$v - cy \neq 0$
then
$cy := cy + 1 \parallel$
$cresult := h(cx, cy, cresult)$
end
end

The new system has two visible events:

1. The first event $computation$ intends to model the end of the iteration and it concretizes the event $computation$.

2. The second event $step$ is the visible underlying step of the previous stuttering step.

The computation process is organized by the two guards of the two events; it leads us to the following algorithm, which captures the essence of the last B models.

precondition : $u, v \in \mathbb{N}$
postcondition : $result = f(u, v)$
local variables : $cx, cy, cresult \in \mathbb{N}$

$cx := u$;
$cy := 0$;
$cresult := G(u)$;
while $cy \leq v$ **do**
  Invariant : $0 \leq cy \wedge cy \leq v \wedge cx = u \wedge cresult = f[cx, cy]$
  $cresult := H[cx, cy, cresult]$;
  $cy := cy+1$;
;
$result := cresult$;

Algorithme 1: Iterative algorithm F for computing the primitive recursive function f

The final development includes two B models related by the refinement relationship and provides an algorithm for computing the specified function. The resulting algorithm is called F and it uses the algorithms of g and h. The invariant is derived from the B model and does not need further proofs. The development can be instantiated with respect to functions g and h which are supposed to be primitive recursive.

4.1.4 Applying the Development for Addition, Multiplication, Exponentiation

Addition

The mathematical function $addition$ is defined by the following rules:

$$\forall x, y \in \mathbb{N} : \left\{ \begin{array}{lcl} addition(x, 0) & = & {\pi_1}^1(x) \\ addition(x, y{+}1) & = & \sigma(addition(x, y)) \end{array} \right. ,$$

We assign to g the primitive recursive function ζ and to h the primitive recursive function σ; the primrec development can be replayed. The resulting algorithm is given by substituting g and h respectively by ζ and σ. The algorithm is denoted $ADDITION$.

precondition : $x, y \in \mathbb{N}$
postcondition : $result = ADDITION(x, y)$
local variables : $cx, cy, cresult \in \mathbb{N}$

$cx := x$;
$cy := 0$;
$cresult := {\pi_1}^1(x)$;
while $cy \leq y$ **do**
  Invariant : $0 \leq cy \wedge cy \leq y \wedge cx = x \wedge cresult = ADDITION[cx, cy]$
  $cresult := \sigma[cresult]$;
  $cy := cy{+}1$;
;
$result := cresult$;

Algorithme 2: Iterative algorithm $ADDITION$ for computing the primitive recursive function $addition$

Multiplication

The mathematical function $multiplication$ is defined by the following rules:

$$\forall x, y \in \mathbb{N} : \left\{ \begin{array}{lcl} multiplication(x, 0) & = & \zeta() \\ multiplication(x, y{+}1) & = & addition(x, multiplication(x, y)) \end{array} \right. ,$$

We assign to g the primitive recursive function $\zeta()$ and to h the primitive recursive function $addition$; the primrec development can be replayed. The resulting algorithm is given by substituting g and h respectively by ${\pi_1}^1$ and $addition$. The algorithm is denoted $MULTIPLICATION$.

Exponentiation

The mathematical function $exponentiation$ is defined by the following rules:

$$\forall x, y \in \mathbb{N} : \left\{ \begin{array}{lcl} exponentiation(x, 0) & = & \sigma(\zeta()) \\ exponentiation(x, y{+}1) & = & multiplication(x, exponentiation(x, y)) \end{array} \right. ,$$

We assign to g the primitive recursive function $\sigma(\zeta())$ (since the composition of two primitive recursive functions is still primitive recursive) and to h the primitive recursive function $multiplication$; the primrec development can be replayed. The resulting algorithm is given by substituting g and h respectively by $\sigma(\zeta())$ and $multiplication$. The algorithm is denoted $EXPONENTIATION$.

precondition : $x, y \in \mathbb{N}$
postcondition : $result = multiplication(x, y)$
local variables : $cx, cy, cresult \in \mathbb{N}$

$cx := x$;
$cy := 0$;
$cresult := \zeta()$;
while $cy \leq y$ **do**
    Invariant : $0 \leq cy \wedge cy \leq y \wedge cx = x \wedge cresult = multiplication[cx, cy]$
    $cresult := addition[cx, cresult]$;
    $cy := cy{+}1$;
;
$result := cresult$;

Algorithme 3: Iterative algorithm $MULTIPLICATION$ for computing the primitive recursive function $multiplication$

precondition : $x, y \in \mathbb{N}$
postcondition : $result = exponentiation(x, y)$
local variables : $cx, cy, cresult \in \mathbb{N}$

$cx := x$;
$cy := 0$;
$cresult := \sigma(\zeta())$;
while $cy \leq y$ **do**
    Invariant : $0 \leq cy \wedge cy \leq y \wedge cx = x \wedge cresult = exponentiation[cx, cy]$
    $cresult := MULTIPLICATION[cx, cresult]$;
    $cy := cy{+}1$;
;
$result := cresult$;

Algorithme 4: Iterative algorithm $EXPONENTIATION$ for computing the primitive recursive function $exponentiation$

4.2 Other ways to compute addition and multiplication

If we consider the development for the functions $addition$ and $multiplication$, we can reuse the first model of each one and improve the final resulting algorithms. We assume that the mathematical functions are supported by the B prover and we do not need to define them. The proved models can be reused in other developments and we are going to refine, in a different way, both functions.

4.2.1 Developing a new multiplication algorithm

The first model states the problem to solve namely the multiplication of two natural numbers; the second one provides the essence of the algorithmic solution and the last one implements naturals by sequences of digits. Let a and b two naturals. The problem is to compute the value of the expression $a\cdot$, where $\cdot$ is the mathematical function standing for natural multiplication. The function $multiplication$ is defined by an infix operator $\cdot$. The first model is a *one-shot* model computing in one step the result.

MODEL $multiplication0$
CONSTANTS a, b
PROPERTIES
$a \in \text{NAT} \wedge$
$b \in \text{NAT}$
VARIABLES
x, y, m
INVARIANT
$x \in \mathbb{N} \wedge$
$y \in \mathbb{N} \wedge$
$x = a \wedge$
$y = b \wedge$
$m \in \mathbb{N}$
INITIALISATION
$x := a \parallel y := b \parallel m :\in \mathbb{N}$
EVENTS
computation $=$
begin
$m := a \cdot b$
end
end

Now, we should get an idea and apply it on the model $multiplication0$. There are several ways to define the multiplication: either $(a{-}1){\cdot}b$ (primitive recursive function) or $a{\cdot}b = (2{\cdot}a){\cdot}(b/2)$. We choose the last one, since it is the faster one and simple to implement. We define two new variables namely cx et cy, for taking care of initial values of a and b (values-passing mechanism). The induction step will be driven by B which is strictly decreasing. The new variable M stores any value of cx when cy is odd.

VARIABLES
cx, cy, x, y, M, m
INVARIANT
$cx \in \mathbb{N} \wedge$
$cy \in \mathbb{N} \wedge$
$M \in \mathbb{N} \wedge$
$cx \cdot cy + M = x \cdot y$
INITIALISATION
$cx, cy, x, y, m :\in (x = a \wedge y = b \wedge cx = a \wedge cy = b \wedge m \in \mathbb{N}) \parallel$
$M := 0$

The event $computation$ occurs, when cy is equal to 0. The gluing invariant allows us to conclude that M contains the value of $a\cdot$.

computation =
when
$(cy = 0)$
then
$m := M$
end

Two new events $prog1$ and $prog2$ will help in the progression of cy towards 0.

$prog1$ =
when
$(cy \neq 0) \wedge even(cy)$
then
$cx := cx \cdot 2 \parallel cy := cy/2$
end

$prog2$ =
when
$(cy \neq 0) \wedge odd(cy)$
then
$cx := cx{\cdot}2 \parallel$
$cy := cy/2 \parallel$
$M := M{+}cx$
end

Where $even(cy) = \exists x \cdot (x \in \mathbb{N} \wedge cy = 2{\cdot}x)$ and $odd(cy) = \exists x \cdot (x \in \mathbb{N} \wedge cy = 2{\cdot}x{+}1)$. The proofs are not hard; Atelier B generated 18 proof obligations only 3 are discharged interactively. Finally, we obtain the model $multiplication1$:

REFINEMENT $multiplication1$
REFINES $multiplication0$
VARIABLES
cx, cy, x, y, M, m
INVARIANT
$cx \in \mathbb{N} \wedge$
$cy \in \mathbb{N} \wedge$
$M \in \mathbb{N} \wedge$
$cx \cdot cy + M = x \cdot y$
INITIALISATION
$cx, cy, x, y, m :\in (x = a \wedge y = b \wedge cx = a \wedge cy = b \wedge m \in \mathbb{N}) \parallel$
$M := 0$
EVENTS

computation =
when
$(cy = 0)$
then
$m := M$
end

$prog1$ =
when
$(cy \neq 0) \wedge even(cy)$
then
$cx := cx \cdot 2 \parallel cy := cy/2$
end

$prog2$ =
when
$(cy \neq 0) \wedge odd(cy)$
then
$cx := cx{\cdot}2 \parallel$
$cy := cy/2 \parallel$
$M := M{+}cx$
end

end

A further refinement may lead to the implementation of natural numbers by sequences of digits; The division and the multiplication by two are implemented by shifting digits. On the other hand, one can derive a well-known algorithm 5 for computing the multiplication function.

4.2.2 Addition of two natural numbers

The $addition$ function can also be redeveloped. The development is decomposed into three steps. The first step writes a *one-shot* model computing in one step the required result, namely the addition of two natural numbers. Let a and b be two naturals. The problem is to compute the value of the expression $a{+}b$, where $+$ is the mathematical function standing for the natural addition.

precondition : $a, b \in \mathbb{N}$
postcondition : $m = multiplication(x, y)$
local variables : $cx, cy, x, y, m, M \in \mathbb{N}$

$x := a$;
$y := b$;
$cx := x$;
$cy := y$;
$M := 0$;
while $cy \neq 0$ **do**
  Invariant : $0 \leq M \wedge 0 \leq cy \wedge cy \leq y \wedge cx{\cdot}cy{+}M = x{\cdot}y \wedge x = a \wedge y = b$
  if $(cy \neq 0) \wedge even(cy)$ **then**
    $cx := cx{\cdot}2 || cy := cy/2$
  ;
  if $(cy \neq 0) \wedge odd(cy)$ **then**
    $cx := cx{\cdot}2 || cy := cy/2 || M := M{+}cx$
  ;
;
$m := M$;

Algorithme 5: New Iterative algorithm $MULTIPLICATION$ for computing the primitive recursive function $multiplication$

MODEL $addition0$
CONSTANTS a, b
PROPERTIES
  $a \in \text{NAT} \wedge$
  $b \in \text{NAT}$
VARIABLES
  $x, y, result$
INVARIANT
  $x \in \mathbb{N} \wedge$
  $y \in \mathbb{N} \wedge$
  $x = a \wedge$
  $y = b \wedge$
  $result \in \mathbb{N}$
INITIALISATION
  $x := a \parallel y := b \parallel result :\in \mathbb{N}$
EVENTS
computation $=$
  begin
    $result := a \circ b$
  end
end

The definition of $a{+}b$ using $a/2$ (and $b/2$) is based on the following properties:

a	b	$a{+}b$
$2{\cdot}n$	$2{\cdot}m$	$2{\cdot}(n{+}m)$
$2{\cdot}n$	$2{\cdot}m{+}1$	$2{\cdot}(n{+}m){+}1$
$2{\cdot}n{+}1$	$2{\cdot}m$	$2{\cdot}(n{+}m){+}1$
$2{\cdot}n{+}1$	$2{\cdot}m{+}1$	$2{\cdot}(n{+}m){+}2$

Using the four properties, we try to obtain a general induction schema verified by variables and the four

properties lead to the general form: $(a{+}b){\cdot}C{+}P$. The discovery of the relation is based on the analysis of possible transformations over variables; Manna [78] has given hints for stating an inductive assertion from properties over values of variables. Associativity and the commutativity of the mathematical addition justify the form. Moreover, the form can also be justified by the binary coding of A and B as follows:

$$\left(\sum_{i=0}^{n} A_i 2^i\right) + \left(\sum_{i=0}^{n} B_i 2^i\right) = \sum_{i=0}^{n} (A_i{+}B_i)\, 2^i \tag{4.1}$$

$$\sum_{i=0}^{n} (A_i{+}B_i)\, 2^i = \left(\left(\sum_{i=1}^{n} A_i 2^{i-1}\right) + \left(\sum_{i=1}^{n} B_i 2^{i-1}\right)\right).2{+}(A_0{+}B_0) \tag{4.2}$$

$$\left(\sum_{i=0}^{n} A_i 2^i\right) + \left(\sum_{i=0}^{n} B_i 2^i\right) = \left(\left(\sum_{i=1}^{n} A_i 2^{i-1}\right) + \left(\sum_{i=1}^{n} B_i 2^{i-1}\right)\right).2{+}(A_0{+}B_0) \tag{4.3}$$

The last equation 4.3 tells us that we obtain a binary addition of the last digits of the two numbers and we have to store powers of 2, while computing. Two new variables are introduced: C for storing the powers of 2 and P for storing the partial result. We derive the following invariant and the initial conditions:

variables
A, B, P, a, b, p, C
invariant
$A \in \mathbb{N} \ \wedge \ B \in \mathbb{N} \ \wedge \ P \in \mathbb{N} \ \wedge \ C \in \mathbb{N} \ \wedge$
$(A{+}B){\cdot}C{+}P = a{+}b$
initialisation
$a, b, A, B, p : (a \in \mathbb{N} \ \wedge \ b \in \mathbb{N}$
$\wedge \ p \in \mathbb{N} \ \wedge \ P \in \mathbb{N} \ \wedge \ C \in \mathbb{N} \ \wedge \ A = a \ \wedge \ B = b) \ ||$
$P, C := 0, 1$

The one-shot event of the previous model is then refined by the next event; the result is in the variable P, when A and B are two variables containing 0.

$add \ \widehat{=}$
when $(B = 0) \ \wedge \ (A = 0)$
then
$p := P$
end;

Four new events are added to the current model; each event corresponds to a case of properties given in the array above. Four cases are under consideration. The four new events introduced in this model are the following

$prog1 =$
when
$even(A) \ \wedge \ even(B)$
then
$A := A/2 \ || \ B := B/2 \ ||$
$C := 2{\cdot}C$
end;

$prog2 =$
when
$odd(A) \ \wedge \ even(B)$
then
$A := A/2 \ || \ B := B/2 \ ||$
$C := 2{\cdot}C \ || \ P := C{+}P$
end;

$prog3 =$
when
$even(A) \ \wedge \ odd(B)$
then
$A := A/2 \ || \ B := B/2 \ ||$
$C := 2{\cdot}C \ || \ P := C{+}P$
end;

$prog4 =$
when
$odd(A) \ \wedge \ odd(B)$
then
$A := A/2 \ || \ B := B/2 \ ||$
$C := 2{\cdot}C \ || \ P := 2{\cdot}C{+}P$
end

We have to code basic operations for computing $C{+}P$, $2{\cdot}C$ and $2{\cdot}C{+}P$. $C{+}P$ is solved by storing a 1 digit in the corresponding location. $2{\cdot}C$ is a shifting operation. $2{\cdot}C{+}P$ is solved by managing a carry. Now, we can refine the current model.

4.2.3 Managing the carry

The goal of the carry is to implement the basic operation $2{\cdot}C{+}P$; P is concretized by the store Q and the carry R.

variables
A, B, Q, R, a, b, p, C
invariant
$Q \in \mathbb{N} \ \wedge \ R \in \mathbb{N} \ \wedge \ (R = 0 \vee R = 1) \ \wedge$
$P = C{\cdot}R{+}Q$
initialisation
$a, b, A, B, p : (a \in \mathbb{N} \ \wedge \ b \in \mathbb{N} \ \wedge \ p \in \mathbb{N} \ \wedge \ P \in \mathbb{N} \ \wedge \ C \in \mathbb{N} \ \wedge \ A = a \ \wedge \ B = b) \ ||$
$Q, R, C := 0, 0, 1$

The refined event add uses the new variables Q and C. The gluing invariant (thanks to it) maintains the relationship over P and the new variables.

$add =$ **when** $(B = 0) \ \wedge \ (A = 0)$ **then** $p := C{\cdot}R{+}Q$ **end**;

Events $prog1$, $prog2$, $prog3$, $prog4$ are refined and modified by introducing the two new variables. The new variables are modified according to P.

$prog1 =$
when
$even(A) \ \wedge \ even(B)$
then
$A := A/2 \ || \ B := B/2 \ || \ C := 2{\cdot}C \ ||$
$Q := C{\cdot}R{+}Q \ || \ R := 0$
end;

$prog4 =$
when
$odd(A) \ \wedge \ odd(B)$
then
$A := A/2 \ || \ B := B/2 \ || \ C := 2{\cdot}C \ ||$
$Q := C{\cdot}R{+}Q \ || \ R := 1$
end

$prog2 =$
when
$odd(A) \ \wedge \ even(B)$
then
$A := A/2 \ || \ B := B/2 \ || \ C := 2{\cdot}C \ ||$
if $R = 0$ **then**
$Q := C{+}Q$
end
end;

$prog3 =$
when
$even(A) \ \wedge \ odd(B)$
then
$A := A/2 \ || \ B := B/2 \ || \ C := 2{\cdot}C \ ||$
if $R = 0$ **then**
$Q := C{+}Q$
end
end;

This model is validated by the tool Atelier B [55] which generate 56 proof obligations and 15 are discharched interactively. Details are incrementally added; each model provides a view of the computing function. The models are related by the refinement relationship and the last model can now be refined to produce codes.

4.2.4 Production of codes

The refinement process leads to basic operations over natural numbers that can be implemented by operations over bits. The B language provides sequences but experience shows that proofs are harder when sequences are used in a given model and we use the following definitions of sequences:

sets
$bit = \{ZERO, ONE\}$
constants
$code$
properties
$code \in \mathbb{N}\times\mathbb{Z} \longrightarrow (\mathbb{Z} \nrightarrow bit) \ \wedge$
$\forall k \cdot (k \in \mathbb{Z} \Rightarrow code(0,k) = \emptyset \ \wedge$
$\forall (n,k) \cdot (n \in \mathbb{N} \ \wedge \ n \neq 0 \ \wedge \ k \in \mathbb{Z} \ \Rightarrow \ code(2{\cdot}n,k) = \{k \mapsto ZERO\} \cup code(n,k{+}1)) \ \wedge$
$\forall (n,k) \cdot (n \in \mathbb{N} \ \wedge \ k \in \mathbb{Z} \ \Rightarrow \ code(2{\cdot}n{+}1,k) = \{k \mapsto ONE\} \cup code(n,k{+}1))) \ \wedge$
$\forall (n,k,x) \cdot (n \in \mathbb{N} \ \wedge \ k \in \mathbb{Z} \ \wedge \ x \in \mathbf{dom}(code(n,k)) \ \Rightarrow \ x \geq k)$

The recursive definition is validated by our previous works [44] on the development of recursive functions using the B event-based method. We have defined schemas allowing to evaluate those functions. A sequence is coded by an integer interval. For instance, we give an example of the second model of the multiplication: shifting of digits is implemented by an insertion of 0 at the head of the sequence; removing a bit at the head corresponds to the multiplication by 2. Questions on the reusability and the decomposition of systems remain to be solved and will be part of further works making the method more practical.

variables
$A, B, P, a, b, p, cA, cB, kA, kB$
invariant
$kA \in \mathbb{Z} \ \wedge \ kB \in \mathbb{Z} \ \wedge$
$cA \in \mathbb{Z} \nrightarrow bit \ \wedge \ cA = code(A, kA) \ \wedge$
$cB \in \mathbb{Z} \nrightarrow bit \ \wedge \ cB = code(B, kB)$

$prog1 =$
when $(cB \neq \emptyset) \ \wedge \ cB(kB) = ZERO$ **then**
if $cA \neq \emptyset$ **then** $cA := \{kA{-}1 \mapsto ZERO\} \cup cA \ || \ kA := kA{-}1$ **end** ||
$cB := \{kB\} \lhd cB \ || \ kB := kB{+}1 \ || \ A := 2{\cdot}A \ || \ B := B/2$
end;

$prog2 =$
when $(cB \neq \emptyset) \ \wedge \ cB(kB) = ONE$ **then**
if $cA \neq \emptyset$ **then** $cA := \{kA{-}1 \mapsto ZERO\} \cup cA \ || \ kA := kA{-}1$ **end** ||
$cB := \{kB\} \lhd cB \ || \ kB := kB{+}1 \ || \ A := 2{\cdot}A \ || \ B := B/2 \ || \ M := M{+}A$
end

The coding allows us to implement the addition $C{+}Q$, since C is a power of two and since C is greater than Q.

$$code(C{+}Q, 0) = code(C, 0) \Leftarrow code(Q, 0)$$

These properties (and other ones) are really proved in another B machine using only the **properties** and **assertions** clauses like in the work on structure [9]. Atelier B generated 10 proof obligations which are discharged interactively. The reader can find this machine in the annex.
We can give a refinement of the addition but only two events are really given. cp is the code of p, cQ the code of Q and cC the code of C.

$add =$
when $cB = \emptyset \ \wedge \ cA = \emptyset$ **then**
if $R = 1$ **then** $cp := cC \Leftarrow cQ$
else $cp := cQ$
end
end;

$prog1 =$
when
$cB(kB) \neq ONE \ \wedge \ cA(kA) \neq ONE$ **then**
$cB := \{kB\} \lhd cB \ || \ kB := kB{+}1 \ || \ cA := \{kA\} \lhd cA \ || \ kA := kA{+}1 ||$
$cC := \{0 \mapsto ZERO\} \cup shift(cC) \ || \ R := 0 \ ||$
if $R = 1$ **then** $cQ := cC \Leftarrow cQ$ **end**
end;

The function $shift$ shifts any value of a sequence (to begin always by 0). Atelier B generated 95 proof obligations and 53 are discharched interactively but we can do better using the assertion clauses.
A stronger refinement can now be obtained from the current developed model. A coding on finite sequence of bits ($bs{+}1$) constrains the abstract code to contain a bounded number of bits. We consider the natural numbers a and b are codable and we obtain a concrete code for variables A and B, namely CA and CB.

$$CA, CB : (CA \in 0..bs \rightarrow bit \ \wedge \ CA = code(a,0) \cup ((0..bs) - \mathbf{dom}(code(a,0))) \times \{ZERO\} \ \wedge \\ CB \in 0..bs \rightarrow bit \ \wedge \ CB = code(b,0) \cup ((0..bs) - \mathbf{dom}(code(b,0))) \times \{ZERO\})$$

A variable K plays the role of kA and kB and the process halts, when k is $bs{+}1$. The gluing invariant for variables A, B, p and Q (Cp and CQ are the concrete code) is the following one:

$K \in 0..bs{+}1 \ \wedge \ K = kA \ \wedge \ K = kB \ \wedge \ LO \in -1..K{-}1 \ \wedge$
$CA \in 0..bs \rightarrow bit \ \wedge$
$((K..bs) \lhd CA) = cA \cup ((K..bs) - \mathbf{dom}(cA)) \times \{ZERO\} \ \wedge$
$CB \in 0..bs \rightarrow bit \ \wedge$
$((K..bs) \lhd CB) = cB \cup ((K..bs) - \mathbf{dom}(cB)) \times \{ZERO\} \ \wedge$
$Cp \in 0..bs{+}1 \rightarrow bit \ \wedge$
$CQ \in 0..bs \rightarrow bit \ \wedge$
$(0..LO \lhd CQ = cQ) \ \wedge$
$(LO \geq 0 \Rightarrow CQ(LO) = ONE) \ \wedge$
$\forall i \cdot (i \in (LO{+}1)..bs \Rightarrow CQ(i) = ZERO)$

Where LO is a new variable; it is the position of the last ONE in CQ. Events add and $prog1$ are refined in the following concrete events:

$add =$
when $K = bs{+}1$ **then**
if $R = 1$ **then** $Cp := CQ \Leftarrow \{bs{+}1 \mapsto ONE\}$
else $Cp := CQ \Leftarrow \{bs{+}1 \mapsto ZERO\}$
end
end;

$prog1 =$
when $K \leq bs \ \wedge \ CB(K) \neq ONE \ \wedge \ CA(K) \neq ONE$ **then**
$K := K{+}1 \ || \ R := 0 \ ||$
if $R = 1$ **then** $CQ(K) := ONE \ || \ LO := K$ **end**
end;

We have to express that the coding of the result is in $0..bs{+}1 \rightarrow bit$ and that it might have an overflow. Multiplication by two ($K := K{+}1$), division by 2 ($K := K{+}1$) and addition ($CQ(K) := ONE$) are implemented using this coding. Atelier B generated 81 proof obligations and 25 are discharched interactively.

4.2.5 Properties of models

In the following machine, we have proved all properties used on the abstract coding. Two induction theorems are also proved in this machine (the second and third assertion).

machine
$Code$
sets
$bit = \{ZERO, ONE\}$
constants
$divtwo, code, power2, suc, shift, pred1$
properties
Definition of $divtwo$
$divtwo \in \mathbb{N} \rightarrow \mathbb{N} \ \wedge$
$\forall x \cdot (x \in \mathbb{N} \Rightarrow divtwo(x) = x/2) \ \wedge$
Definition of suc (successor)
$suc \in \mathbb{N} \rightarrow \mathbb{N} \ \wedge$
$\forall x \cdot (x \in \mathbb{N} \Rightarrow suc(x) = x{+}1) \ \wedge$
Definition of $code$
$code \in \mathbb{N}{\times}\mathbb{Z} \rightarrow (\mathbb{Z} \leftrightarrow bit) \ \wedge$
$\forall k \cdot (k \in \mathbb{Z} \Rightarrow code(0, k) = \emptyset) \ \wedge$
$\forall (n, k) \cdot (n \in \mathbb{N} \ \wedge \ n \neq 0 \ \wedge \ k \in \mathbb{Z} \Rightarrow code(2{\cdot}n, k) = \{k \mapsto ZERO\} \cup code(n, k{+}1)) \ \wedge$
$\forall (n, k) \cdot (n \in \mathbb{N} \ \wedge \ k \in \mathbb{Z} \Rightarrow code(2{\cdot}n{+}1, k) = \{k \mapsto ONE\} \cup code(n, k{+}1))) \ \wedge$
Definition of $power2\,(2^n)$
$power2 \in \mathbb{N} \rightarrow \mathbb{N} \ \wedge$
$power2(0) = 1 \ \wedge$
$\forall k \cdot (k \in \mathbb{N} \Rightarrow power2(k{+}1) = 2{\cdot}power2(k)) \ \wedge$
Definition of $pred1$ (predecessor)
$pred1 \in \mathbb{Z} \rightarrow \mathbb{Z} \ \wedge$
$\forall x \cdot (x \in \mathbb{Z} \Rightarrow pred1(x) = x{-}1) \ \wedge$
Definition of $shift$ (shift code)
$shift \in (\mathbb{Z} \nrightarrow bit) \rightarrow (\mathbb{Z} \nrightarrow bit) \ \wedge$
$\forall y \cdot (y \in \mathbb{Z} \nrightarrow bit \Rightarrow shift(y) = (pred1; y))$
assertions
$\forall c \cdot (c \in \mathbb{N} \Rightarrow \exists y \cdot (y \in \mathbb{N} \ \wedge \ (c = 2{\cdot}y \ \vee \ c = 2{\cdot}y{+}1)));$
A number c is odd or even
$\forall P \cdot (P \subseteq \mathbb{N} \ \wedge \ 0 \in P \ \wedge \ suc[P] \subseteq P \ \Rightarrow \ \mathbb{N} \subseteq P);$
It's the recurrence theorem. P is the set of all value which satify a property
$\forall K \cdot (K \subseteq \mathbb{N} \ \wedge \ 0 \in K \ \wedge \ divtwo^{-1}[K] \subseteq K \ \Rightarrow \ \mathbb{N} \subseteq K);$
It's another recurrence theorem, like $P(n/2) \Rightarrow P(n)$ ..
$\forall (n, k, x) \cdot (n \in \mathbb{N} \ \wedge \ k \in \mathbb{Z} \ \wedge \ x \in \mathbf{dom}(code(n, k)) \ \Rightarrow \ x \geq k);$
All value in $\mathbf{dom}(code(n, k))$ are greater or equals than k
$code \in \mathbb{N}{\times}\mathbb{Z} \rightarrow (\mathbb{Z} \nrightarrow bit);$
Now a code is a partial function
$\forall n \cdot (n \in \mathbb{N} \Rightarrow power2(n) > 0);$
2^n is always greater than 0
$\forall (n, c, k) \cdot (n \in \mathbb{N} \ \wedge \ c \in \mathbb{N} \ \wedge \ power2(n) > c \ \wedge \ k \in \mathbb{Z} \Rightarrow$
$code(power2(n){+}c, k) = code(power2(n), k) \Leftrightarrow code(c, k));$
It's our property to implement the addition
$\forall (n, k, x) \cdot (n \in \mathbb{N} \ \wedge \ k \in \mathbb{Z} \ \wedge \ x \in \mathbf{dom}(shift(code(n, k))) \ \Rightarrow \ x > k);$
$\forall (n, k) \cdot (n \in \mathbb{N} \ \wedge \ k \in \mathbb{Z} \ \Rightarrow \ shift(code(n, k)) = code(n, k{+}1))$
A useful property of $shift$ (it's now a shift)
$\forall n \cdot (n \in \mathbb{N} \ \Rightarrow \ code(power2(n), 0) = (0..n{-}1){\times}ZERO{\cup}\{n \mapsto ONE\})$
A property which evaluates the code of 2^n
end

4.3 Design of sequential algorithms

The design of a sequential algorithm starts by the statement of the specification of the algorithm; the specification of the algorithm is expressed by a precondition over input data, a postcondition over output data and a relation betwenn input and output data. The extension of the guarded command language by C. Morgan [83] allows to initiate a development by refinement according to a set of rules. However, no mechanical tool allows one to check the refinement; the notation $x : [pre, post]$ intends to mean a statement which is correct with respect to the pre and post conditions. It is exactly the case, when one starts an event-B development, since one should state a *magical* event which is correct with respect to the pre and post conditions. If we consider $x : [pre, post]$ and if we assume that x is free in pre and $post$, $x : [pre, post]$ is a statement which may modify x but only x and which satisfies the HOARE triple:

$$\{pre\} \ \ x : [pre, post] \ \ \{post\} \tag{4.4}$$

An equivalent event is defined as follows:

event $=$
  any z
  where
    $pre(x) \wedge post(x, z)$
  then
    $x := z$
  end

We have illustrated the event B method by simple sequential algorithms and we have emphasized the possibility to reuse the previous development. In the next section, we developed a sorting algorithm.

Chapter 5

Combining coordination and refinement for sorting

The coordination paradigm improves the development of concurrent/distributed solutions, because it provides simple way to communicate between processes via a data structure called a tuple space. Coordination principles and event-driven system development principles can be fruitfully combined to develop systems and to analyse the development of different solutions of a given problem. Benefits are inherited from both frameworks: the B event-driven approach provides the refinement and the coordination framework provides a simple computation model. The sorting problem is redeveloped in the B event-driven method using coordination principles for algorithms and two programming paradigms are applied ie merging and splitting list to sort.

5.1 Introduction

Overview The coordination paradigm [92, 50] improves the development of concurrent/distributed solutions, because it provides simple way to communicate between processes via a data structure called a tuple space. Coordination and event-driven system development can be fruitfully combined to construct sequential recursive programs and to analyse the development of different solutions of a given problem, namely the sorting problem. The combination exploits the fundamental refinement relationship defined in the B event-driven approach [7, 22, 10, 3, 6, 12, 41, 11] and leads to a practical framework for addressing the analysis of programs development.

Coordination The coordination paradigm appears in different programming environments as LINDA [92, 50]; the main idea is really simple: a collection of processes or agents can cooperate, communicate and exchange data through a unique structure called a tuple space. A tuple space is a heap that can contains items and several operations are authorized to processes, namely to put an item in the tuple space, to withdraw an item or to consult. Implementation details are hidden. Any programming language can be extended by specific operations related to the tuple space, as for instance the C LINDA environment which extends the C programming language. The coordination paradigm focuses on the development of activities that are inherently concurrent and that are simply made coherent through the coordination primitives. As soon as a coordination program is written, tools as compilers provide a translation into a lower level which manages communications; it means that communications are used without toil, since we do not take care how communications are really implemented. The coordination computation model is developed in the GAMMA [26] model and a kernel of a methodology related to the proof if given; Chaudron [53] defines a refinement in a language of coordination for GAMMA close to techniques of bisimulation. We do not define new refinements. The CHAM (Chemical Abstract Machine) is a chemical view of the coordination computation model. However, even if GAMMA intends to promote the methodological aspects of programming development, nothing is clearly studied for the relationship with the refinement of events systems.

Integration of coordination and event-driven systems Event-driven systems are incrementally derived from a very abstract model into a final concrete model through refinement steps. The B event-driven technique is based on the validation by proof of each refinement step and it starts by a system analysis where mathematical details are carefully analysed and proved or disproved by the proof tool. The idea is to add the

coordination primitives as events which modify the tuple space an to get for free a refinement in the coordination framework. A consequence is to provide a way to execute event-driven systems as coordinative events set and to allows the refinement of general coordinative structures. This exercise focuses on the use of both techniques for analysing the sorting problem; we apply two main sorting paradigm namely the splitting (quicksort) or the merging. Finally, we obtain a final concrete model which is a sequential algorithm using a stack and which gives a non recursive algorithm in the quicksort family.
The coordination paradigm was introduced and implemented in LINDA [92, 50] and a C LINDA compiler was effectively developed. The original idea is to synchronise processes or agents through a shared data space called a tuple space, using specific primitives extending the programming language. The programming language can be C, SML or a Prolog-like one; coordination primitives manage communication among processes or agents. Coordination is information-driven and makes interaction protocols simple and expressive. For instance, the implementation of Galibert [64] provides a simple way to program in C++ and to use a powerful high performant computer namely the Origin 2000 SGI. Here, we use coordination as a simple way to state actions on data; it is a less structured approach contrary to classical programming languages. Every abstract model (in the B event-based approach) can be transformed into a coordinative program; however, we refine as much as possible to obtain a sequential algorithm.
When one write a coordinative program, one has to identify processes or agents of the system; processes are expressed in a programming notation and the coordination framework allows to state communications between processes through the tuple space. Coordination primitives includes the reading of a value in the tuple space, the writing of a value in the tuple space, the waiting of a value in the tuple space, ... Events play the rôle of actions of agents or processes and cooperate to the global computation, if any.

5.2 A famous case study: the sorting problem

Sorting a list of values means that one tries to find a permutation of values such that the resulting list is sorted. We define two constants, f and m, with the following properties:

$$m \in \mathbb{N}1 \wedge$$
$$f \in 1..m \rightarrowtail \mathbb{N}$$

f stands for the abstract array which contains m natural numbers. All elements of the list are different. The variable g initially set to the initial value f of the list, contains the sorted list in an ascending way. The invariant must state that values are preserved between g and f.

$$g \in 1..m \longrightarrow \mathbb{N} \wedge$$
$$\text{RAN}(g) = \text{RAN}(f)$$

The invariant holds at the beginning, since $g = f$; the unique event of the system is *sorting* and it sorts in one step g.

sorting = **begin**
g : ($g \in 1..m \rightarrowtail \mathbb{N} \wedge$
$\text{RAN}(g) = \text{RAN}(f) \wedge$
$\forall xx.(xx \in 1..m-1 \Rightarrow g(xx) \leq g(xx+1))$
)
end;

We know that there is one (and only one) permutation for sorting the list. The event *sorting* is then enabled. The simplicity of the sorting event allows us to derive the correctness of the abstract system. The sorting is done in one step, which may seem to be magical. The abstract system is refined into another event system which implements a sorting technique as for instance the quicksort, the merge sort, ... The main idea is to use the coordination paradigm to remove the recursiveness of the solution. The first abstract model is called BASIC_SORTING.

5.3 Applying two sorting paradigms

The previous system is an abstract view of the sorting process and sorting algorithms are based on specific paradigms leading to well known solutions. In our case, we consider two paradigms:

- MERGING TWO SORTED LISTS TO PRODUCE A SORTED LIST: merge sorts and insertion sorts use the basic technique of merging two sorted lists; the way for combining sorted lists may be different and the size of the two list may be also different. The insertion sort combines a list with only one element and any other sorted list. The Von Neuman sort combines two lists having the same size. Nevertheless, the basic technique is the merging of two sorted lists and the global process increments the size of intermediate lists, which is a termination condition.

- SPLITTING A LIST INTO TWO LISTS TO OBTAIN TWO PARTITIONED LISTS: on the contrary, a list can be splitted into two lists such that elements of the first list are smaller than elements of the second one; the famous quicksort is an application of the paradigm; the introduction of the pivot is very important for the complexity of the sort. The selection sort is another example of sorting technique and is an extreme case of the quicksort - ie the pivot is the extreme left or right position in the splitted list. The process converge to a list of one-element sorted lists, which are correctly located.

The coordination paradigm provides us a computation model and we use the event-driven paradigm for defining operations on the tuple space. The data structures are supported by the tuple space. A list is defined as an interval over the set of discrete values $1..m$ where m is a constant of the problem. An interval contains successive values, when non empty. An interval is a subset of $1..m$ with consecutive values and intervals are a partition of $1..m$. The invariant will be strengthened to take into account properties of intervals later. For the moment the following invariant says that the tuple space TS is a partition of $1..m$; operations on the tuple space are expressed by events modifying the variable TS.

$$
\begin{array}{l}
TS \subseteq \mathcal{P}(\ 1..m\ ) \land \\
\forall I.(I \in TS \Rightarrow I \neq \emptyset) \land \\
\forall (I, J).(I \in TS \land \\
\qquad J \in TS \land \\
\qquad I \neq J \\
\qquad \Rightarrow \\
\qquad\ I \cap J = \emptyset\ ) \land \\
\forall i.(i \in 1..m \Rightarrow \exists I.(I \in TS \land i \in I))
\end{array}
$$

The refinement of the current model BASIC_MODEL leads us either to split intervals, or to combine intervals; we obtain two possible refined models:

- MERGE_SORT *merging two intervals to produce an interval*: the sorting process will stop when only one interval is remaining in the tuple space.

- SPLIT_SORT *splitting an interval into two intervals* : the splitting sorting will stop when no more splitting will be possible.

We give no more details about the way intervals are chosen, since these details may appear later in the refinement process. Both models are still to refine to detail operations of merging and splitting. No implementation detail is addressing the problem of parallel execution, since it is an abstract model.

5.3.1 Bottom Up Process MERGE_SORT

The bottom up process combines intervals by maintaining the invariant of the sorting problem. The merging of two intervals assumes that the restriction of g on each interval is sorted. The property is added to the previous invariant.

$$
\begin{array}{l}
\forall I.(I \in TS \\
\quad \Rightarrow \\
\quad\quad \forall (i,j).(i \in I \wedge \\
\quad\quad\quad\quad\quad j \in I \wedge \\
\quad\quad\quad\quad\quad i \leq j \\
\quad\quad\quad\quad\quad \Rightarrow \\
\quad\quad\quad\quad\quad\quad g(i) \leq g(j)))
\end{array}
$$

Initial conditions state that the tuple space contains only intervals with one element; there is an interval for every possible values of $1..m$; g is set to the initial value of the list to sort.

$$
\begin{array}{l}
Init \ = \ \textbf{begin} \\
\quad\quad\quad g \ := \ f \\
\quad\quad \| \ TS \ := \ \{x | x \subseteq 1..m \wedge \exists i.(i \in 1..m \wedge x = i..i)\} \\
\quad\quad \textbf{end}
\end{array}
$$

We recall that the merge process stops, when only one interval is in the tuple space and it contains only $1..m$. Using the invariant we can prove that g is sorted. The refined $sorting$ event is

$$
\begin{array}{l}
sorting \ = \ \textbf{when} \ 1..m \ \in \ TS \ \textbf{then} \\
\quad\quad\quad \text{SKIP} \\
\quad\quad \textbf{end};
\end{array}
$$

The sorting process is detailed in a way that identifies intermediate states of the variable g; these intermediate states state that the set of intervals is converging towards a unique interval modeling the sorted list. A progress event is defined to model the computation of a merging step. The new event $merge\_progress$ withdraws two intervals from *TS* and deposits a new interval which is the merging of the two withdrawn intervals in *TS*. The merging of two intervals decrements the number of intervals and helps in the convergence of the process.

$$
\begin{array}{l}
merge\_progress \ = \\
\quad\quad \textbf{any} \ I, \ J \ , \ gp \ \textbf{where} \\
\quad\quad\quad I \ \in \ TS \wedge \\
\quad\quad\quad J \ \in \ TS \wedge \\
\quad\quad\quad I \ \neq \ J \wedge \\
\quad\quad\quad gp \ \in \ I \cup J \ \longrightarrow \ \mathbb{N} \wedge \\
\quad\quad\quad \text{RAN}(gp) = \text{RAN}((I \cup J) \lhd g) \\
\quad\quad\quad \forall (i1, i2).(i1 \in \ I \cup J \ \wedge \\
\quad\quad\quad\quad\quad\quad i2 \in \ I \cup J \ \wedge \\
\quad\quad\quad\quad\quad\quad i1 \ \leq i2 \\
\quad\quad\quad\quad\quad\quad \Rightarrow \\
\quad\quad\quad\quad\quad\quad\quad gp(i1) \leq gp(i2)) \\
\quad\quad \textbf{then} \\
\quad\quad\quad g \ := \ g \ \lhd\!\!+ \ gp \ \| \\
\quad\quad\quad TS \ := \ TS - \{I, J\} \cup \{I \cup J\} \\
\quad\quad \textbf{end}
\end{array}
$$

The model is not yet the merging sort, since it is not efficiently implemented. However, the essence of the merging sort is expressed in the current model. Further refinements introduce details to obtain different sorting algorithms based on the merging paradigm, as the merging sort, the insertion sort or the Von Neumann sort. At this point, we not really an interval, since $I \cup J$ is not necessarily an interval, but a further refinement will be able to choose adequately intervals to satisfy that constraint.

5.3.2 Top Down SPLIT_SORT

The quicksort is based on a strategy of decomposition called splitting list and the refinement of the model BASIC_SORTING adds a new invariant expressing the states of intervals resulting from splitting them. The final goal is to obtain a tuple space containing only intervals with one element. Remember that the quicksort

splits an interval into two intervals in a way such that elements of the first interval are smaller than elements of the second one. The invariant is strengthened by the property, that intervals can be sorted with respect to their values.

$$
\begin{array}{l}
\forall(I,J).(I \in TS \wedge \\
\quad J \in TS \wedge \\
\quad I \neq J \\
\quad \Rightarrow \\
\quad\ (\forall(i,j).(i \in I \wedge \\
\qquad\qquad j \in J \wedge \\
\qquad\qquad i < j \\
\qquad\qquad \Rightarrow \\
\qquad\qquad\ g(i) \leq g(j))))
\end{array}
$$

When two numbers are in an interval, values between those two values are also in the interval.

$$
\begin{array}{l}
\forall I.(I \in TS \ \Rightarrow\ (\forall(i,j).(i \ \in\ I \\
\qquad\qquad\qquad\qquad\quad j \ \in\ I \\
\qquad\qquad\qquad\qquad\quad \Rightarrow \\
\qquad\qquad\qquad\qquad\quad\ i..j \ \subseteq\ I)))
\end{array}
$$

Initial conditions satisfy the invariant by setting a unique interval into the tuple space: only $1..m$ is in the tuple space.

$$
\begin{array}{l}
Init \ =\ \textbf{begin} \\
\qquad g \ :=\ f \\
\quad \parallel\ TS := \{1..m\} \\
\textbf{end}
\end{array}
$$

The split process starts in a tuple space with only one interval and halts, when every interval $i..i$ (for every value i in $1..m$) is in the tuple space. In fact, no more splitting event is possible.

$$
\begin{array}{l}
sorting \ =\ \textbf{when}\ \forall i.(i \in 1..m \Rightarrow\ i..i \in TS)\ \textbf{then} \\
\qquad \text{SKIP} \\
\textbf{end};
\end{array}
$$

The progress of the global process is achieved by splitting as long as possible intervals of the tuple space; only intervals with at least two elements can be splitted. The new event chooses a value called a pivot: it splits an interval into two smaller ones and it updates g. Obviously, the way to update g is very crucial for the implementation, as well as the choice of the pivot. The selection sorting is one possible refined model that can be derived, if the choice of the pivot is specially done: the pivot is the greatest or the smallest value of the interval.

$$
\begin{array}{l}
split\_progress \ = \\
\quad \textbf{any}\ I,\ k,\ gp,\ x\ \textbf{where} \\
\qquad I \ \in\ TS \wedge \\
\qquad k \ \in\ I \wedge \\
\qquad \exists j.(j \in I \wedge j > k) \wedge \\
\qquad gp \in\ I \ \longrightarrow\ \mathbb{N} \wedge \\
\qquad x \in \text{RAN}(gp) \wedge \\
\qquad \text{RAN}(gp) = \text{RAN}(I \lhd g) \wedge \\
\qquad \forall z.(z \in I \ \wedge z \leq k \Rightarrow gp(z) \ \leq\ x) \wedge \\
\qquad \forall z.(z \in I \ \wedge z > k \ \Rightarrow gp(z) \ \geq\ x) \\
\quad \textbf{then} \\
\qquad g :=\ g \lhd\!\!+ gp \\
\quad \parallel\ TS \ :=\ TS - \{I\} \cup \{\{y|y \in I \wedge y \leq k\}, \{y|y \in I \wedge y > k\}\} \\
\textbf{end}
\end{array}
$$

The model has two main events; one event splits the intervals as long as there is at least one interval with two values and an event for completing the process.

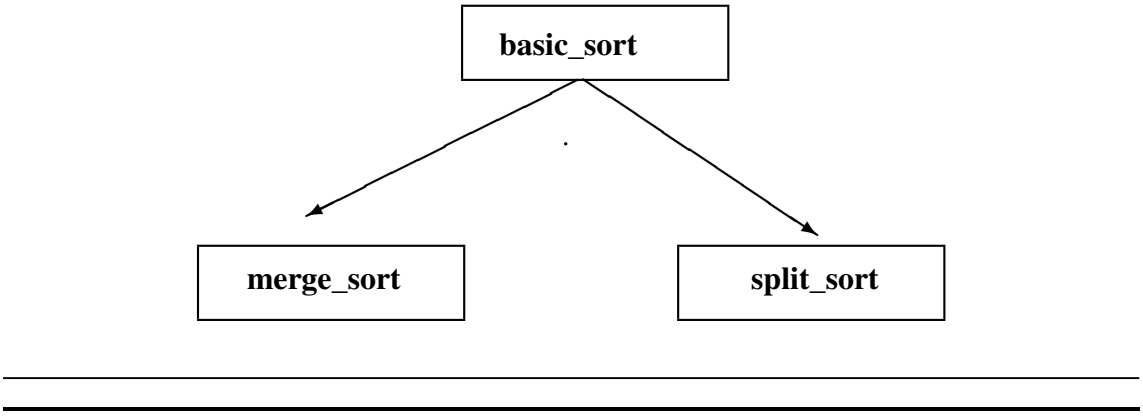

Figure 5.1: Sorting development

5.3.3 Duality of sorting models

Two models refine the basic model for the sorting problem; the tuple space frees the designer from implementation details and structure the computation process. In the figure 5.1, we summarize the refinement relationship between the three models developed in the previous subsections. Two families of sorting techniques can be redeveloped and we will develop the family of sorting techniques based on the split paradigm.

We do not develop, in this paper, sorting algorithms of the merge family and we restrict our illustration to the split family.

5.4 Introducing a pivot and an index

The quicksort splits arrays by choosing a pivot variable and it reorganizes both intervals such that any value of the first interval is smaller than any value of the second interval. The next refinement defines a pivot (*piv*) and a concrete index (*k*), which allows to split the current interval (*I*). Two index variables, namely (*binf*) and *bsup*), define the middle part of an interval. The middle part is not processed by the partitioning process. The partitioning algorithm is not used in our current process, since it can split the current interval in three parts. The control of *binf* and *bsup* is fundamental: the increasing of *binf* and the decreasing of *bsup*. The new invariant is enriched by statements on properties satisfied by the new variables, namely *piv*,*k*, *binf* and *bsup*. The variable *ToSplit* detects what is the phase of the partitioning process; it can contain three values: *No*, when no split phase is running; *Yes* if the partitioning process is progressing, *End* when the partitioning process for a given interval is completed.

The resulting invariant expresses intuitive properties over variables; the proof assistant generates proof obligations for validating the refinement and helps us to add details over variables that were missing. When developing abstract models, a proof assistant like Atelier B is crucial and it avoids errors in brain-aided proofs. The proof helps us to choose the correct index (*k*) to partition the resulting interval, when the splitting process stops ($ToSplit = End$). Explications are necessary to read and to understand the invariant.

The first part expresses typing information. I is the current interval, which satisfies properties resulting from the guard of *choice_interval* event.

$$
\begin{array}{l}
\mathit{ToSplit} \in \{\mathit{No}, \mathit{Yes}, \mathit{End}\} \wedge \\
I \subseteq 1..m \wedge \\
\mathit{piv} \in \mathbb{N} \wedge \\
\mathit{binf} \in 1..m \wedge \\
\mathit{bsup} \in 1..m \wedge \\
k \in \mathbb{N} \wedge \\
(\mathit{ToSplit} \neq \mathit{No} \Rightarrow \mathit{piv} \in \mathrm{RAN}(I \lhd g)) \wedge \\
(\mathit{ToSplit} \neq \mathit{No} \Rightarrow I \in \mathit{TS}) \wedge \\
(\mathit{ToSplit} \neq \mathit{No} \Rightarrow I - \{\mathrm{MAX}(I)\} \neq \emptyset) \wedge \\
(\mathit{ToSplit} = \mathit{Yes} \Rightarrow \mathit{binf} \in I) \wedge \\
(\mathit{ToSplit} = \mathit{Yes} \Rightarrow \mathit{bsup} \in I) \wedge
\end{array}
$$

The splitting of the current interval in two intervals is made possible by controling the two variables *binf* and *bsup*. *binf* may increase and *bsup* may decrease: *left_partition* can increase *binf* and *right_partition* can decrease *binf*. Both events are possibly occuring when $binf < bsup$ and are complementary with respect to guards. The *swap* event is enabled, when both *left_partition* and right_partition are no more enabled and when the two bounds are still satisfying the relationship $binf < bsup$. In this case, e must decide the new bound k which must split the interval in two non-empty intervals:
$(\mathit{ToSplit} = \mathit{End} \Rightarrow k \in I - \{\mathrm{MAX}(I)\})$.
If one chooses*binf*-1 or *bsup*, these values must be different to the initial value of the greater bound. So, if this greater bound does not change, the other bound must be less and the pivot is still in the first part.

$$
\begin{array}{l}
(\mathit{ToSplit} = \mathit{Yes} \wedge \mathit{binf} = \mathrm{MIN}(I) \Rightarrow \mathit{piv} \notin \mathrm{RAN}(\mathit{bsup}+1..\mathrm{MAX}(I) \lhd g)) \wedge \\
(\mathit{ToSplit} = \mathit{Yes} \wedge \mathit{bsup} = \mathrm{MAX}(I) \Rightarrow \mathit{binf} < \mathit{bsup}) \wedge \\
(\mathit{ToSplit} = \mathit{Yes} \wedge \mathit{bsup} = \mathrm{MAX}(I) \Rightarrow \mathit{piv} \notin \mathrm{RAN}(\mathrm{MIN}(I)..\mathit{binf}-1 \lhd g)) \wedge \\
(\mathit{ToSplit} = \mathit{Yes} \wedge \mathit{bsup} = \mathrm{MAX}(I) \Rightarrow \mathit{piv} \in \mathrm{RAN}(I - \{\mathrm{MAX}(I)\} \lhd g)) \\
(\mathit{ToSplit} = \mathit{Yes} \Rightarrow \forall z.(z \in \mathrm{MIN}(I)..\mathit{binf}-1 \Rightarrow g(z) \leq \mathit{piv})) \wedge \\
(\mathit{ToSplit} = \mathit{Yes} \Rightarrow \forall z.(z \in (\mathit{bsup}+1)..\mathrm{MAX}(I) \Rightarrow g(z) \geq \mathit{piv})) \wedge \\
(\mathit{ToSplit} = \mathit{Yes} \wedge \mathit{bsup} < \mathit{binf} \Rightarrow \mathit{binf} \leq \mathrm{MAX}(I)) \wedge \\
(\mathit{ToSplit} = \mathit{Yes} \wedge \mathit{bsup} \leq \mathit{binf} \Rightarrow (\mathit{binf} = \mathit{bsup} \vee \mathit{binf} = \mathit{bsup}+1)) \wedge \\
(\mathit{binf} = \mathit{bsup} \Rightarrow \mathit{bsup} < \mathrm{MAX}(I)) \wedge \\
\\
(\mathit{ToSplit} = \mathit{End} \Rightarrow k \in I - \{\mathrm{MAX}(I)\}) \wedge \\
(\mathit{ToSplit} = \mathit{End} \Rightarrow \forall z.(z \in \mathrm{MIN}(I)..k \Rightarrow g(z) \leq \mathit{piv})) \wedge \\
(\mathit{ToSplit} = \mathit{End} \Rightarrow \forall z.(z \in k+1..\mathrm{MAX}(I) \Rightarrow g(z) \geq \mathit{piv}))
\end{array}
$$

Safety properties can be proved from the invariant and are stated in the clause ASSERTIONS of the B machine. These properties are useful to validate the system itself.

$$
\begin{array}{l}
(\mathit{ToSplit} = \mathit{Yes} \Rightarrow I - \{\mathrm{MAX}(I)\} = \mathrm{MIN}(I)..\mathrm{MAX}(I)-1) \wedge \\
(\mathit{ToSplit} = \mathit{Yes} \Rightarrow \mathrm{MIN}(I)..\mathrm{MAX}(I) \subseteq I) \wedge \\
(\mathit{ToSplit} = \mathit{Yes} \Rightarrow \mathit{binf}..\mathit{bsup} \subseteq I)
\end{array}
$$

The invariant is proved to be satisfied by the refined events and we list the refined events; the first one is the initialisation event called *Init*. The tuple space contains only one interval, namely $1..m$ and the splitting process is not running at the initialisation state.

$$
\begin{array}{l}
\mathit{Init} = \textbf{begin} \\
\quad q := f \\
\quad \| \ \mathit{TS} := \{1..m\} \\
\quad \| \ \mathit{ToSplit} := \mathit{No} \\
\quad \| \ I := \emptyset \\
\quad \| \ \mathit{piv} :\in \mathbb{N} \\
\quad \| \ \mathit{binf} :\in 1..m \\
\quad \| \ \mathit{bsup} :\in 1..m \\
\quad \| \ k :\in 1..m \\
\textbf{end}
\end{array}
$$

The event *sorting* does not change; the guard of *split_progress* is very simple. When the partition process is finished ($ToSplit = End$), *k* is the index result for the partition (see event partition)

split_progress =
when
ToSplit = *End*
then
ToSplit := *No*
|| *TS* := *TS* $-\{I\} \cup \{\{y|y \in I \wedge y \leq k\}, \{y|y \in I \wedge y > k\}\}$
end;

We introduce five new events. The first one, namely *choice_interval*, chooses an interval (not a singleton) in the tuple space and initializes both index and the pivot. After the activation of this event, we can cut the current interval (*ToSplit* = *Yes*).

choice_interval = **when** *ToSplit* = *No* **then**
any *J*, *PIV* **where**
$J \in TS \wedge$
$PIV \in \text{RAN}((J - \{\text{MAX}(J)\}) \lhd g) \wedge$
$\text{MIN}(J) < \text{MAX}(J)$
then
$I := J \parallel$
$piv := PIV \parallel$
ToSplit := *Yes* ||
$binf := \text{MIN}(J) \parallel$
$bsup := \text{MAX}(J)$
end
end;

The three next events move the index to leave element less than pivot before *binf* and greater than pivot after *bsup*.

left_partition = **when**
ToSplit = *Yes* $\wedge$ *binf* $<$ *bsup* $\wedge$ *g*(*binf*) $<$ *piv*
then
binf := *binf*+1
end;

right_partition = **when**
ToSplit = *Yes* $\wedge$ *binf* $<$ *bsup* $\wedge$
g(*binf*) $\geq$ *piv* $\wedge$ *g*(*bsup*) $>$ *piv*
then
bsup := *bsup*-1
end;

swp = **when**
ToSplit = *Yes* $\wedge$ *binf* $<$ *bsup* $\wedge$
g(*binf*) $\geq$ *piv* $\wedge$ *g*(*bsup*) $\leq$ *piv*
then
binf, *bsup* := *binf*+1, *bsup*-1
|| $g := g \Leftarrow \{binf \mapsto g(bsup)\} \Leftarrow \{bsup \mapsto g(binf)\}$
end;

The last one stops the partitioning process and defines the index *k*, which makes progress possible (see event *split_progress*).

```
partition = when
              ToSplit = Yes ∧ binf ≥ bsup
            then
              ToSplit := End ||
              if binf = bsup then
                if g(binf) ≤ piv then
                  k := binf
                else
                  k := binf−1
                end
              else
                k := bsup
              end
            end
```

5.5 A set of bounds and a concrete pivot

The goal of the next refinement is to implement the tuple space by a set of initial bounds from every interval in the abstract tuple space. Initially, we have tried to introduce this implementation in the first refinement but it leads us to a unique proof obligation, whose proof was very long. Hence, we have found another abstraction, which produces more proof obligations than the initial choice but they were easier to prove. The implementation of the pivot is the middle of the chosen interval and now, the choice is deterministic. The relationship between pairs of bounds of the new tuple space (*TB*) and the tuple space (*TS*) is stated by a gluing invariant and the relationship is a one to one relation.

$$
\begin{array}{l}
TB \subseteq 1..m{+}1 \wedge \\
\forall(a,b).(a \in TB \wedge b \in TB \wedge \\
\quad a < b \wedge a{+}1..b{-}1 \cap TB = \emptyset \\
\quad \Rightarrow a..b{-}1 \in TS)
\end{array}
$$

We add two new variables, namely A and B, which are the bounds of the current abstract interval I. Variables satisfy the following gluing invariant: invariant:

$$
\begin{array}{l}
(ToSplit = Yes \Rightarrow A \in TB \wedge B \in TB \wedge \\
\quad A < B \wedge A{+}1..B{-}1 \cap TB = \emptyset \wedge \\
\quad A..B{-}1 = I) \wedge \\
(ToSplit = End \Rightarrow A \in TB \wedge B \in TB \wedge \\
\quad A < B \wedge A{+}1..B{-}1 \cap TB = \emptyset \wedge \\
\quad A..B{-}1 = I)
\end{array}
$$

Two new safety properties are derived from the current invariant:

$$
\begin{array}{l}
\forall I.(I \in TS \Rightarrow \text{MIN}(I) \in TB \wedge \text{MAX}(I){+}1 \in TB); \\
\\
\forall(a,b,c).(a \in TB \wedge c \in TB \wedge b \in TB \wedge a < b \wedge b < c \wedge \\
\quad TB \cap a{+}1..b{-}1 = \emptyset \wedge TB \cap b{+}1..c{-}1 = \emptyset \\
\quad \Rightarrow \\
\quad \forall(x,y).(x \in a..b{-}1 \wedge y \in b..c{-}1 \Rightarrow g(x) \leq g(y)))
\end{array}
$$

We refine only two events. The event *split_progress* adds the unique value $k{+}1$ in the concrete tuple space (*TB*).

```
split_progress = when
                   ToSplit = End
                 then
                   ToSplit := No
                 || TB := TB ∪ {k+1}
                 end;
```

The event *choice_interval* initializes the concrete bounds A and B of the abstract interval I. It chooses the pivot as the value $g((a+b-1)/2)$ at the middle of the chosen interval.

choice_interval = **when** *ToSplit* = *No* **then**
any a, b, p **where**
$a \in$ *TB* $\wedge$
$b \in$ *TB* $\wedge$
$a < b-1 \wedge$
$a+1..b-1 \cap$ *TB* $= \emptyset \wedge$
$p = g((a+b-1)/2)$
then
$A := a \parallel$
$B := b \parallel$
piv $:= p \parallel$
ToSplit := *Yes* $\parallel$
binf $:= a \parallel$
bsup $:= b-1$
end
end;

5.6 Implementation of the tuple space by a stack

The next step plans to use a stack for implementing the tuple space; it is clear that the current abstract model might be directly implemented in a coordination language as C LINDA for instance. However, we recall that the coordination paradigm is a methodological support for the development. In this refinement, we implement the tuple space by a stack. We use three new variables *TA*, *top*, *S*, which stands for the old variables *TB*. *S* (*Single*) contains all bounds interval which are singletons and which were on the top of the stack *TA*. All bounds in *TB* are single one ($\in S$) or in the codomain of *TA* and vice versa, according to our gluing invariant. Two consecutive bounds in *TB* are given by two consecutive index of the stack (array). The concrete tuple space *TA* is sorted. *top* is the dimension of *TA*. Notice that *top* is always between 1 and $m+1$. No stack overflow can occur.

$$
\begin{array}{l}
\mathit{top} \in 1..m+1 \wedge \\
\mathit{TA} \in 1..\mathit{top} \longrightarrow 1..m+1 \wedge \\
S \subseteq \mathit{TB} \wedge \\
\mathit{TB} = \text{RAN}(\mathit{TA}) \cup S \wedge \\
\forall (i,j).(i \in \text{DOM}(\mathit{TA}) \wedge \\
\qquad j \in \text{DOM}(\mathit{TA}) \wedge \\
\qquad i < j \\
\qquad \Rightarrow \\
\qquad\quad \mathit{TA}(i) < \mathit{TA}(j)) \wedge
\end{array}
$$

When S is empty, the greater bound in the codomain of *TA* is $m+1$ and, when S is not empty, it contains consecutive index from $m+1$ and the greater bound in the codomain of *TA* and the minimum of S are consecutive. Using this technical invariant, it is easier to prove the previous gluing invariant.

$$
\begin{array}{l}
(S = \emptyset \Rightarrow \text{MAX}(\text{RAN}(\mathit{TA})) = m+1) \wedge \\
(S \neq \emptyset \Rightarrow S = \text{MIN}(S)..m+1) \wedge \\
(S \neq \emptyset \Rightarrow \text{MAX}(\text{RAN}(\mathit{TA}))+1 = \text{MIN}(S)) \wedge
\end{array}
$$

The following properties are proved from the invariant.

$$
\begin{array}{l}
(\mathit{ToSplit} \neq \mathit{No} \Rightarrow (\mathit{top} \mapsto B) \in \mathit{TA}) \wedge \\
(\mathit{ToSplit} \neq \mathit{No} \Rightarrow (\mathit{top}-1 \mapsto A) \in \mathit{TA}) \wedge \\
(\mathit{ToSplit} \neq \mathit{No} \Rightarrow \mathit{top} > 1) \wedge \\
(\mathit{ToSplit} \neq \mathit{No} \Rightarrow \mathit{top} \leq m)
\end{array}
$$

$$
\begin{array}{l}
TA \in 1..top \rightarrowtail 1..m{+}1 \wedge \\
\mathrm{MAX}(\mathrm{RAN}(TA)) = TA(top) \wedge \\
\mathrm{RAN}(TA) \cap S = \emptyset \wedge \\
\\
\forall(h,n).(\ n \in 1..m{+}1 \wedge \\
\qquad h \in 1..n \rightarrowtail 1..n \wedge \\
\qquad \forall(x,y).(x \in 1..n \wedge \\
\qquad\qquad y \in 1..n \wedge \\
\qquad\qquad x < y \\
\qquad\qquad \Rightarrow \\
\qquad\qquad h(x) < h(y)) \\
\qquad \Rightarrow \\
\qquad\ h = \mathrm{ID}(1..n))
\end{array}
$$

The last one is very important in proving that there is no run stack overflow on our stack. It expresses that the unique increasing into function between $1..m{+}1$ and $1..m{+}1$ is the identity. We have proved it in another B machine with other preliminary lemmas like previous assertions. The initial event is written from the previous one.

$$
\begin{array}{l}
Init = \textbf{begin} \\
\qquad g := f \\
\quad \| \ TA := \{1 \mapsto 1, 2 \mapsto m{+}1\} \\
\quad \| \ S := \emptyset \\
\quad \| \ top := 2 \\
\quad \| \ ToSplit := No \\
\quad \| \ A, B := m{+}1,\ 1 \\
\quad \| \ piv :\in \mathbb{N} \\
\quad \| \ binf :\in 1..m \\
\quad \| \ bsup :\in 1..m \\
\quad \| \ k :\in 1..m \\
\textbf{end}
\end{array}
$$

Only three old events change. Now, the guard of *sorting* is $top = 1$: remember that the proof of the refinement assumes that in this case all intervals are singleton. The implementation is very close.

$$
\begin{array}{l}
sorting = \textbf{when } top = 1 \textbf{ then} \\
\qquad \mathrm{SKIP} \\
\textbf{end};
\end{array}
$$

$$
\begin{array}{l}
split\_progress = \\
\quad \textbf{when } ToSplit = End \textbf{ then} \\
\qquad ToSplit := No \\
\quad \| \ top := top + 1 \\
\quad \| \ TA := (TA \ \lhd\!\!\!\!- \{\ top \mapsto k{+}1\}) \ \lhd\!\!\!\!- \ \{top{+}1 \mapsto BB\ \} \\
\quad \textbf{end};
\end{array}
$$

The event which chooses the interval is now completely deterministic. The bounds of the chosen interval are on the top of the stack *TA*. Notice, that the chosen interval is not a singleton ($TA(top{-}1){+}1 \neq TA(top)$. Singleton on the top of the stack is removed by a new event as follows:

```
choice_interval =
        when
            top > 1 ∧
            (TA(top−1)+1 ≠ TA(top)) ∧
            ToSplit = No
        then
            ToSplit := Yes ||
            A, B, piv, binf, bsup ∈ ( A = TA( top−1) ∧
                                      B = TA(top) ∧
                                      piv = g((A+B−1)/2) ∧
                                      binf = A ∧
                                      bsup = B−1 )
        end;
```

New event, so-called *elim_single*, eliminates every singleton on the top of the stack.

```
elim_single = when
                  top > 1 ∧
                  TA(top−1)+1 = TA(top) ∧
                  ToSplit = No
              then
                   S := S ∪ {TA(top)}
                || top := top−1
                || TA := 1.. top−1 ◁ TA
              end;
```

All guards of the previous system are very simple to implement and all events are deterministic. We can easily derive from this system an iterative program using array and loops. The set of singleton S is not important in this implementation. If somebody wants to use it, one can store it in TA from the index m in a decreasing way. The iterative version of the algorithm is given in the figure 5.2.

5.7 Conclusion

The iterative algorithm is three times faster that the quicksort; it is obtained by combining the coordination paradigm and the event-driven paradigm. Every abstract model can be implemented by a coordination program but we use the coordination paradigm as a computation model and the refinement allows us to transit from the coordination model to the classical sequential model. Moreover, it provides us a way to develop a split algorithm without use of recursive aspect. The experience shows that coordination gives a simple way to think on the activity of events and it helps in explaining what is really happening, when, for instance, a paradigm is applied for sorting. We have not completely explored the promise land of coordination and we have not compared our works to refinements for coordination.

```
g := f;
TA[1] := 1;
TA[2] := m+1;
top := 2;
ToSplit = No
while top ≠ 1 do
    while top > 1 ∧ TA[top−1]+1 = TA[top] do
        top := top−1
    end;
    if top > 1 then
      A := TA[top−1];
      B := TA[top];
      binf := A;
      bsup := B−1;
      piv := g[(binf+bsup) div 2];
      ToSplit = Yes
      while (binf < bsup) do
          while binf < bsup ∧ g[binf] < piv do
              binf := binf+1
          end;
          while binf < bsup ∧ g[bsup] > piv do
              bsup := bsup−1
          end;
          if binf < bsup then
              temp := g[binf];
              g[binf] := g[bsup];
              g[bsup] := temp;
              binf := binf+1;
              bsup := bsup−1
          end
      end;
      if binf = bsup then
        if g[binf] ≤ piv then
          k := binf
        else
            k := binf−1
        end
       else
          k := bsup
       end;
       ToSplit = End
       TA[top] := k+1;
       top := top+1;
       TA[top] := B
       ToSplit = No
    end
end
```

Figure 5.2: A correct iterative program

Chapter 6

Spanning trees algorithms

6.1 Introduction

Graphs algorithms and graph-theoretical problems provide a challenging battle field for the incremental development of proved models. The B event-based approach implements the incremental and proved development of abstract models which are translated into algorithms; we focus our methodology on the minimum spanning tree problem and on Prim's algorithm. The correctness of the resulting solution is based on properties over trees and we show how the greedy strategy is efficient in this case. We compare properties proven mechanically to the properties found in a classical algorithms textbook. This section analyses the proof-based development of Minimal Spanning Tree algorithms and Prim's algorithm in particular [88] is produced in fine.

6.2 The Minimum Spanning Tree Problem

The Minimum Spanning Tree Problem, Minimal Spanning Tree problem for short, is the problem of finding a minimum spanning tree with respect to a connected graph. The literature contains several algorithmic solutions like Prim's algorithm [88] or Kruskal's algorithm [70]. Both algorithms implement the greedy method. Typically, we assume that a cost function is related to every edge and the problem is to infer a globally minimum spanning tree, which covers the initial graph. The cost function returns integer values. The Minimal Spanning Tree problem is strongly related to practical problems like the optimisation of circuitry and the greedy strategy advocates making the choice that is the best one at the moment; It does not always guarantee the optimality but certain greedy strategies yield a Minimal Spanning Tree.
Prim's algorithm is easy to explain but it underlies mathematical properties related to the graph theory and especially the general theory of trees. We consider two kinds of solutions; a first one is called *generic algorithm* because it does not use a cost function. This first *generic* solution allows us to develop a second solution: the Minimal Spanning Tree one.
Let us summarize how Prim's algorithm works. The state of the algorithm while executing contains two sets of nodes of the current graphs. A first set of nodes, equipped with a restriction of the relation over the global set of nodes, defines the current spanning tree starting from a special node called the root of the spanning tree. A second set of nodes is the complement of the first set. The acyclicity of the spanning tree must be preserved, while adding a new edge in the current spanning tree and the basic computation step consists of taking an edge between a node in the current spanning tree and a node which is in the other set. The choice leads to maintaining the acyclicity of the current spanning tree with the new node, since both sets of nodes are disjoint. The process is repeated as long as the set of remaining and unchosen nodes is empty. The final computed tree is a spanning tree computed by the generic algorithm. Now, if one adds the cost function, one gets Prim's algorithm by modifying the choice of the new node and edge to add to the current spanning tree. In fact, the minimum edge is chosen and the final spanning tree is then the minimum spanning tree. However, the addition of the cost function is a refinement of the generic solution.
The generic Minimal Spanning Tree algorithm without cost function is sketched as follows:

- Precondition: *A undirected connected graph, g, over a set of nodes ND and a node r*

- Initial Step tr_nodes *(the current set of nodes) contains only* r *and is included into* ND *and* tr *(the current set of edges) is empty*
- Computation Step *If* $ND{-}tr\_nodes$ *is not empty, then choose a node* x *in* tr_nodes *and a node* y *in* $ND{-}tr\_nodes$ *such that the link* (x, y) *is in* g *with the minimum cost and add it to* tr*; then add* y *to* tr_nodes *and* (x, y) *to* tr
- Termination Step *If* $ND{-}tr\_nodes$ *is empty (*$ND = tr\_nodes$*), then* tr *is a minimum spanning tree on* ND
- Postcondition (ND, tr) *is a minimum spanning tree*

The termination of the algorithm is ensured by decreasing the set $ND{-}tr\_nodes$. The genericity of the solution leads us to the refinement by introducing the cost function in the computation step. We have a clear simple abstract view of the problem and of the solution. We can, in fact, state the problem in the B event-based framework. It remains to prove the optimality of the resulting spanning tree and that will be derived using tools and models. Before starting the modeling, we recall the B-event-based modeling technique.

6.3 Development of a spanning tree algorithm

6.3.1 Formal specification of the spanning tree problem

First we define elements of the current graph namely g over the set of nodes namely ND. The graph is assumed to be undirected, which is modeled by the symmetry of the relation of the graph. Node r is the root of the resulting tree and we obtain the following B definitions:

$$\begin{array}{l} g \subseteq ND \times ND \ \wedge \\ g = g^{-1} \ \wedge \\ r \in ND \end{array}$$

The termination of the algorithm is clearly related to properties of the current graph; the existence of the spanning tree is based on the connectivity of the graph. The modelling of a tree uses the acyclicity of the graph. A tree is defined by a root r, a node: $r \in ND$, and a parent function t (each node has an unique parent node, but the root): $t \in ND{-}\{r\} \longrightarrow ND$. A tree is an acyclic graph. A cycle c in a finite graph t built on a set ND, is a subset of ND whose elements are members of the inverse image of c under t, formally $c \subseteq t^{-1}[c]$. To fulfill the requirement of acyclicity, the only set c that enjoys this property is necessarily the empty set. We formalize it by the left predicate that follows, which can be proved to be *equivalent* to the one on the right, which can be used as an induction rule:

$$\begin{array}{l} \forall c \cdot (\\ \quad c \subseteq ND \ \wedge \\ \quad c \subseteq t^{-1}[c] \\ \quad \Rightarrow \\ \quad c = \emptyset \,) \end{array} \quad \Leftrightarrow \quad \begin{array}{l} \forall q \cdot (\\ \quad q \subseteq ND \ \wedge \\ \quad r \in q \ \wedge \\ \quad t^{-1}[q] \subseteq q \\ \quad \Rightarrow \\ \quad ND = q \,) \end{array}$$

We prove the equivalence using Atelier B. We can now define a spanning tree (rooted by r and with the parent function t) of a graph g as one whose parent function is included in g, formally:

$$\mathsf{spanning}\,(t, g) \ \widehat{=} \left(\begin{array}{l} t \in ND{-}\{r\} \longrightarrow ND \quad \wedge \\ \forall q \cdot (q \subseteq ND \ \wedge \ r \in q \ \wedge \ t^{-1}[q] \subseteq q \ \Rightarrow \ ND = q \,) \ \wedge \\ t \subseteq g \end{array} \right)$$

Now we can define the set $\mathsf{tree}\,(g)$ of all spanning trees (with root r) of the graph g, formally:

$$\mathsf{tree}\,(g) \;=\; \{t|\mathsf{spanning}\,(t,g)\}$$

We define the property of *being a connected graph* by $\mathsf{connected}(g)$:

$$\mathsf{connected}\,(g) \;\widehat{=}\; \left(\begin{array}{l} g \;\in\; ND \;\leftrightarrow\; ND \quad \wedge \\ \forall S \cdot (\, S \subseteq ND \;\wedge\; r \in S \;\wedge\; g\,[S] \subseteq S \;\Rightarrow\; ND = S\,) \end{array}\right)$$

The graph g and the node r are two global constants of our problem and must satisfy properties stated above. Moreover, we assert that there is at least one solution to our problem. The optimality of the solution will be analyzed later, while introducing the cost function. Now, we build the first model which computes the solution in one shot. The event span corresponds to producing a spanning tree among the non-empty set of possible spanning trees for g. The variable st contains the resulting spanning tree.

span $\widehat{=}$
begin
$st :\in \mathsf{tree}(g)$
end

The invariant is very simple and only a type invariant.

$$st \in ND \leftrightarrow ND$$

The initialization establishes this invariant.

The current model is in fact the specification of the simple spanning tree problem; we have not yet mentioned the cost function. The next step is to refine the current model into a simple spanning tree algorithm.

6.3.2 Development of a simple spanning tree algorithm

The second model introduces a new event which gradually computes the spanning tree by constructing the spanning tree in a progressive way. The new event adds a new edge to the current tree tr which partly spans g. The chosen edge is such that the first component of the pair is in tr_nodes and the second one is in $remaining\_nodes$. These two new variables partition the set of nodes and we obtain the following new properties to add to the invariant of the current model.

$$\begin{array}{l} tr\_nodes \;\subseteq\; ND \quad \wedge \\ remaining\_nodes \;\subseteq\; ND \quad \wedge \\ tr\_nodes \cup remaining\_nodes \;=\; ND \quad \wedge \\ tr\_nodes \cap remaining\_nodes \;=\; \emptyset \end{array}$$

A new event, progress, simulates the computation step of the current solution by choosing a pair maintaining the updated invariant.

progress $\widehat{=}$
select
$remaining\_nodes \neq \emptyset$
then
any x, y **where**
$x, y \in g \;\wedge\; x, y \in tr\_nodes \times remaining\_nodes$
then
$tr := tr \cup \{y \mapsto x\} \;||$
$tr\_nodes := tr\_nodes \cup \{y\} \;||$
$remaining\_nodes := remaining\_nodes - \{y\}$
end
end

The event span is simply refined by modifying the guard of the previous instance of the event in the abstract model. The event is triggered when the set of remaining nodes is empty: the variable st contains a spanning tree for the graph g.

```
span ≙
  select
    remaining_nodes = ∅
  then
    st := tr
  end
```

The invariant of the new model states the properties of the two new variables and relates them to previous ones.

$$
\begin{array}{l}
tr\_nodes \subseteq ND \quad \wedge \\
remaining\_nodes \subseteq ND \quad \wedge \\
tr\_nodes \cup remaining\_nodes = ND \quad \wedge \\
tr\_nodes \cap remaining\_nodes = \emptyset \quad \wedge \\
tr \in tr\_nodes - \{r\} \longrightarrow tr\_nodes \quad \wedge \\
\forall q \cdot (q \subseteq tr\_nodes \ \wedge \ r \in q \ \wedge \ tr^{-1}[q] \subseteq q \ \Rightarrow \ tr\_nodes = q)
\end{array}
$$

The following initialization establishes the invariant:

$$
\begin{array}{l}
tr := \emptyset \ || \\
tr\_nodes := \{r\} \ || \\
remaining\_nodes := ND - \{r\}
\end{array}
$$

The expression of the absence of deadlock is simply stated as follows:

$$
\begin{array}{l}
remaining\_nodes = \emptyset \ \vee \\
remaining\_nodes \neq \emptyset \ \wedge \ \exists (x,y) \cdot \left(\begin{array}{l} x,y \in g \ \wedge \\ x,y \in tr\_nodes \times remaining\_nodes \end{array} \right)
\end{array}
$$

We have obtained a simple iterative solution for the simple Minimal Spanning Tree problem; the solution follows the sketch of the algorithm given in the subsection describing the so called generic algorithm in the book of Cormen et al. [58]. We can derive the following algorithm from the current model:

```
algorithm generic_MST
  tr := ∅;
  tr_nodes = {r};
  while remaining_nodes ≠ ∅ do
    let x, y where
      x, y ∈ g  ∧  x, y ∈ tr_nodes×remaining_nodes
    then
      tr := tr ∪ {y ↦ x};
      tr_nodes := tr_nodes∪ {y};
      remaining_nodes := remaining_nodes−{y}
    end
  end_while
  st := tr
```

The next step refines the current model into a model where the cost function is effectively used.

6.3.3 A proof view of the spanning tree algorithm

The previous model computes a spanning tree, when the graph is connected. This algorithm looks like a proof of existence of a spanning tree; the following lemma allows us to prove that the set of spanning trees is not empty and hence a minimum spanning tree exists:

Propriété 6.1 (Existence of a spanning tree)
$\mathsf{connected}\,(g) \Rightarrow \mathsf{tree}\,(g) \;\neq\; \emptyset$

However, the previous lemma requires to construct a tree from the hypothesis related to the connectivity of the graph. Hence, we must prove a first inductive theorem on finite sets, which will include the existence of a tree. We suppose that the set ND is finite and there exists a function from ND to $1..n$, where n is the cardinality of ND.

Propriété 6.2 (An inductive theorem on finite sets)

$$\begin{array}{l}\forall P \cdot (\\ \quad P \subseteq \mathbb{P}(ND) \;\wedge\\ \quad \emptyset \in P \;\wedge\\ \quad \forall A \cdot (\, A \in P \;\wedge\; A \neq ND \;\Rightarrow\; \exists a \cdot (\, a \in ND{-}A \;\wedge\; A{\cup}\{a\} \in P))\\ \Rightarrow\\ \quad ND \in P)\end{array}$$

We can use the previous lemma with the following set:

$$\{A | A \subseteq ND \;\wedge\; \exists f \cdot \left(\begin{array}{l} f \in A{-}\{r\} \;\longrightarrow\; A \;\wedge\\ f \subseteq g \;\wedge\\ \forall S \cdot \left(\begin{array}{l} S \subseteq ND \;\wedge\; r \in S \;\wedge\; f^{-1}[S] \subseteq S\\ \Rightarrow\\ A \subseteq S \end{array}\right) \end{array}\right)\}$$

to prove that the set of spanning trees of g is not empty.

6.4 Development of Prim's algorithm

The cost function is defined on the set of edges and is extended over the global set of possible pairs of nodes.

$$\begin{array}{l} cost : g \longrightarrow \mathbb{Z} \;\wedge\\ \forall (x,y) \cdot (x,y \in g \;\Rightarrow\; cost(x \mapsto y) = cost(y \mapsto x)) \;\wedge\\ Cost : \mathbb{P}(g) \longrightarrow \mathbb{Z} \;\wedge\\ Cost(\{\}) = 0 \;\wedge\\ \forall (s,x,y) \cdot \left(\begin{array}{l} s \in \mathbb{P}(g) \;\wedge\; x,y \in g{-}s\\ \Rightarrow\\ Cost(s{\cup}\{x \mapsto y\}) = Cost(s){+}cost(x \mapsto y) \end{array}\right) \end{array}$$

We have proved that $\mathsf{tree}(g)$ is not empty, since the graph g is connected; the $mst\_set(g)$ containing every minimum spanning tree of the graph g is defined as follows:

$$\begin{array}{l} mst\_set(g) =\\ \qquad \{mst | mst \in \mathsf{tree}(g) \;\wedge\; \forall tr \cdot (tr \in \mathsf{tree}(g) \Rightarrow Cost(mst) \leq Cost(tr))\} \end{array}$$

The set $mst\_set(g)$ is clearly not empty. The first «one shot» model is refined into the new model which contains only one event span. We strengthen the definition of the choice of the resulting tree by strengthening the condition over the set and by choosing a candidate in the set of possible Minimal Spanning Tree trees.

```
span ≙
  begin
    st :∈ mst_set(g)
  end
```

The second model gradually computes the spanning tree by adding a new edge to the current «under construction» tree tr spanning a part of g. The tree tr is defined over the set of already treated nodes, called tr_nodes. The event progress is modified to handle the minimality criterion: the guard is modified to integrate the choice of the minimum edge among the remaining possible ones.

```
progress ≙
  select
    remaining_nodes ≠ ∅
  then
    any x, y where
      x, y ∈ g  ∧  x, y ∈ tr_nodes×remaining_nodes  ∧
      ∀(a, b) · (a ∈ tr_nodes  ∧
                 b ∈ remaining_nodes  ∧
                 a, b ∈ g
                ⇒
                 cost(y ↦ x) ≤ cost(b ↦ a))
    then
      tr := tr ∪ {y ↦ x} ||
      tr_nodes := tr_nodes∪ {y} ||
      remaining_nodes := remaining_nodes−{y}
    end
  end
```

The event span remains unchanged:

```
span ≙
  select
    remaining_nodes = ∅
  then
    st := tr
  end
```

The invariant includes the invariant of the refined model of the generic refinement and we add that the current spanning tree tr is a part of a minimum spanning tree of the graph g:

$$\exists T \cdot (T \in mst\_set(g) \ \wedge \ tr \ \subseteq T)$$

The invariant implies that after completion, when the event span occurs, the current spanning tree tr is finally a minimal one. Since tree(g) is not empty, then $mst\_set(g)$ is not empty and a tree can be chosen in this non-empty set to prove that a Minimal Spanning Tree exists (this Minimal Spanning Tree contains $\emptyset$). So the invariant holds for the initialization, using the lemma 1. The difficult task is to prove that the event progress maintains the invariant. We can take the minimum spanning tree given by the invariant, if $y \mapsto x$ is in this tree. Or else we must provide another minimum tree which includes the current one and the new edge $y \mapsto x$.

In fact, textbooks provide algorithms implementing the greedy strategy and we refer our explanations to the book of Cormen et al. [58]. The authors prove a theorem page 501 numbered 24.1 to assert that the choice of the two edges is done following a given requirement, namely a safe edge (a safe edge is a edge allowing the progress of the algorithm). We recall the theorem:

Théorème 3 *(24.1, p 501from [58])*
Let g be a connected, undirected graph on ND (set of nodes) with a real-valued weight function $cost$ defined on g (edges). Let tr be a subset of g that is included in some minimum spanning tree for g, let $(tr\_nodes, ND{-}tr\_nodes)$ be any cut of g that respects tr_nodes, and let (x, y) be a light edge crossing $(tr\_nodes, ND{-}tr\_nodes)$. Then edge (x, y) is safe for tr_nodes.

Let us explain notions of cut, crosses and light edge. A cut
$(tr\_nodes, ND{-}tr\_nodes))$ of an undirected graph g is a partition of ND. An edge (x, y) crosses the cut $(tr\_nodes, ND{-}tr\_nodes)$ if one of its endpoints is in tr_nodes and the other is in $ND{-}tr\_nodes$. An edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the cut. A light edge is not unique.

Proof: *Let T be a minimum spanning tree that includes tr, and assume that T does not contain the light edge (x, y), since if it does, we are done. We shall construct another minimum spanning tree T' that includes $tr \cup \{(x, y)\}$ by using a cut-and-paste technique, thereby showing that (x, y) is a safe edge for tr. The edge (x, y) forms a cycle with the edges on the path p from x to y in T. Since x and y are on opposite sides of the cut $(tr\_nodes, ND{-}tr\_nodes)$, there is at least one edge in T on the path p that also crosses the cut. Let (a, b) be any such edge. The edge (a, b) is not in tr, because the cut respects tr. Since (a,b) is on the unique path from x to y in T, removing (a, b) breaks T into two components. Adding (x, y) reconnects them to form a new spanning tree $T' = T{-}\{(a, b)\} \cup \{(x, y)\}$. We next show that T' is a minimum spanning tree. Since (x, y) is a light edge crossing $(tr\_nodes, ND{-}tr\_nodes)$ and (a, b) also crosses this cut, $cost(x, y) \leq cost(a, b)$. Therefore,*

$$\begin{array}{rcl} Cost(T') & = & Cost(T){-}cost(a,b){+}cost(x,y) \\ & \leq & Cost(T) \end{array}$$

But T is a minimum spanning tree, so that $Cost(T) \leq Cost(T')$; thus, T' must be a minimum spanning tree also. It remains to show that (x, y) is actually a safe edge for tr. We have $tr \subseteq T'$, since $tr \subseteq T$ and $(a, b) \notin tr$; thus, $tr{\cup}\{(x, y)\} \subseteq T'$. Consequently, since T' is a minimum spanning tree, (x, y) is safe for tr. □

We have to prove the property above that has been in fact adapted into the B proof engine. However, it is not a simple exercise of translation but a complete formulation of graph-theoretical aspects; moreover, the proof has been completely mechanized, as we will show in the next subsection. Let us compare the theorem and our formulation. The pair $(tr\_nodes, ND{-}tr\_nodes)$ is a cut in the left part of the implication; the restriction of the tree f to the set of nodes tr_nodes is a tree rooted by r; (x, y) crosses the cut. Those assumptions imply that there exists a spanning tree sp rooted by r that is minimum on tr_nodes and such that there exists a light cut (a, b) preserving the minimality property.

We must give a formal description of this theorem. We introduce a predicate $\mathsf{atree}(root, nodes, tree)$ stating that a structure $tree$ is a tree on the set $nodes$ and whose root is $root$:

$$\mathsf{atree}(root, nodes, tree) \;\widehat{=}\; \left(\begin{array}{l} root \in nodes \quad \wedge \\ tree \in nodes{-}\{root\} \longrightarrow nodes \quad \wedge \\ \forall q \cdot (q \subseteq nodes \;\wedge\; root \in q \;\wedge\; tree^{-1}[q] \subseteq q \;\Rightarrow\; nodes = q) \end{array} \right)$$

Hence, we must add the following property which is proved separately.

$$
\begin{array}{l}
\forall(T, tr\_nodes, x, y) \cdot (\\
\quad tr\_nodes \subseteq ND \ \wedge \\
\quad y \in ND \ \wedge \\
\quad \mathsf{atree}(r, ND, T) \\
\quad r \in tr\_nodes \ \wedge \\
\quad x \in tr\_nodes \ \wedge \\
\quad (y \notin tr\_nodes) \ \wedge \\
\quad \mathsf{atree}(r, tr\_nodes, (tr\_nodes - \{r\} \lhd T \rhd tr\_nodes)) \ \wedge \\
\quad \forall S \cdot (S \subseteq ND \ \wedge \ y \in S \ \wedge \ T[S] \subseteq S \Rightarrow S \cap tr\_nodes \neq \emptyset) \\
\Rightarrow \\
\quad \exists(a, b, T') \cdot (\\
\qquad a, b \in T \ \wedge \ a \notin tr\_nodes \ \wedge \ b \in tr\_nodes \ \wedge \\
\qquad \mathsf{atree}(r, ND, T') \ \wedge \\
\qquad T' \subseteq (T \cup T^{-1} - \{b \mapsto a, a \mapsto b\}) \cup \{y \mapsto x\} \ \wedge \\
\qquad Cost(T') = Cost(T) - cost(b \mapsto a) + cost(y \mapsto x) \ \wedge \\
\qquad y \mapsto x \in T' \ \wedge \\
\qquad (tr\_nodes - \{r\} \lhd T \rhd tr\_nodes) \subseteq T'))
\end{array}
$$

The property is the key result for ensuring the optimality of the greedy strategy in this process. In the next subsection, we detail the proof of our theorem.

6.5 On the theory of trees

As we have mentioned previously, trees play a central role in the justification of the algorithm; the optimality of the greedy strategy is mainly based on the proof of the theorem used by Cormen et al. [58]. We should now detail the theory of trees and intermediate lemmas required for deriving the theorem. Both the development of the tree identification protocol IEEE 1394 [12] and the development of recursive functions [44] require proofs related to the closure of relations; we apply the same technique for the closure of a function defining a tree.
Let (T, r) be a tree defined by a tree function T and a root r; they satisfy the following axioms $\mathsf{atree}(r, ND, T)$. The closure cl of T^{-1} is the smallest relation containing $\mathsf{id}(ND)$ and stable by application of T^{-1}, that is:

$$
\begin{array}{l}
cl \in ND \leftrightarrow ND \ \wedge \\
\mathsf{id}(ND) \subseteq cl \ \wedge \\
(cl; T^{-1}) \subseteq cl \ \wedge \\
\forall r \cdot (\\
\quad r \in ND \leftrightarrow ND \ \wedge \\
\quad \mathsf{id}(ND) \subseteq r \ \wedge \\
\quad (r; T^{-1}) \subseteq r \ \wedge \\
\quad \Rightarrow \\
\quad cl \subseteq r \\
)
\end{array}
$$

Useful properties on the closure can be derived from those definitions; for instance, the closure is a fixpoint; the root r is connected to every node of the connected component; the closure is transitive, etc. We summarize those properties using our notations:

$$
\begin{array}{l}
cl = \mathsf{id}(ND) \cup (cl; T^{-1}); \\
r \times ND \subseteq cl; \\
(T^{-1}; cl) \subseteq cl; \\
(cl; cl) \subseteq cl; \\
T \cap cl = \emptyset; \\
cl \cap cl^{-1} \subseteq \mathsf{id}(ND);
\end{array}
$$

Figure **??** contains a tree with the edge $b \mapsto a$ and without the edge $y \mapsto x$. The construction of a new tree which contains the edge $y \mapsto x$ but not the edge $b \mapsto a$ is done according to the following points (see the result in Figure **??**):

1. remove the edge $b \mapsto a$
2. reverse all edges between y to b (dashed arrows)
3. add the edge $y \mapsto x$

The resulting object seems to be a tree rooted by r.

Propriété 6.3 (Concatenation of two separate trees)
Let $T_1, r_1, N_1, T_2, r_2, N_2, x$ be such that: $\begin{cases} \mathsf{atree}(r_1, N_1, T_1) \\ \mathsf{atree}(r_2, N_2, T_2 \\ N_1 \cap N_2 = \emptyset \\ N_1 \cup N_2 = ND \\ x \in N_1 \end{cases}$
Then $\mathsf{atree}(r_1, ND, T_1 \cup T_2 \cup \{r_2 \mapsto x\})$.

Proof Sketch: The proof is made up of several steps. A first step proves that the concatenation is a total function over the set $N_1 \cup N_2$. A second one leads to a more technical task and we should prove the inductive property over trees using a splitting of the inductive variable S ($S \cap N_1$ and $S \cap N_2$). □

Propriété 6.4 (Subtree property)
Let (T, r) be a tree on ND ($\mathsf{atree}(r, ND, T)$) and b a node in ND.
Then $\mathsf{atree}(b, cl[\{b\}], (cl[\{b\}] - \{b\} \lhd T))$

Proof Sketch: The main difficulty is related to the inductive part. We must prove that, if $S \subseteq cl[\{b\}]$, $b \in S$ and $(cl[\{b\}] - \{b\} \lhd T)^{-1}[S] \subseteq S$, then $cl[\{b\}] \subseteq S$. We use the inductive property on T with the set $S \cup ND - cl[\{b\}]$. □

Propriété 6.5 (Complement of a sub-tree)
Let (T, r) be a tree on ND and b a node in ND.
Then $\mathsf{atree}(r, ND - cl[\{b\}], (cl[\{b\}] \lhd\!\!\!\!- T))$.

Proof Sketch: We should prove that, if $S \subseteq ND - cl[\{b\}]$, $b \in S$ and $(cl[\{b\}] \lhd\!\!\!\!- T)^{-1}[S] \subseteq S$, then $ND - cl[\{b\}] \subseteq S$. A hint is to use the inductive property on T with the set $S \cup cl[\{b\}]$. □
Now, we must characterize the sub-tree, where we have reversed the edge between y to the root b. Let $subtree(T, b)$ be the subtree of T with b as root (it's $cl[\{b\}] - \{b\} \lhd T$). This following function seems to be a good choice:

$$(cl^{-1}[\{y\}] \lhd\!\!\!\!- subtree(T, b)) \cup (cl^{-1}[\{y\}] \lhd subtree(T, b))^{-1}$$

$(cl^{-1}[\{y\}] \lhd subtree(T, b))^{-1}$ is exactly all reverse edges. $cl^{-1}[\{y\}]$ is the set of all parents of y.

Propriété 6.6 (Reverse from y to b produces a tree)
Let b, y such that: $\begin{cases} b \in ND \\ y \in cl[\{b\}] \end{cases}$
Then $\mathsf{atree}(y, cl[\{b\}], (cl^{-1}[\{y\}] \lhd\!\!\!\!- subtree(T, b)) \cup (cl^{-1}[\{y\}] \lhd subtree(T, b))^{-1})$

Proof Sketch: In this case we must use an induction on the tree $cl[\{b\}]$ and sometimes use an second induction with the inductive property in hypothesis.□

Propriété 6.7 (Existence of a spanning tree)

Let a, b, x, y such that $\begin{cases} b, a \in T \\ y \in cl[\{b\}] \\ x : ND{-}cl[\{b\}] \end{cases}$

Then there exists a tree T' such that:

$$\begin{cases} T' \subseteq (T \cup T^{-1}{-}\{a \mapsto b, b \mapsto a\}) \cup \{y \mapsto x\} \\ \mathsf{atree}(r, ND, T') \\ Cost(T') = Cost(T){-}cost(b \mapsto a){+}cost(y \mapsto x) \\ y \mapsto x \in T' \\ cl[\{b\}] \lhd T \subseteq T' \end{cases}$$

Proof Sketch: T' is obtained by concatenation of . the two trees identified in the two previous lemmas. Both trees are linked by the edge $y \mapsto x$. □

Finally, we have to prove the existence of an edge $b \mapsto a$ which is safe in the sense of the greedy strategy.

Propriété 6.8 (Existence of $b \mapsto a$)

Let tr_nodes, y such that: $\begin{cases} tr\_nodes \subseteq ND \\ y \in ND{-}tr\_nodes \\ r \in tr\_nodes \\ \forall S \cdot \left(\begin{array}{l} S \subseteq ND \ \wedge \ y \in S \ \wedge \ T[S] \subseteq S \\ \Rightarrow \\ S{\cap}tr\_nodes \neq \emptyset \end{array} \right) \end{cases}$

Then there exists a and b such that: $\begin{cases} a \in tr\_nodes \\ b \mapsto a \in T \\ b \notin tr\_nodes \\ b \in cl^{-1}[\{y\}] \end{cases}$.

The property of the existence of a minimum spanning tree can now be derived using lemmas and the proof of the property is then completely mechanized. The development of Prim's algorithm leads us to state and to prove properties over trees. The inductive definition of trees helps in deriving intermediate lemmas asserting that the growing tree converges to the Minimal Spanning Tree, according to the greedy strategy. The resulting algorithm is completely proved and we can partially reuse current developed models to obtain Dijkstra's algorithm or Kruskal's one. The greedy strategy is not always efficient and the optimality of the resulting algorithm is proved by the theorem 24.1 [58]. The greedy method is based on optimisation criteria and we have developed a collection of models [49] which can be used to be instantiated, when the greedy strategy is applicable and when some optimisation criterion is verified.

Chapter 7

Design of distributed algorithms by refinement

Sommaire

7.1 Design of distributed algorithms by refinement

Developing distributed algorithms can be made simpler and safer by the use of refinement techniques. Refinement allows one to gradually develop a distributed algorithm step by step, and to tackle complex problems like the PCI Transaction Ordering Problem [47] or the IEEE 1394 [19]. The B event-based method [14] provides a framework integrating refinement for deriving models solving distributed problems. The systems under consideration for our technique are general software systems, control systems, protocols, sequential and distributed algorithms, operating systems and circuits; these are generally very complex and have parts interacting with an environment. A discrete abstraction of such systems constitutes an adequate framework: such an abstraction is called a *discrete model*. A discrete model is more generally known as a *discrete transition system* and provides a view of the current system; the development of a model in B follows an incremental process validated by refinement. A system is modeled by a sequence of models related by the refinement and managed in a project. We limit the scope of our work to distributed algorithms modeled under the *local computation rule* [51] in graphs and we specialize the proof obligations with respect to the target of the development which is a distributed algorithm fitting safety and liveness requirements.
The goal of the IEEE 1394 protocol is to elect in a *finite time* a specific node, called the *leader*, in a network made of various nodes linked by some communication channels. Once the leader is elected, each non-leader node in the network should have a well defined way to communicate with it. This election of the leader has to be done in a distributed and non-deterministic way. The current development partially replays the IEEE 1394 protocol development: the resulting algorithm is not the IEEE 1394 protocol. In fact, we are presenting the development of a distributed leader election and we partially reuse the models of the IEEE 1394 protocol development: the first, second and third models are reused from our paper [19] and the contention is solved by assigning a static priority to each site. The resulting algorithm is derived from the last B model. The first development is the IEEE 1394 tree identify protocol and the second one is discovered from the first development.

7.2 The IEEE 1394 tree identify protocol

7.2.1 Introduction

Overview. Distributed systems are inherently complex to understand, to design and to verify. In order to master this complexity, people have developed various approach such as model-checking and theorem proving. In this paper, we illustrate the latter by applying it to the IEEE 1394 protocol [12].

Proof-based Development. Proof-based development methods integrate formal proof techniques in the development of software systems. The main idea is to start with a very abstract model of the system under development. We then gradually add details to this first model by building a sequence of more concrete ones. The relationship between two successive models in this sequence is that of *refinement* [25, 21, 52]. It is controlled by means of a number of, so-called, *proofs obligations*, which guarantee the correctness of the development. Such proof obligations are proved by automatic (and interactive) proof procedures supported by a proof engine. The essence of the refinement relationship is that it preserves already proved *system properties* including safety properties and termination properties. The invariant of an abstract model plays a central rôle for deriving safety properties and our methodology focuses on the incremental discovery of the invariant; the goal is to obtain a formal statement of properties through the final invariant of the last refined abstract model. When developing formal models for the IEEE 1394 protocol, we use the environment Atelier B [55] for generating and proving proof obligations.

Understanding Distributed Systems. As already mentioned, a distributed system is *complex*. In this paper, the IEEE 1394 protocol is used to illustrate a method for understanding how a typical distributed system is working. Understanding a distributed system means that we are able to explain why it is working safely and how it meets its requirements. In the case of the IEEE 1394 protocol, the same piece of code is duplicated at each node of an acyclic and connected network. And the process that should be performed by these codes, each working concurrently but with a limited knowledge, is the *leader election*: in other words, at the end of the process a node should be given a special status, that of the leader, and other nodes should have a means

to eventually communicate with it. It is, in fact, not clear at all that these distributed *local* computations indeed converge towards the leader election, which is a *global* result. The refinement technique we use allows us to decompose the IEEE 1394 system into four embedded models, each one providing an additional view by bringing more informations into the current invariant. For instance, the first two models contains the essence of the underlying structure of the acyclic and connected graph representing the network. They also convey the main ideas of the distributed computation. They express the way the protocol works at a very high level abstraction. The third model formalizes how the nodes communicate by means of various kinds of messages: it helps us understanding the contention problem, which is one of the critical question of the IEEE 1394 protocol. The last model deals with the *localization* of the abstract data structures used in the previous models.

Related works. The IEEE 1394 protocol is a distributed algorithm for electing a leader in a network. The idea of the algorithm has already been sketched by N. Lynch [**?**](page 501). This sketch fits our second formal model. The PVS verification [59] derives the correctness of the IEEE 1394 protocol for an I/O automaton SPEC, which corresponds to our third formal model. We notice that this I/O automaton is not detailed enough to express the *confirmation* event, which appears in our third model. Their proofs are not really helpful for understanding the rôle of the underlying structure in the convergence of the algorithmic solution. The expressiveness of their invariant is not really clear. The PVS models includes an I/O automaton TIP that corresponds to our first formal model. A specific refinement relation is used to define the link between the two I/O automata, but it is not really useful to derive safety properties. Our approach keeps a link with the documentation and tends to explain in a formal way why the current abstract model is working correctly. We are really close to the IEEE 1394 protocol in our fourth abstract model. We shall not compare our approach to that of model checking, since our modeling is completely proved and is not restricted to a given network.

7.2.2 The Case Study: Basic Approach

The goal of the IEEE 1394 protocol is to elect in a *finite time* a specific node, called the *leader*, in a network made of various nodes linked by some communication channels. Once the leader is elected, each non-leader node in the network should have a well defined way to communicate with it. This election of the leader has to be done in a distributed and non-deterministic way.

The Basic Mathematical Structure

Before considering details of the protocol, we choose to give a very solid definition to the main topology of the network. It is essentially formalized by means of a set ND of nodes subjected to the following assumptions:

1. the network is represented by a graph g built on ND,
2. all nodes are concerned with the network,
3. the links between the nodes are *bidirectional*,
4. a node is *not directly connected to itself*.

$$
\begin{array}{l}
g \subseteq ND \times ND \\
\mathsf{dom}\,(g) = ND \\
g = g^{-1} \\
\mathsf{id}(ND) \cap g \;=\; \emptyset
\end{array}
$$

Items 2 and 3 above are formally represented by a *symmetric graph* whose domain (and thus co-domain too) corresponds to the entire *finite set* of nodes. The symmetry of the graph is due to the representation of the non-oriented graph by pairs of nodes and the link $x{-}y$ is represented by the two pairs $x \mapsto y$ and $y \mapsto x$. Item 4 is rendered by saying that the graph is *not reflexive*.

There are two other very important properties of the graph: it is *connected and acyclic*. Both these properties are formalized by claiming that the relation between each node and the spanning trees of the graph having that node as a root, that this relation is *total* and *functional*. In other words, each node in the graph can be associated with one and exactly one tree rooted at that node and spanning the graph. We can model a tree by a root r, which is a node: $r \in ND$, and a father functions t (each node has an unique father node, except the root): $t \;\in\; ND{-}\{r\} \;\longrightarrow\; ND$. The tree is an acyclic graph. A cycle c in a finite graph t built

on a set ND is a subset of ND whose elements are members of the inverse image of c under t, formally: $c \subseteq t^{-1}[c]$. To fulfil the requirement of acyclicity, the only set c that enjoys this property is thus the empty set. This can be formalized by the left predicate that follows, which can be proved to be *equivalent* to the one situated on the right, which can be used as an induction rule:

$$\boxed{\forall c \cdot (\, c \subseteq ND \;\wedge\; c \subseteq t^{-1}[c] \;\Rightarrow\; c = \emptyset\,)} \;\Leftrightarrow$$

$$\boxed{\forall q \cdot (\, q \subseteq ND \;\wedge\; r \in q \;\wedge\; t^{-1}[q] \subseteq q \;\Rightarrow\; ND = q\,)}$$

We prove the equivalence using the tool Atelier B. We can now define a spanning tree (with root r and father function t) of a graph g as one whose father function is included in g, formally:

$$\boxed{\mathsf{spanning}\,(r,t,g) \;\widehat{=}\; \left(\begin{array}{l} r \in ND \quad \wedge \\ t \in ND - \{r\} \longrightarrow ND \quad \wedge \\ \forall q \cdot (\, q \subseteq ND \;\wedge\; r \in q \;\wedge\; t^{-1}[q] \subseteq q \;\Rightarrow\; ND = q\,) \quad \wedge \\ t \subseteq g \end{array}\right)}$$

As mentioned above, each node in the graph can be associated with exactly one tree rooted at that node and which spans the graph. For this, we define the following total function f connecting each node r of the graph with its spanning tree $f(r)$:

$$\boxed{\begin{array}{l} f \in ND \rightarrow (ND \nrightarrow ND) \\ \\ \forall (r,t) \cdot \left(\begin{array}{l} r \in ND \;\wedge \\ t \in ND \nrightarrow ND \\ \Rightarrow \\ t = f(r) \;\Leftrightarrow\; \mathsf{spanning}\,(r,t,g) \end{array}\right) \end{array}}$$

The graph g and the function f are thus *two global constants of the problem*.

The First Model

From the basic mathematical structure developed in previous section, the essence of the abstract algorithm implemented by the protocol is very simple: it consists in building gradually (and non-deterministically) *one of the spanning trees* of the graph. Once this is done, then the *root* of that tree is *the elected leader* and the communication structure between the other nodes and the leader is obviously the *spanning tree itself*. The protocol, considered globally, has thus *two variables*: (1) the future spanning tree, sp, and (2) the future leader, ld.

The *first formal model* of the development contains the definitions and properties of the two global constants (the above graph g and function f together with their properties), and the definition of the two mentioned global variables sp and ld typed in a very loose way: sp is a binary relation built on ND and ld is a node. The dynamic aspect of the protocol is essentially made of one *event*, called elect, which claims *what the result of the protocol is, when it is completed*. In other words, at this level, there is no protocol, just the formal definition of its intended result, namely a spanning tree sp and its root ld.

```
elect ≙
  begin
    ld, sp : spanning (ld, sp, g)
  end
```

As can be seen, the election is done in one step. In other words, the spanning tree appears at once. The analogy of someone closing and opening eyes can be used here to "explain" the process of election at this very abstract level.

7.2.3 Refining the First Model

In this section, we present two successive refinements of the previous initial model. In the first one, we give the essence of the *distributed* algorithm. In the second refinement, we introduce some *communication mechanisms* between the nodes.

First Refinement: Gradual Construction of a Spanning Tree

In the first model, the construction of the spanning tree was performed in "one shot". Of course, in a more realistic (concrete) formalization, this is not the case any more. In fact, the tree is constructed on a step by step basis. For this, a new variable, called tr, and a new event, called progress, are introduced. The variable tr represents a sub-graph of g, it is made of several trees (it is thus a *forest*) which will *gradually converge* to the final tree, which we intend to build eventually. This convergence is performed by the event progress. This event involves two nodes x and y, which are neighbours in the graph g. Moreover, x and y are supposed to be both outside the domain of tr. In other words, each of them has no "father" yet in tr. However, the node x is the father of all its *other neighbours* (if any) in g. This last condition can be formalized by means of the predicate $g[\{x\}] = tr^{-1}[\{x\}] \cup \{y\}$ since the set of neighbours of x in g is $g[\{x\}]$ while the set of sons of x in tr is $tr^{-1}[\{x\}]$. When these conditions are fulfiled, then the event progress can be enabled and its action has the effect of making the node y the father of x in tr. The abstract event elect is now refined. Its new version is concerned with a node x which happens to be the father of all its neighbours in g. This condition is formalized by the predicate $g[\{x\}] = tr^{-1}[\{x\}]$. When this condition is fulfiled the action of elect makes x the leader ld and tr the spanning tree sp. Next are the formal representations of these events

progress $\widehat{=}$
any x, y **where**
$x, y \in g \ \wedge \ x \notin \mathsf{dom}(tr) \ \wedge \ y \notin \mathsf{dom}(tr) \ \wedge$
$g[\{x\}] = tr^{-1}[\{x\}] \cup \{y\}$
then
$tr := tr \cup \{x \mapsto y\}$
end

elect $\widehat{=}$
any x **where**
$x \in ND \ \wedge$
$g[\{x\}] = tr^{-1}[\{x\}]$
then
$ld, sp := x, tr$
end

The new event progress clearly refines $skip$ since it only updates the variable tr which is a *new variable* of this refinement with no existence in the abstraction. Also notice that progress clearly decreases the quantity $\mathsf{card}(g) - \mathsf{card}(tr)$. The situation is far less clear concerning the refinement of event elect. We have to prove that when its guard is true then tr is indeed a spanning tree of the graph g whose root is precisely x. Formally, this leads to proving the following

$$\forall x \cdot (x \in ND \ \wedge \ g[\{x\}] = tr^{-1}[\{x\}] \ \Rightarrow \ \mathsf{spanning}\,(x, tr, g))$$

According to the definition of the constant function f, the previous property is clearly equivalent to

$$\forall x \cdot (x \in ND \ \wedge \ g[\{x\}] = tr^{-1}[\{x\}] \ \Rightarrow \ tr = f(x))$$

This means that tr and $f(x)$ should have the same domain, namely $ND - \{x\}$, and that for all n in $ND - \{x\}$, $tr(n)$ is equal to $f(x)(n)$. This amounts to proving the following:

$$ND = \{x\} \cup \{ n \mid n \in ND - \{x\} \ \wedge \ f(x)(n) = tr(n) \}$$

This is done using the *inductive property* associated with each spanning tree $f(x)$. Notice that we also need the following invariants:

$$tr \in ND \nrightarrow ND$$
$$\mathsf{dom}\,(tr) \lhd (tr \cup tr^{-1}) = \mathsf{dom}\,(tr) \lhd g$$
$$tr \cap tr^{-1} = \emptyset$$

This new model, although more concrete than the previous one, is nevertheless still an abstraction of the "real" protocol: it just explains how the leader can be eventually elected by the gradual transformation of the forest tr into a unique tree spanning the graph g.

Second Refinement: Introducing Communication Channels

In the previous refinement, the event progress was still very abstract: as soon as two nodes x and y with the required properties were detected, the corresponding action took place immediately: in other words, y became the father of x "in one shot". In the "real" protocol things are not so "magic": once a node x has detected that it is the father of all its neighbours except one y, it sends a *request* to y in order to ask it to become its father. Node y then *acknowledges* this request and finally node x establishes the "father" connection with node y. This connection, which is thus established in *three distributed steps*, is clearly closer to what happens in the real protocol. We shall see however in the next refinement that what we have just described is not yet the final word. But let us formalized this for the moment. In order to do so, we need to define at least two new variables: req, to handle the requests, and ack, to handle the acknowledgements. req is a partial function from ND to itself. When a pair $x \mapsto y$ belongs to req it means that node x has send a request to node y asking it to become its father: the functionality of req is due to the fact that x has only one father. Clearly, req is also included in the graph g. When node y sends an acknowledgement to x this is because y has *already* received a request from x: ack is thus a partial function included in req.

$$
\begin{array}{l}
req \in ND \nrightarrow ND \\
req \subseteq g \\
ack \subseteq req \\
tr \subseteq ack \\
ack \cap ack^{-1} = \emptyset
\end{array}
$$

Notice that when a pair $x \mapsto y$ belongs to ack, it means that y has sent an acknowledgment to x (clearly y can send several acknowledgements since it might be the father of several nodes). It is also clear that it is not possible in this case for the pair $y \mapsto x$ to belong to ack. The final connection between x and y is still represented by the function tr. Thus tr is included in ack. All this can be formalized as shown.

Two new events are defined in order to manage requests and acknowledgements: send_req, and send_ack. As we shall see, event progress is modified, whereas event elect is left unchanged. Here are the new events and the refined version of progress:

send_req $\widehat{=}$
any x, y **where**
$x, y \in g \ \wedge \ y, x \notin ack \ \wedge$
$x \notin \mathsf{dom}\,(req) \ \wedge$
$g[\{x\}] = tr^{-1}[\{x\}] \cup \{y\}$
then
$req := req \cup \{x \mapsto y\}$
end

send_ack $\widehat{=}$
any x, y **where**
$x, y \in req \ \wedge$
$x, y \notin ack \ \wedge$
$y \notin \mathsf{dom}\,(req)$
then
$ack := ack \cup \{x \mapsto y\}$
end

progress $\widehat{=}$
any x, y **where**
$x, y \in ack \ \wedge$
$x \notin \mathsf{dom}\,(tr)$
then
$tr := tr \cup \{x \mapsto y\}$
end

Event send_req is enabled when a node x discovers that it is the father of all its neighbours except one y: $g[\{x\}] = tr^{-1}[\{x\}] \cup \{y\}$. Notice that, as expected, this condition is exactly the one that allowed event progress in the previous model to be enabled. Moreover x must not have sent already a request to any node: $x \notin \mathsf{dom}\,(req)$. Finally x must not have already sent an acknowledgement to node y: $y, x \notin ack$. When these conditions are fulfiled then the pair $x \mapsto y$ is added to req. Event send_ack is enabled when a node y receives a request from node x, moreover y must not have already sent an acknowledgement to node x: $x, y \in req$ and $x, y \notin ack$. Finally node y must not have sent a request to any node: $y \notin \mathsf{dom}\,(req)$ (we shall see very soon what happens when this condition does not hold). When these conditions are fulfiled, node y sends an acknowledgement to node x: the pair $x \mapsto y$ is thus added to ack. Event progress is enabled when a node x receives an acknowledgement from node y: $x, y \in ack$. Moreover node x has not yet established any father connection: $x \notin \mathsf{dom}\,(tr)$. When these conditions are fulfiled the connection is established: the pair $x \mapsto y$ is added to tr.

Events send_req and send_ack clearly refine $skip$. Moreover their actions increment the cardinal of req and ack respectively (these cardinals are bounded by that d g). It remains for us to prove that the new version of event progress is a correct refinement of its abstraction. The actions being the same, it just remains for us to prove that the concrete guard implies the abstract one. This amounts to proving the following left predicate, which is added as an invariant:

$$\forall\,(x,y)\cdot\left(\begin{array}{l} x,y\in ack \quad \wedge \\ x\notin \mathsf{dom}\,(tr) \\ \Rightarrow \\ x,y\in g \quad \wedge \\ x\notin \mathsf{dom}(tr) \quad \wedge \\ y\notin \mathsf{dom}(tr) \quad \wedge \\ g[\{x\}] \;=\; tr^{-1}[\{x\}]\cup\{y\} \end{array}\right)$$

$$\forall\,(x,y)\cdot\left(\begin{array}{l} x,y\in req \quad \wedge \\ x,y\notin ack \\ \Rightarrow \\ x,y\in g \quad \wedge \\ x\notin \mathsf{dom}(tr) \quad \wedge \\ y\notin \mathsf{dom}(tr) \quad \wedge \\ g[\{x\}] \;=\; tr^{-1}[\{x\}]\cup\{y\} \end{array}\right)$$

When trying to prove that the left predicate is maintained by event send_ack, we find that the right predicate above must also be proved. It is thus added as a new invariant, which is, this time, easily proved to be maintained by all events.

The problem of contention. The guard of the event send_ack above contains the condition $y \notin \mathsf{dom}\,(req)$. If this condition does not hold while the other two guarding conditions hold, that is $x, y \in req$ and $x, y \notin ack$ hold, then clearly x has sent a request to y and y has sent a request to x: each one of them wants the other to be its father! This problem is called the *contention* problem. In this case, no acknowledgements should be sent since then each node x and y would be the father of the other. In the "real" protocol the problem is "solved" by means of timers. As soon as a node y discovers a contention with node x, it waits for very a short delay in order to be certain that the other node x has also discovered the problem. The very short delay in question is at least equal to the message transfer time between nodes (such a time is supposed to be *bounded*). After this, each node randomly chooses (with probability 1/2) to wait for either a "short" or a "large" delay (the difference between the two is at least twice the message transfer time). After the chosen delay has passed each node sends a new request to the other *if it is in the situation to do so*. Clearly, if both nodes choose the same delay, the contention situation will reappear. However if they do not choose the same delay, then the one with the largest delay becomes the father of the other: when it wakes up, it discovers the request from the other while it has not itself already sent its own request, it can therefore send an acknowledgement and thus become the father. According to the *law of large numbers*, the probability for both nodes to indefinitely choose the same delay is null. Thus, at some point, they will (in probability) choose different delays and one of them will thus become the father of the other. We shall only present here a partial formalization of the contention problem. The idea is to introduce a *virtual channel* called cnt.

$$\begin{array}{l} cnt \;\subseteq\; req \\ ack \cap cnt \;=\; \emptyset \end{array}$$

When this "channel" contains a pair $x \mapsto y$, this means that y has discovered the contention with node x. When both pairs $x \mapsto y$ and $y \mapsto x$ are present in cnt, this means that both nodes x and y have discovered the contention. Notice that cnt is included in req and clearly disjoint with ack, as shown. We have two new events.

The first one is called discover_cnt. The only difference with the guard of event send_ack concerns the condition $y \in \mathsf{dom}\,(req)$, which is true in discover_cnt and false in send_ack. The action of this event adds the pair $x \mapsto y$ to cnt. The second new event is called solve_cnt. It is enabled when both pairs $x \mapsto y$ and $y \mapsto x$ are present in cnt. The action removes these pairs from req and resets cnt. This formalizes what happens after the "very short delay". Notice that this event is not part of the protocol: it corresponds to a "deamon" acting when the very short delay has just passed. Here are the events

discover_cnt $\widehat{=}$
any x, y **where**
$x, y \in req{-}ack \quad \wedge$
$y \in \mathsf{dom}\,(req)$
then
$cnt := cnt \cup \{x \mapsto y\}$
end

solve_cnt $\widehat{=}$
any x, y **where**
$x, y \in cnt \quad \wedge$
$y, x \in cnt$
then
$req, cnt := req{-}cnt, \emptyset$
end

In order to prove the invariant $ack \cap cnt = \emptyset$, we need the following extra invariants

$$\forall\,(x,y)\cdot\left(\begin{array}{l} x, y \in req{-}ack \quad \wedge \\ y \in \mathsf{dom}\,(req) \\ \Rightarrow \\ y, x \in req{-}ack \end{array}\right) \qquad \forall\,(x,y)\cdot\left(\begin{array}{l} x, y \in req{-}ack \quad \wedge \\ y \notin \mathsf{dom}\,(req) \\ \Rightarrow \\ x, y \notin cnt \end{array}\right)$$

The complete formalization of the contention solution of the real IEEE 1394 protocol (involving the timers and the random choices) is not difficult, just a little too long to be presented within the framework of this paper.

7.2.4 Last Refinement: Localization

In the previous refinement, the guards of the various events were defined in terms of some *global* constants or variables such as g, tr, req, ack. A closer look at this refinement shows that these constants or variables are used in expressions of the following shapes: $g^{-1}[\{x\}]$, $tr^{-1}[\{x\}]$, $ack^{-1}[\{x\}]$, $\mathsf{dom}\,(req)$, and $\mathsf{dom}\,(tr)$. These shapes dictate the kind of *data refinement* we now undertake. We declare five new variables nb (for neighbours), ch (for children), ac (for acknowledged), dr (for domain of req), and dt (for domain of tr). Next are the declarations of these variables together with their simple definitions in terms of the global variables.

$nb \in ND \to \mathbb{P}(ND)$
$ch \in ND \to \mathbb{P}(ND)$
$ac \in ND \to \mathbb{P}(ND)$
$dr \subseteq ND$
$dt \subseteq ND$

$\forall x \cdot (\, x \in ND \Rightarrow nb(x) = g^{-1}[\{x\}]\,)$
$\forall x \cdot (\, x \in ND \Rightarrow ch(x) \subseteq tr^{-1}[\{x\}]\,)$
$\forall x \cdot (\, x \in ND \Rightarrow ac(x) = ack^{-1}[\{x\}]\,)$
$dr = \mathsf{dom}\,(req)$
$dt = \mathsf{dom}\,(tr)$

Given a node x, the sets $nb(x)$, $ch(x)$, and $ac(x)$ are supposed to be "stored" locally within the node. As the varying sets $ch(x)$ and $ac(x)$ are subsets of the constant set $nb(x)$, it is certainly possible to further refine their encoding. Likewise the two sets dr and dt still appears to be global, but they can clearly be encoded locally in each node by means of local boolean variables.

It is worth noticing that the "definition" of variable ch above is not given in terms of an equality, rather in terms of an inclusion (this is thus not really a definition). This is due to the fact that the set $ch(y)$ cannot be updated while the event progress takes place: this is because this event can only act on its *local* data. A new event, receive_cnf (for receive confirmation) is thus necessary to update the set $ch(y)$. Next are the refinement of the various events.

elect $\widehat{=}$
any x **where**
$x \in ND \ \wedge$
$nb(x) = ch(x)$
then
$ld := x$
end

send_req $\widehat{=}$
any x, y **where**
$x \in ND{-}dr \ \wedge$
$y \in ND{-}ac(x) \ \wedge$
$nb(x) = ch(x) \cup \{y\}$
then
$req := req \cup \{x \mapsto y\} \ \|$
$dr := dr \cup \{x\}$
end

send_ack $\widehat{=}$
any x, y **where**
$x, y \in req \quad \wedge$
$x \notin ac(y) \quad \wedge$
$y \notin dr$
then
$ack := ack \cup \{x \mapsto y\} \ \|$
$ac(y) := ac(y) \cup \{x\}$
end

progress $\widehat{=}$
any x, y **where**
$x, y \in ack \quad \wedge$
$x \notin bt$ **then**
$tr := tr \cup \{x \mapsto y\} \ \|$
$dt := dt \cup \{x\}$
end

receive_cnf $\widehat{=}$
any x, y **where**
$x, y \in tr \ \wedge$
$x \notin ch(y)$
then
$ch(y) := ch(y) \cup \{x\}$
end

The proofs that these events correctly refine their respective abstractions are technically trivial. We now give in the following table, the *local node* "in charge" of each event as encoded above

event	*node*
elect	x
send_req	x
send_ack	y
progress	x
receive_cnf	y

The reader could be surprised to still see formulas such as $req := req \cup \{x \mapsto y\}$ or $x, y \in req$. They correspond in fact to writing and reading operations done by corresponding local nodes as explained in the following table:

$formula$	$explanation$
$req := req \cup \{x \mapsto y\}$	x sends a request to y
$x, y \in req$	y reads a request from x
$ack := ack \cup \{x \mapsto y\}$	y sends an acknowledgement to x
$x, y \in ack$	x reads an acknowledgement from y
$tr := tr \cup \{x \mapsto y\}$	x sends a confirmation to y
$x, y \in tr$	y reads a confirmation from y

7.2.5 Conclusion

The total number of proofs (all done mechanically with Atelier B) amounts to 106, where 24 required an easy interaction. Proofs help us to understand the contention problem and the rôle of graph properties in the correctness of the solution. The refinements gradually introduce the various invariants of the system. No assumption is made on the size of the network. The proof leads us to the discovery of the confirmation event to get the complete correctness, which was not the case of the I/O automata modelling.
In our opinion, this text, whose notation is very close to that of classical mathematics, is very simple to understand (provided, of course, the corresponding mathematical concepts, namely sets, functions, relations, and the like are well mastered), with the exception of our formulation of tree structures described under the form of the father function together with a universal quantification formalizing the corresponding induction rule. This formulation requires some more mathematical background. The question concerning the mythical *average programmer* understanding our solution is a bit irrelevant here: this problem is first, we believe, an abstract algorithm problem requiring a certain background in discrete mathematics. The lack of such background may lead to very awkward solutions due to the fact that they precisely try to convince the famous *average programmer*. In fact, in these solutions, the mathematical essence of the problem is hidden behind a curtain of technicalities all presented in a flat manner (no abstraction, thus no refinement, hence proof obligation explosion).
The essence of our approach is the methodology of *separation of concerns*: first prove the algorithm at an abstract (mathematical) level, then, and only then, gradually introduce the peculiarity of the specific protocol. What is important about our approach is that the fundamental properties we have proved at the beginning, namely the reachability and the uniqueness of a solution, are kept through the refinement process (provided, of course, the required proofs are done). It seems to us that this sort of approach is highly ignored in the literature of protocol developments where, most of the time, things are presented in a flat manner directly at the level of the final protocol itself.

7.3 A new leader election distributed algorithm

7.3.1 The Basic Mathematical Structure

Before considering details of the protocol, we choose to give a very solid definition to the main topology of the network. It is essentially formalized by means of a set ND of nodes subjected to the following assumptions:

1. the network is represented by a graph g built on ND,
2. the links between the nodes are *bidirectional*,
3. a node is *not directly connected to itself*.

$$\begin{array}{l} g \subseteq ND \times ND \\ g = g^{-1} \\ \mathsf{id}(ND) \cap g \;=\; \emptyset \end{array}$$

Items 2 and 3 above are formally represented by a *symmetric graph* whose domain (and thus co-domain too) corresponds to the entire *finite set* of nodes. The symmetry of the graph is due to the representation of the non-oriented graph by pairs of nodes and the link $x{-}y$ is represented by the two pairs $x \mapsto y$ and $y \mapsto x$. Item 4 is rendered by saying that the graph is *not reflexive*.

There are two other very important properties of the graph: it is *connected and acyclic*. Both these properties are formalized by claiming that the relation between each node and the spanning trees of the graph having that node as a root, that this relation is *total* and *functional*. In other words, each node in the graph can be associated with one and exactly one tree rooted at that node and spanning the graph. We can model a tree by a root r, which is a node: $r \in ND$, and a parent function t (each node has an unique parent node, except the root): $t \;\in\; ND{-}\{r\} \;\longrightarrow\; ND$. The tree is an acyclic graph. A cycle c in a finite graph t built on a set $N < D$ is a subset of ND whose elements are members of the inverse image of c under t, formally: $c \;\subseteq\; t^{-1}[c]$. To fulfil the requirement of acyclicity, the only set c that enjoys this property is thus the empty set. This can be formalized by the left predicate that follows, which can be proved to be *equivalent* to the one situated on the right, which can be used as an induction rule:

$$\boxed{\forall c \cdot (\, c \subseteq ND \;\wedge\; c \subseteq t^{-1}[c] \;\Rightarrow\; c = \emptyset \,)} \;\Leftrightarrow$$

$$\boxed{\forall q \cdot (\, q \subseteq ND \;\wedge\; r \in q \;\wedge\; t^{-1}[q] \subseteq q \;\Rightarrow\; ND = q \,)}$$

We prove the equivalence using the tools Atelier B [55] and B4free/Click'n'Prove [56]. We can now define a spanning tree (with root r and parent function t) of a graph g as one whose parent function is included in g, formally:

$$\mathsf{spanning}\,(r,t,g) \;\widehat{=}\; \left(\begin{array}{l} r \in ND \quad \wedge \\ t \;\in\; ND{-}\{r\} \;\longrightarrow\; ND \quad \wedge \\ \forall q \cdot (\, q \subseteq ND \;\wedge\; r \in q \;\wedge\; t^{-1}[q] \subseteq q \;\Rightarrow\; ND = q \,) \quad \wedge \\ t \;\subseteq\; g \end{array} \right)$$

As mentioned above, each node in the graph can be associated with exactly one tree rooted at that node and which spans the graph. For this, we define the following total function f connecting each node r of the graph with its spanning tree $f(r)$:

$$\begin{array}{l} f \in ND \rightarrow (ND \nrightarrow ND) \\ \\ \forall (r,t) \cdot \left(\begin{array}{l} r \in ND \;\wedge \\ t \in ND \nrightarrow ND \\ \Rightarrow \\ t = f(r) \;\Leftrightarrow\; \mathsf{spanning}\,(r,t,g) \end{array} \right) \end{array}$$

The graph g and the function f are thus *two global constants of the problem.* Since g and f are not instantiated, we have not to deal with the size of network and automatic techniques based on model checking

are not helpful for understanding how the algorithm is working. The special issue [57] presents a collection of verification techniques using model checking and the size of the network is clearly a practical bound. On the contrary, the verification using PVS [59] and I/O automata is more adequate than model checking, but invariants and proofs remain very difficult to understand. It is why we advocate the use of the refinement which provides and incremental way to derive both the algorithm and the proof. Moreover, the refinement allows us to derive a new leader election distributed algorithm, which is not possible in the verification-oriented approach.

7.3.2 The First Model $leaderelection0$: the one-shot election

From the basic mathematical structure developed in previous section, the essence of the abstract algorithm implemented by the protocol is very simple: it consists in building gradually (and non-deterministically) *one of the spanning trees* of the graph. Once this is done, then the *root* of that tree is *the elected leader* and the communication structure between the other nodes and the leader is obviously the *spanning tree itself*. The protocol, considered globally, has thus *two variables*: (1) the future spanning tree, sp, and (2) the future leader, ld. The gradual construction of the spanning tree simulates induction steps.

The *first formal model* of the development contains definitions and properties of the two global constants (the above graph g and function f together with their properties), and the definition of the two mentioned global variables sp and ld typed in a very loose way: sp is a binary relation built on ND and ld is a node. The dynamic aspect of the protocol is essentially made of one *event*, called elect, which claims *what the result of the protocol is, when it is completed*. In other words, at this level, there is no protocol, just the formal definition of its intended result, namely a spanning tree sp and its root ld.

```
elect ≙
  begin
    ld, sp : spanning (ld, sp, g)
  end
```

As can be seen, the election is done in one step. In other words, the spanning tree appears at once. The analogy of someone closing and opening eyes can be used here to *explain* the process of election at this very abstract level.

7.3.3 Refining the First Model $leaderelection0$

In this section, we present two successive refinements of the previous initial model. In the first one, we give the essence of the *distributed* algorithm. In the second refinement, we introduce some *communication mechanisms* between the nodes.

First Refinement $leaderelection1$: Gradual Construction of a Spanning Tree

In the first model $leaderelection0$, the construction of the spanning tree was performed in *one shot*. Of course, in a more realistic (concrete) formalization, this is not the case any more. In fact, the tree is constructed on a step by step basis. For this, a new variable, called tr, and a new event, called progress, are introduced. The variable tr represents a sub-graph of g, it is made of several trees (it is thus a *forest*) which will *gradually converge* to the final tree, which we intend to build eventually. This convergence is performed by the event progress. This event involves two nodes x and y, which are neighbours in the graph g. Moreover, x and y are supposed to be both outside the domain of tr. In other words, each of them has no *parent* yet in tr. However, the node x is the parent of all its *other neighbours* (if any) in g. This last condition can be formalized by means of the predicate $g[\{x\}] = tr^{-1}[\{x\}] \cup \{y\}$ since the set of neighbours of x in g is $g[\{x\}]$ while the set of sons of x in tr is $tr^{-1}[\{x\}]$. When these conditions are fulfilled, then the event progress can be enabled and its action has the effect of making the node y the parent of x in tr. The abstract event elect is now refined. Its new version is concerned with a node x which happens to be the parent of all its neighbours in g. This condition is formalized by the predicate $g[\{x\}] = tr^{-1}[\{x\}]$. When this condition is fulfilled the action of elect makes x the leader ld and tr the spanning tree sp. Next are the formal representations of these events

MODEL
$leaderelection0$
SETS
ND
CONSTANTS
g, f
DEFINITIONS
$spanning(r,t,g) ==$
$(r \in ND \wedge$
$t \in ND - \{r\} \longrightarrow ND \wedge$
$t \subseteq g \wedge$
$\forall S.(S \subseteq ND \wedge$
$r \in S \wedge$
$t^{-1}[S] \subseteq S$
$\Rightarrow$
$ND \subseteq S$
$))$
PROPERTIES
$g \in ND \leftrightarrow ND \wedge$
$g = g^{-1} \wedge$
$\mathrm{ID}(ND) \cap g = \emptyset \wedge$
$f \in ND \leftrightarrow (ND \nrightarrow ND) \wedge$
$\forall(n,fi).(n \in ND \wedge$
$fi \in ND \nrightarrow ND$
$\Rightarrow$
$($
$((n,fi) \in f)$
$\Leftrightarrow$
$spanning(n,fi,g)$
$)$
$) \wedge$
$f \in ND \longrightarrow (ND \nrightarrow ND)$
VARIABLES
ld, ts
INVARIANT
$ld \in ND \wedge$
$sp \in ND \nrightarrow ND$
ASSERTIONS
$\forall(n,fi).(n \in ND \wedge fi \in ND \nrightarrow ND \wedge (n,fi) \in f$
$\Rightarrow$
$fi \cap fi^{-1} = \emptyset$
$)$
INITIALISATION
$ld :\in ND \parallel sp :\in ND \nrightarrow ND$
EVENTS
$elect =$
begin
$ld, sp \in | (ld \in ND \wedge sp = f(ld))$
end
end

Figure 7.1: First model $leaderelection0$ for the distributed leader election algorithm

progress $\widehat{=}$
any x, y **where**
$x, y \in g \ \wedge \ x \notin \mathsf{dom}(tr) \ \wedge \ y \notin \mathsf{dom}(tr)$
$\wedge \ \ g[\{x\}] \ = \ tr^{-1}[\{x\}] \cup \{y\}$
then
$tr := tr \cup \{x \mapsto y\}$
end

elect $\widehat{=}$
any x **where**
$x \in ND \ \wedge$
$g[\{x\}] \ = \ tr^{-1}[\{x\}]$
then
$ld, sp := x, tr$
end

The new event progress clearly refines $skip$ since it only updates the variable tr which is a *new variable* of this refinement with no existence in the abstraction. Also notice that progress clearly decreases the quantity $\mathsf{card}(g) - \mathsf{card}(tr)$. The situation is far less clear concerning the refinement of event elect. We have to prove that when its guard is true then tr is indeed a spanning tree of the graph g whose root is precisely x. Formally, this leads to proving the following

$$\forall x \cdot (\ x \in ND \ \wedge \ g[\{x\}] = tr^{-1}[\{x\}] \ \Rightarrow \ \mathsf{spanning}\,(x, tr, g) \ )$$

According to the definition of the constant function f, the previous property is clearly equivalent to

$$\forall x \cdot (\ x \in ND \ \wedge \ g[\{x\}] = tr^{-1}[\{x\}] \ \Rightarrow \ tr = f(x) \ )$$

This means that tr and $f(x)$ should have the same domain, namely $ND - \{x\}$, and that for all n in $ND - \{x\}$, $tr(n)$ is equal to $f(x)(n)$. This amounts to proving the following:

$$ND \ = \ \{x\} \ \cup \ \{ \, n \mid n \in ND - \{x\} \ \wedge \ f(x)(n) = tr(n) \, \}$$

This is done using the *inductive property* associated with each spanning tree $f(x)$. Notice that we also need the following invariants:

$$\begin{array}{l} tr \in ND \nrightarrow ND \\ \mathsf{dom}\,(tr) \lhd (tr \cup tr^{-1}) \ = \ \mathsf{dom}\,(tr) \lhd g \\ tr \cap \ tr^{-1} \ = \ \emptyset \end{array}$$

This new model, although more concrete than the previous one, is nevertheless still an abstraction of the *real* protocol: it just explains how the leader can be eventually elected by the gradual transformation of the forest tr into a unique tree spanning the graph g.

Second Refinement $leaderelection2$: Introducing Communication Channels

In the previous refinement, the event progress was still very abstract: as soon as two nodes x and y with the required properties were detected, the corresponding action took place immediately: in other words, y became the parent of x *in one shot*. In the *real* protocol things are not so *magic*: once a node x has detected that it is the parent of all its neighbours except one y, it sends a *request* to y in order to ask it to become its parent. Node y then *acknowledges* this request and finally node x establishes the *parent* connection with node y. This connection, which is thus established in *three distributed steps*, is clearly closer to what happens in the real protocol. We shall see however in the next refinement that what we have just described is not yet the final word. But let us formalized this for the moment. In order to do so, we need to define at least two new variables: req, to handle the requests, and ack, to handle the acknowledgements. req is a partial function from ND to itself. When a pair $x \mapsto y$ belongs to req it means that node x has send a request to node y asking it to become its parent: the functionality of req is due to the fact that x has only one parent. Clearly, req is also included in the graph g. When node y sends an acknowledgement to x this is because y has *already* received a request from x: ack is thus a partial function included in req.

REFINEMENT
$leaderelection1$
REFINES
$leaderelection0$
VARIABLES
ld, sp, tr
INVARIANT
$tr \in ND \leftrightarrow ND \wedge$
$\text{DOM}(tr) \lhd (tr \cup tr^{-1}) = \text{DOM}(tr) \lhd g \wedge$
$tr \cap tr^{-1} = \emptyset$
ASSERTIONS
$\forall x.(x \in ND \wedge$
$g[\{x\}] = tr^{-1}[\{x\}]$
$\Rightarrow$
$f(x) = tr$
$);$
$\forall x.(x \in ND \wedge$
$g[\{x\}] = tr^{-1}[\{x\}]$
$\Rightarrow$
$(x, tr) \in f$
$)$
INITIALISATION
$ld :\in ND \parallel sp :\in ND \leftrightarrow ND \parallel tr := \emptyset$
EVENTS
elect =
any x **where**
$x \in ND \wedge$
$g[\{x\}] = tr^{-1}[\{x\}]$
then
$ld, sp := x, tr$
end ;
progress =
any x, y **where**
$x, y \in g \wedge$
$x \notin \text{DOM}(tr) \wedge y \notin \text{DOM}(tr) \wedge$
$g[\{x\}] = tr^{-1}[\{x\}] \cup \{y\}$
then
$tr := tr \cup \{x \mapsto y\}$
end
end

Figure 7.2: Second model $leaderelection1$ for the distributed leader election algorithm

$$
\begin{array}{l}
req \in ND \nrightarrow ND \\
req \subseteq g \\
ack \subseteq req \\
tr \subseteq ack \\
ack \cap ack^{-1} = \emptyset
\end{array}
$$

Notice that when a pair $x \mapsto y$ belongs to ack, it means that y has sent an acknowledgment to x (clearly y can send several acknowledgements since it might be the parent of several nodes). It is also clear that it is not possible in this case for the pair $y \mapsto x$ to belong to ack. The final connection between x and y is still represented by the function tr. Thus tr is included in ack. All this can be formalized as shown.

Two new events are defined in order to manage requests and acknowledgements: send_req, and send_ack. As we shall see, event progress is modified, whereas event elect is left unchanged. Here are the new events and the refined version of progress:

send_req $\widehat{=}$
any x, y **where**
$x, y \in g \ \wedge \ y, x \notin ack \ \wedge$
$x \notin \mathsf{dom}\,(req) \ \wedge$
$g[\{x\}] = tr^{-1}[\{x\}] \cup \{y\}$
then
$req := req \cup \{x \mapsto y\}$
end

send_ack $\widehat{=}$
any x, y **where**
$x, y \in req \ \wedge$
$x, y \notin ack \ \wedge$
$y \notin \mathsf{dom}\,(req)$
then
$ack := ack \cup \{x \mapsto y\}$
end

progress $\widehat{=}$
any x, y **where**
$x, y \in ack \ \wedge$
$x \notin \mathsf{dom}\,(tr)$
then
$tr := tr \cup \{x \mapsto y\}$
end

Event send_req is enabled when a node x discovers that it is the parent of all its neighbours except one y: $g[\{x\}] = tr^{-1}[\{x\}] \cup \{y\}$. Notice that, as expected, this condition is exactly the one that allowed event progress in the previous model to be enabled. Moreover x must not have sent already a request to any node: $x \notin \mathsf{dom}\,(req)$. Finally x must not have already sent an acknowledgement to node y: $y, x \notin ack$. When these conditions are fulfilled then the pair $x \mapsto y$ is added to req. Event send_ack is enabled when a node y receives a request from node x, moreover y must not have already sent an acknowledgement to node x: $x, y \in req$ and $x, y \notin ack$. Finally node y must not have sent a request to any node: $y \notin \mathsf{dom}\,(req)$ (we shall see very soon what happens when this condition does not hold). When these conditions are fulfilled, node y sends an acknowledgement to node x: the pair $x \mapsto y$ is thus added to ack. Event progress is enabled when a node x receives an acknowledgement from node y: $x, y \in ack$. Moreover node x has not yet established any parent connection: $x \notin \mathsf{dom}\,(tr)$. When these conditions are fulfilled the connection is established: the pair $x \mapsto y$ is added to tr.

Events send_req and send_ack clearly refine $skip$. Moreover their actions increment the cardinal of req and ack respectively (these cardinals are bounded by that d g). It remains for us to prove that the new version of event progress is a correct refinement of its abstraction. The actions being the same, it just remains for us to prove that the concrete guard implies the abstract one. This amounts to proving the following left predicate, which is added as an invariant:

$$
\forall\,(x, y) \cdot \left(
\begin{array}{l}
x, y \in ack \quad \wedge \\
x \notin \mathsf{dom}\,(tr) \\
\Rightarrow \\
x, y \in g \quad \wedge \\
x \notin \mathsf{dom}(tr) \quad \wedge \\
y \notin \mathsf{dom}(tr) \quad \wedge \\
g[\{x\}] \ = \ tr^{-1}[\{x\}] \cup \{y\}
\end{array}
\right)
$$

$$\forall\,(x,y)\cdot\left(\begin{array}{l} x,y\in req \quad \wedge \\ x,y\notin ack \\ \Rightarrow \\ x,y\in g \quad \wedge \\ x\notin \mathsf{dom}(tr) \quad \wedge \\ y\notin \mathsf{dom}(tr) \quad \wedge \\ g[\{x\}] \;=\; tr^{-1}[\{x\}]\cup\{y\} \end{array}\right)$$

When trying to prove that the left predicate is maintained by event send_ack, we find that the right predicate above must also be proved. It is thus added as a new invariant, which is, this time, easily proved to be maintained by all events.

The problem of contention. The guard of the event send_ack above contains the condition $y \notin \mathsf{dom}\,(req)$. If this condition does not hold while the other two guarding conditions hold, that is $x, y \in req$ and $x, y \notin ack$ hold, then clearly x has sent a request to y and y has sent a request to x: each one of them wants the other to be its parent! This problem is called the *contention* problem. In this case, no acknowledgements should be sent since then each node x and y would be the parent of the other. In the *real* protocol the problem is *solved* by means of timers. As soon as a node y discovers a contention with node x, it waits for very a short delay in order to be certain that the other node x has also discovered the problem. The very short delay in question is at least equal to the message transfer time between nodes (such a time is supposed to be *bounded*). After this, each node randomly chooses (with probability 1/2) to wait for either a *short* or a *large* delay (the difference between the two is at least twice the message transfer time). After the chosen delay has passed each node sends a new request to the other *if it is in the situation to do so*. Clearly, if both nodes choose the same delay, the contention situation will reappear. However if they do not choose the same delay, then the one with the largest delay becomes the parent of the other: when it wakes up, it discovers the request from the other while it has not itself already sent its own request, it can therefore send an acknowledgement and thus become the parent. According to the *law of large numbers*, the probability for both nodes to indefinitely choose the same delay is null. Thus, at some point, they will (in probability) choose different delays and one of them will thus become the parent of the other. Rather than to reuse the complete IEEE 1394 development [12], we reuse a part of the development and develop a new solution for solving the contention problem; the new algorithm was discovered after a misunderstanding of the IEEE 1394 initial solution.

When two nodes are in contention (and at most two nodes can be in contention, it has been proved mechanically and formally), each node can not send an acknowlegment to the other node; one of them should not be able to send this ack and the other one must do it. The main idea is to introduce a *unique counter* called ctr and it means that each node is uniquely identified and must be identifiable. In a real network, one can assume that equipments might be uniquely identified by an unique address, for instance, but it not the general rule. The IEEE 1394 protocol does not make any assumption on the identification of nodes.

$$ctr \;\in\; ND \rightarrowtail \mathbb{N}$$

The new event is called solve_cnt. Like for send_ack, the action of this event adds the pair $x \mapsto y$ to ack. The two differences with the guard of event send_ack concern the condition $y \in \mathsf{dom}\,(req)$, which is true in solve_cnt and false in send_ack and the guard $ctr(x) < ctr(y)$ is added to the event solve_cnt. Since ctr is an injection, both nodes x and y can not both trigger this event.

solve_cnt $\widehat{=}$
any x, y **where**
$x, y \in req{-}ack \quad \wedge$
$y \in \mathsf{dom}\,(req) \;\wedge$
$ctr(x) < ctr(y)$
then
$ack := ack \cup \{x \mapsto y\}$
end

The proof of the invariant requires the following extra invariants:

$$\forall\,(x,y)\cdot\left(\begin{array}{l} x,y\in req{-}ack \quad\wedge \\ y\in \mathsf{dom}\,(req) \\ \Rightarrow \\ y,x\in req \end{array}\right)$$

$$\forall\,(x,y)\cdot\left(\begin{array}{l} x,y\in req{-}ack \quad\wedge \\ y\in \mathsf{dom}\,(req)\ \wedge \\ ctr(x)<ctr(y) \\ \Rightarrow \\ x,y\notin ack \end{array}\right)$$

$$\forall\,(x,y,z)\cdot\left(\begin{array}{l} x,y\in req \quad\wedge \\ z\in g[\{x\}] \quad\wedge \\ z\neq y \\ \Rightarrow \\ z,x\in tr \end{array}\right)$$

The complete formalization of the contention solution of the real IEEE 1394 protocol (involving the timers and the random choices) is not addressed neither in the current development, nor in the paper [19]. Further work on the integration of timers should be done.

7.3.4 Last Refinements: Localization

In the previous refinement, the guards of the various events were defined in terms of *global* constants or variables such as g, tr, req, ack. A closer look at this refinement shows that these constants or variables are used in expressions of the following shapes: $g^{-1}[\{x\}]$, $tr^{-1}[\{x\}]$, $ack^{-1}[\{x\}]$, $\mathsf{dom}\,(req)$, and $\mathsf{dom}\,(tr)$. These shapes dictate the kind of *data refinement* we now undertake. Fourth, fifth and sixth models progressively introduce local informations, which are related to abstract global values. The models are in the figures 7.4 and 7.5; the model $leaderelection5$ introduces messages communications (TR, REQ, ACK). We declare five new variables nb (for neighbours), ch (for children), ac (for acknowledged), dr (for domain of req), and dt (for domain of tr). Next are the declarations of these variables together with their simple definitions in terms of the global variables.

$$\begin{array}{l} nb\in ND\to\mathbb{P}(ND) \\ ch\in ND\to\mathbb{P}(ND) \\ ac\in ND\to\mathbb{P}(ND) \\ dr\ \subseteq\ ND \\ dt\ \subseteq\ ND \end{array}$$

$$\begin{array}{l} \forall x\cdot(\,x\in ND\ \Rightarrow\ nb(x)\ =\ g^{-1}[\{x\}]\,) \\ \forall x\cdot(\,x\in ND\ \Rightarrow\ ch(x)\ \subseteq\ tr^{-1}[\{x\}]\,) \\ \forall x\cdot(\,x\in ND\ \Rightarrow\ ac(x)\ =\ ack^{-1}[\{x\}]\,) \\ dr=\mathsf{dom}\,(req) \\ dt=\mathsf{dom}\,(tr) \end{array}$$

Given a node x, the sets $nb(x)$, $ch(x)$, and $ac(x)$ are supposed to be *stored* locally within the node. As the varying sets $ch(x)$ and $ac(x)$ are subsets of the constant set $nb(x)$, it is certainly possible to further refine their encoding. Likewise the two sets dr and dt still appears to be global, but they can clearly be encoded locally in each node by means of local boolean variables.
It is worth noticing that the *definition* of variable ch above is not given in terms of an equality, rather in terms of an inclusion (this is thus not really a definition). This is due to the fact that the set $ch(y)$ cannot be updated while the event progress takes place: this is because this event can only act on its *local* data. A new event in $leaderelection3$, receive_cnf (for receive confirmation) is thus necessary to update the set $ch(y)$. Next are the refinement of the various events.

elect $\widehat{=}$
any x **where**
$x\in ND\ \wedge$
$nb(x)=ch(x)$
then
$ld:=x$
end

send_req $\widehat{=}$
any x,y **where**
$x\in ND{-}dr\ \wedge$
$y\in ND{-}ac(x)\ \wedge$
$nb(x)=ch(x)\cup\{y\}$
then
$req:=req\cup\{x\mapsto y\}\ \parallel$
$dr:=dr\cup\{x\}$
end

REFINEMENT
$leaderelection2$
REFINES
$leaderelection1$
CONSTANTS
ctr
PROPERTIES
$ctr \in ND \rightarrowtail \mathbb{N}$
VARIABLES
ld, tr, req, ack
INVARIANT
$req \in ND \nrightarrow ND \wedge$
$ack \in ND \nrightarrow ND \wedge$
$tr \subseteq ack \wedge$
$ack \subseteq req \wedge$
$req \subseteq g \wedge$
$\forall(x,y).(x,y \in req-ack \Rightarrow (x,y) \in g \wedge x \notin \text{DOM}(tr) \wedge g[\{x\}] = tr^{-1}[\{x\}] \cup \{y\}) \wedge$
$\forall(x,y).(x,y \in ack \wedge x \notin \text{DOM}(tr) \Rightarrow (x,y) \in g \wedge x \notin \text{DOM}(tr) \wedge y \notin \text{DOM}(tr) \wedge g[\{x\}] = tr^{-1}[\{x\}] \cup \{y\}) \wedge$
$ack \cap ack^{-1} = \emptyset \wedge$
$\forall(x,y).(x,y \in req \Rightarrow \forall z.(z \in g[\{x\}] \wedge z \neq y \Rightarrow z,x \in tr)) \wedge$
$\forall(x,y).(x,y \in req-ack \wedge y \in \text{DOM}(req) \Rightarrow (y,x) \in req) \wedge$
$\forall(x,y).(x,y \in req-ack \wedge y \in \text{DOM}(req) \wedge ctr(x) < ctr(y) \Rightarrow (y,x) \notin ack)$

ASSERTIONS
$ack \cap i\ d(ND) = \emptyset \wedge$
$req \cap i\ d(ND) = \emptyset \wedge$
$tr \cap \text{ID}\ (ND) = \emptyset$

INITIALISATION
$ld :\in ND \parallel tr := \emptyset \parallel req := \{\} \parallel ack := \emptyset$
EVENTS
elect =
any x **where**
$x \in ND \wedge g[\{x\}] = tr^{-1}[\{x\}]$
then
$ld := x$
end;
send_req =
any x, y **where**
$x,y \in g \wedge (y,x) \notin ack \wedge g[\{x\}] = tr^{-1}[\{x\}] \cup \{y\} \wedge x \notin \text{DOM}(req)$
then
$req := req \cup \{x \mapsto y\}$
end ;
send_ack =
any x, y **where**
$x,y \in req-ack \wedge y \notin \text{DOM}(req)$
then
$ack := ack \cup \{x \mapsto y\}$
end;
solve_cont =
any x, y **where**
$x,y \in req-ack \wedge y \in \text{DOM}(req) \wedge ctr(x) < ctr(y)$
then
$ack := ack \cup \{x \mapsto y\}$
end;
progress =
any x, y **where**
$x,y \in ack \wedge x \notin \text{DOM}(tr)$
then
$tr := tr \cup \{x \mapsto y\}$
end
end

Figure 7.3: Third model $leaderelection2$ for the distributed leader election algorithm

REFINEMENT
$leaderelection3$
REFINES
$leaderelection2$
CONSTANTS
nb
PROPERTIES
$nb \in ND \longrightarrow \mathcal{P}(ND) \wedge$
$\forall x.(x \in ND \Rightarrow nb(x) = g[\{x\}])$
VARIABLES
ld, ch, tr, req, ack
INVARIANT
$ch \in ND \longrightarrow \mathcal{P}(ND) \wedge$
$\forall x.(x \in ND \Rightarrow ch(x) \subseteq tr^{-1}[\{x\}])$
INITIALISATION
$ld :\in ND \parallel ch := ND \times \{\emptyset\} \parallel$
$req := \emptyset \parallel ack := \emptyset \parallel tr := \emptyset$
EVENTS
Local node x
elect =
any x **where**
$x \in ND \wedge nb(x) = ch(x)$
then
$ld := x$
end;
Local node x
send_req =
any x, y **where**
$x \in ND \wedge y \in ND \wedge$
$x \notin \text{DOM}(req) \wedge (y, x) \notin ack \wedge$
$nb(x) = ch(x) \cup \{y\}$
then
$req := req \cup \{x \mapsto y\}$
sending the message x to y
end ;
Local node y
receive_cnf =
any x, y **where**
$x, y \in tr \wedge x \notin ch(y)$
y is reading the x confirmation
then
$ch(y) := ch(y) \cup \{x\}$
end
end

REFINEMENT
$leaderelection4$
REFINES
$leaderelection3$
VARIABLES
$ld, ch, tr, req, ack, dr, ac, dt$
INVARIANT
$dr = \text{DOM}(req) \wedge$
$ac \in ND \longrightarrow \mathcal{P}(ND) \wedge$
$\forall x.(x \in ND \Rightarrow ac(x) = ack^{-1}[\{x\}]) \wedge$
$dt = \text{DOM}(tr)$

INITIALISATION
$ld :\in ND \parallel ch := ND \times \{\emptyset\} \parallel$
$req := \emptyset \parallel ack := \emptyset \parallel$
$tr := \emptyset \parallel dr, dt := \emptyset, \emptyset \parallel$
$ac := ND \times \{\emptyset\}$
EVENTS
Local node x
elect =
any x **where**
$x \in ND \wedge nb(x) = ch(x)$
then
$ld := x$
end;
Local node x
send_req =
any x, y **where**
$x \in ND \wedge y \in ND \wedge$
$x \notin dr \wedge y \notin ac(x) \wedge$
$nb(x) = ch(x) \cup \{y\}$
then
$req := req \cup \{x \mapsto y\} \parallel$
$dr := dr \cup \{x\}$
end ;
Local node y
send_ack =
any x, y **where**
$x, y \in req \wedge x \notin ac(y) \wedge y \notin dr$
then
$ack := ack \cup \{x \mapsto y\} \parallel$
$ac(y) := ac(y) \cup \{x\}$
end;
Local node y
solve_cont =
any x, y **where**
$x, y \in req \wedge x \notin ac(y) \wedge y \in dr \wedge$
$cpt(x) < cpt(y)$
then
$ack := ack \cup \{x \mapsto y\} \parallel$
$ac(y) := ac(y) \cup \{x\}$
end;
progress =
any x, y **where**
$x, y \in ack \wedge x \notin dt$
then
$tr := tr \cup \{x \mapsto y\} \parallel$
$dt := dt \cup \{x\}$
end ;
Local node y
receive_cnf =
any x, y **where**
$x, y \in tr \wedge x \notin ch(y)$
then
$ch(y) := ch(y) \cup \{x\}$
end
end

Figure 7.4: Fourth and fifth models $leaderelection3$ and $leaderelection4$ for the distributed leader election algorithm

REFINEMENT $leaderelection5$
REFINES $leaderelection4$
VARIABLES
$ld, ch, dr, ac, dt, REQ, ACK, TR$
INVARIANT
$REQ \in ND \leftrightarrow ND \wedge$
$req = REQ \cup ack \wedge$
$REQ \cap ack = \emptyset \wedge$
$ACK \in ND \leftrightarrow ND \wedge$
$TR \in ND \leftrightarrow ND \wedge$
$TR \subseteq tr \wedge$
$ack = ACK \cup tr \wedge$
$ACK \cap tr = \emptyset \wedge$
$\forall(x, y).(x, y \in TR \Rightarrow x \notin ch(y))$
INITIALISATION
$ld :\in ND \parallel ch := ND \times \{\emptyset\} \parallel$
$REQ := \emptyset \parallel ACK := \emptyset \parallel$
$TR := \emptyset \parallel dr, dt := \emptyset, \emptyset \parallel$
$ac := ND \times \{\emptyset\}$
EVENTS
send_req =
any x, y **where**
$x \in ND \wedge y \in ND \wedge x \notin dr \wedge$
$y \notin ac(x) \wedge nb(x) = ch(x) \cup \{y\}$
then
$REQ := REQ \cup \{x \mapsto y\} \parallel$
$dr := dr \cup \{x\}$
end ;
send_ack =
any x, y **where**
$x, y \in REQ \wedge x \notin ac(y) \wedge y \notin dr$
then
$REQ := REQ - \{x \mapsto y\} \parallel$
$ACK := ACK \cup \{x \mapsto y\} \parallel$
$ac(y) := ac(y) \cup \{x\}$
end;
solve_cont =
any x, y **where**
$x, y \in REQ \wedge x \notin ac(y) \wedge$
$y \in dr \wedge ctr(x) < ctr(y)$
then
$REQ := REQ - \{x \mapsto y\} \parallel$
$ACK := ACK \cup \{x \mapsto y\} \parallel$
$ac(y) := ac(y) \cup \{x\}$
end;
progress =
any x, y **where**
$x, y \in ACK \wedge x \notin dt$
then
$ACK := ACK - \{x \mapsto y\} \parallel$
$TR := TR \cup \{x \mapsto y\} \parallel$
$dt := dt \cup \{x\}$
end;
receive_cnf =
any x, y **where**
$x, y \in TR$
then
$TR := TR - \{x \mapsto y\} \parallel$
$ch(y) := ch(y) \cup \{x\}$
end
end

Figure 7.5: Sixth model $leaderelection5$ for the distributed leader election algorithm

send_ack $\widehat{=}$
any x, y **where**
$x, y \in req \quad \wedge$
$x \notin ac(y) \quad \wedge$
$y \notin dr$
then
$ack := ack \cup \{x \mapsto y\} \ \|$
$ac(y) := ac(y) \cup \{x\}$
end

solve_cnt $\widehat{=}$
any x, y **where**
$x, y \in req \quad \wedge$
$x \notin ac(y) \quad \wedge$
$y \in dr \quad \wedge$
$ctr(x) < ctr(y)$
then
$ack := ack \cup \{x \mapsto y\} \ \|$
$ac(y) := ac(y) \cup \{x\}$
end

progress $\widehat{=}$
any x, y **where**
$x, y \in ack \quad \wedge$
$x \notin dt$
then
$tr := tr \cup \{x \mapsto y\} \ \|$
$dt := dt \cup \{x\}$
end

receive_cnf $\widehat{=}$
any x, y **where**
$x, y \in tr \ \wedge$
$x \notin ch(y)$
then
$ch(y) := ch(y) \cup \{x\}$
end

Proofs that these events correctly refine their respective abstractions are technically trivial. We now give in the following table, the local node *in charge* of each event as encoded above

event	*node*
elect	x
send_req	x
send_ack	y
solve_cnt	y
progress	x
receive_cnf	y

The reader could be surprised yet to see formulas such as $req := req \cup \{x \mapsto y\}$ or $x, y \in req$. They correspond in fact to writing and reading operations done by corresponding local nodes as explained in the following table:

$formula$	$explanation$
$req := req \cup \{x \mapsto y\}$	x sends a request to y
$x, y \in req$	y reads a request from x
$ack := ack \cup \{x \mapsto y\}$	y sends an acknowledgement to x
$x, y \in ack$	x reads an acknowledgement from y
$tr := tr \cup \{x \mapsto y\}$	x sends a confirmation to y
$x, y \in tr$	y reads a confirmation from y

The total number of proofs (all done mechanically with Atelier B [55] and B4free/Click'n'Prove [56]) amounts to 106, where 24 required an easy interaction. Proofs help us to understand the contention problem and the rôle of graph properties in the correctness of the solution. The refinements gradually introduce the various invariants of the system. No assumption is made on the size of the network. The proof leads us to the discovery of the confirmation event to get the complete correctness and we choose to introduce a priority mechanism to solve the contention, which is not the solution of the IEEE 1394 protocol: a new leader election distributed algorithm is proposed. ACK,REQ and TR model communication channels; they contain messages which are currently sent and not yet received. We give the algorithm for the local node x and x sends messages to another node y. We assume that each site has a unique number and ctr is defined by this assignment.

Leader Election Algorithm
Local Node $x \in ND$
Local variables $nb, ch, ac \subseteq ND, ld \in ND, dr, dt \in Bool$

if $nb = ch$ **then** $ld := x$ **fi**
if $mes(y, ack) \in ACK \wedge y \notin dt$
then
$send(mes(x, tr), y) \parallel dt := dt \cup \{y\} \parallel$
$ACK := ACK - \{mes(y, ack)\}$ **fi**
if $\neg dr \wedge y \notin ac \wedge nb = ch \cup \{y\}$
then
$send(mes(x, req), y) \parallel dr := TRUE$ **fi**
if $mes(y, req) \in REQ \wedge y \notin ac \wedge \neg dr$
then
$send(mes(x, ack), y) \parallel ac := ac \cup \{y\} \parallel$
$REQ := REQ - \{mes(y, req)\}$ **fi**
if $mes(y, req) \in REQ \wedge y \notin ac \wedge dr \wedge ctr(y) < ctr(x)$
then
$send(mes(x, ack), y) \parallel ac := ac \cup \{y\} \parallel$
$REQ := REQ - \{mes(y, req)\}$ **fi**
if $mes(y, tr) \in TR \wedge y \notin ch$
then
$ch := ch \cup \{y\} \parallel TR := TR - \{mes(y, tr)\}$ **fi**

We have used programming-like notations for modelling messages communications (see model $leaderelection5$ 7.5) and we detail the meaning of each communication primitive:

- $send(mes(x, req), y)$ adds the message $mes(x, req)$ to REQ.
- $send(mes(x, ack), y)$ adds the message $mes(x, req)$ to ACK.
- $send(mes(x, tr), y)$ adds the message $mes(x, req)$ to TR.

Our algorithm is correct with respect to the invariant of the development; we have not mentionned the question of termination. The termination is derived, when one assumes a minimal fairness for each site: if a site can trigger an event, it will eventually trigger it, as long as it remains enabled.

Chapter 8

Conclusion

B gathers a large community of users whose contributions go beyond the scope of this document; we focus our topics on the event B approach to illustrate the foundations of B. Before to conclude our text, we should complete the B landscape by an outline of work on B and with B.

8.1 Work on B and with B

The series of conferences [66, 29, 33, 30, 31] on B (in association with the Z community) and books [21, 73, 67, 93, 63] on B demonstrate the strong activity on B. The expressivity of the B language lead to three kinds of work using concepts of B: extension of the B method, combination of B with another approach and applications of B. We have already mentioned applications of the B method in the introduction and, now, we sketch extensions of B and proposals to integrate B with other methods:

8.1.1 Extending the B method

The concept of event as introduced in B by Abrial [1] acts on the global state space of the system and has no parameter; on the contrary, Papatsaras and Stoddart [86] contrast this global style of development with one based on interacting components which communicate by means of shared events; parameters in events are permitted. The parametrisation of events is also considered by Butler and Walden [38] who are implementing action systems in the B AMN.
Events may or may not happen and new modalities are required to manage them; the language of assertions of B is becoming too poor to express temporal properties like liveness, for instance. Abrial and Mussat [13] introduce modalities into abstract systems and develop proof obligations related to liveness properties; Méry [80] shows how the B concepts can be easily used to deal with liveness and fairness properties. Bellegarde et al [28] analyse the extension of B using the LTL logic and the impact on the refinement of event systems. Problems are related to the refinement of systems while maintaining liveness and even fairness properties; it is difficult and in many cases not possible, because the refinement maintains previously validated properties of the abstract model and it can not maintain every liveness property.
Recently, McIver et al [79] extend the Generalized Substitution Language to handle probability in B; an abstract probabilistic choice is added to B operators. A methodology is proposed to use this extension.

8.1.2 Combining B with another formalim

The limited expressivity of the B language has inspired work on several proposals. Butler [36] investigates a mixed language including B AMN and CSP; CSP is used to structure abstract machines; the idea is exploited by Schneider and Treharne [96, 91] who control B machines.
Since diagrammatic formalisms offer a visual representation of models, another integration of B with UML is achieved by Butler [37] and by Le Dang et al [75, 74, 76]; B provides a semantical framework to UML components and allows one to analyse UML models. An interesting problem would be to study the impact of the B refinement into UML models.

Mikhailov and Butler [81] combine the theorem proving and the model checking and focus on the B-method and a theorem proving tool associated with it, and the Alloy specification notation and its model checker *Alloy Constraint Analyser*. Software development in B can be assisted using Alloy and Alloy can be used for verifying refinement of abstract specifications.

8.2 On the proof process

The proof process is supported by a proof assistant which is either a part of the environment called Atelier B [55] , or an environment called Click'n'Prove [17]. A free version is available [56]. Works on theories and reusing theories have been addressed by J.-R. Abrial et all in [9].

8.3 Final remarks

The design of (software) systems is an activity based on logico-mathematical concepts such as set-theoretical definitions; it gives rise to proof obligations that capture the essence of its correctness. The use of theoretical concepts is mainly due to the requirements of safety and quality of developed systems; it appears that the mathematics can help in improving the quality of software systems. B is a method that can help the designers to construct safer systems and it provides a realistic framework for developing a pragmatic engineering. Mathematical theories [9] can be derived from scratch or reused; in forthcoming work, mechanisms for re-usability of developments will demonstrate the increasing power of the applicability of B to realistic case studies [12, 45, 18]. Tools are already very helpful and will evolve towards a tool-set for developing systems. The proof tool is probably a crucial element in the B approach and recent developments of the prover, combined with the refinement, validates the applicability of the B method to derive correct reactive systems from abstract specifications. Another promising point is the introduction of patterns in the event B methodology. In [8], Abrial describes the new B method mainly related to B events; the project RODIN [89] aims to create a methodology and supporting open tool platform for the cost effective rigorous development of dependable complex software systems and services, especially using the event B method; it will provide a suitable framework for further work on event B.

Acknowledgements

We thank J.-R. Abrial for his permanent help, support and comments; Dines Bjoerner and Martin Henson have accepted a long delay for obtaining LaTeXfiles and we thank them for their support. It was a pleasure to spend two weeks with Dines and Martin in Slovakia and we especially enjoy the daily pedagogical meetings. Thanks!

Bibliography

[1] J.-R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias, editor, *1st Conference on the B method*, pages 169–190, November 1996.

[2] J.-R. Abrial. Event driven distributed program construction. Internal note, Consultant, August 2001. jr@.abrial.org.

[3] J.-R. Abrial. Event driven distributed program construction. Internal Note Version 5(Août 2001), Consultant, août 2001.

[4] J.-R. Abrial. Event driven electronic circuit construction. Internal note, Consultant, August 2001. jr@.abrial.org.

[5] J.-R. Abrial. Event driven sequential program construction. Internal note, Consultant, August 2001. jr@.abrial.org.

[6] J.-R. Abrial. Formal construction of proved circuits. Internal Note Version 4(Août 2001), Consultant, août 2001.

[7] J.-R. Abrial. Discrete system models. Internal note, Consultant, February 2002. jr@.abrial.org.

[8] J.-R. Abrial. B$^{\#}$: Toward a synthesis between z and b. In D. Bert and M. Walden, editors, *3nd International Conference of B and Z Users - ZB 2003, Turku, Finland*, Lectures Notes in Computer Science. Springer Verlag, June 2003.

[9] J.-R. Abrial, D. Cansell, and G. Laffitte. Higher-Order Mathematics in B. In D. Bert, J.P. Bowen, M.C. Henson, and K. Robinson, editors, *Formal Specification and Development in Z and B - ZB'2002, Grenoble, France*, volume 2272 of *Lecture Notes in Computer Science*, pages 370–393. Springer Verlag, January 2002.

[10] J.-R. Abrial, D. Cansell, and D. Méry. A mechanically proved and incremental development of the IEEE 1394 Tree Identify Protocol . Technical report, LORIA, March 2001.

[11] J.-R. Abrial, D. Cansell, and D. Méry. Formal derivation of spanning trees algorithms. In D. Bert and M. Walden, editors, *3nd International Conference of B and Z Users - ZB 2003, Turku, Finland*, Lectures Notes in Computer Science. Springer Verlag, June 2003.

[12] J.-R. Abrial, D. Cansell, and D. Méry. A mechanically proved and incremental development of IEEE 1394 Tree Identify Protocol. *Formal Aspects of Computing*, 14(3), April 2003.

[13] J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In D. Bert, editor, *B'98 :Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[14] Jean-Raymond Abrial. B$^{\#}$: Toward a synthesis between z and b. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina A. Waldén, editors, *ZB*, volume 2651 of *Lecture Notes in Computer Science*, pages 168–177. Springer, 2003.

[15] Jean-Raymond Abrial. Event based sequential program development: Application to constructing a pointer program. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 51–74. Springer, 2003.

[16] Jean-Raymond Abrial. Formal methods in industry: achievements, problems, future. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 761–768. ACM, 2006.

[17] Jean-Raymond Abrial and Dominique Cansell. Click'n'prove: Interactive proofs within set theory. In David Basin and Burkhart Wolff, editors, *16th Intl. Conf. Theorem Proving in Higher Order Logics (TPHOLs'2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 1–24. Springer Verlag, September 2003.

[18] Jean-Raymond Abrial and Dominique Cansell. Formal construction of a non-blocking concurrent queue algorithm (a case study in atomicity). *J. UCS*, 11(5):744–770, 2005.

[19] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of ieee 1394 tree identify protocol. *Formal Asp. Comput.*, 14(3):215–227, 2003.

[20] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Refinement and reachability in event$_{\mathrm{b}}$. In Treharne et al. [97], pages 222–241.

[21] J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5.

[22] J.R. Abrial. Event-driven sequential programs. ps file, March 2000.

[23] R.-J. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1998.

[24] R.-J. Back and J. von Wright. *Refinement Calculus*. Springer Verlag, 1998.

[25] R. J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1979.

[26] J. P. Banâtre, A. Coutant, and D. Le Métayer. The γ-model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[27] P. Behm, P. Benoit, A. Faivre, and J.-M.Meynadier. METEOR : A successful application of B in a large project. In *Proceedings of FM'99: World Congress on Formal Methods*, Lecture Notes in Computer Science, pages 369–387, 1999.

[28] F. Bellegarde, C. Darlot, J. Julliand, and O. Kouchnarenko. Reformulate dynamic properties during B refinement and forget variants and loop invariants. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *ZB 2000: Formal Specification and Development in Z and B - First International Conference of B and Z Users*, York,UK, August 29 - September 2 2000.

[29] D. Bert, editor. *B'98: Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science*, Montpellier, France, April 22-24 1998. Springer Verlag.

[30] D. Bert, J.-P. Bowen, M. C. Henson, and K. Robinson, editors. *ZB 2002: Formal Specification and Development in Z and B - 2nd International Conference of B and Z Users*, volume 2272 of *Lecture Notes in Computer Science*, Grenoble, France, January 2002. Springer Verlag.

[31] D. Bert, J.-P. Bowen, S. King, and M. Waldén, editors. *ZB 2003: Formal Specification and Development in Z and B - Third International Conference of B and Z Users*, volume 2651 of *Lecture Notes in Computer Science*, Turku, Finland, January 2003. Springer Verlag.

[32] J. Bicarregui, D. Clutterbuck, G. Finnie, H. Haughton, K. Lano, H. Lesan, W. Marsh, B. Matthews, M. Moulding, A. Newton, B. Ritchie, T. Rushton, and P. Scharbach. Formal methods into practise: Case studies in the application of the B method. Internal report, BUT Project, 1995.

[33] J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors. *ZB 2000: Formal Specification and Development in Z and B - First International Conference of B and Z Users*, York,UK, August 29 - September 2 2000.

[34] L. Burdy. *Traitrement des expressions dépourvues de sens de la théorie des ensembles Application à la méthode B*. PhD thesis, CNAM, 2000.

[35] M. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27:139–173, 1996.

[36] M. Butler. csp2b: A practical approach to combining csp and b. *Formal Aspects of Computing*, 12:182–196, 200.

[37] M. Butler and C. Snook. Verifying dynamic properties of UML models by translation to the B language and toolkit. In *UML 2000 WORKSHOP Dynamic Behaviour in UML Models: Semantic Questions*, October 2000.

[38] M. Butler and M. Walden. Parallel Programming with the B Method. In *Program Development by Refinement Cases Studies Using the B Method*, volume [93] of *FACIT*, pages 183–195. Springer Verlag, 1998.

[39] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, April 2003.

[40] M. Büchi and R. Back. Compositional symmetric sharing in B. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[41] D. Cansell, G. Gopalakrishnan, M. Jones, D. Méry, and A. Weinzoepflen. Incremental proof of the producer/consumer property for the PCI protocol. In D. Bert, editor, *Formal Specification and Development in Z and B - ZB'2002, Grenoble, France*, volume 2272 of *Lecture Notes in Computer Science*. Springer Verlag, January 2002.

[42] D. Cansell and D. Méry. Abstraction and refinement of features. In Ryan Stephen, Gilmore et Mark, editor, *Language Constructs for Designing Features*. Springer Verlag, 2000.

[43] D. Cansell and D. Méry. Développement de fonctions définies récursivement en B : Application du B événementiel. Rapport de recherche, LORIA UMR 7503, January 2002.

[44] D. Cansell and D. Méry. Développement de fonctions définies récursivement en B : Application du B événementiel. Rapport de recherche, LORIA UMR 7503, January 2002.

[45] D. Cansell and D. Méry. Formal and incremental construction of distributed algorithms: On the distributed reference counting algorithm. *Theoretical Computer Science*, 2006.

[46] D. Cansell and D. Méry. *Software Specification Methods An Overview Using a Case Study*, chapter Event B. Hermès, 2006. ISBN: 1905209347.

[47] Dominique Cansell, Ganesh Gopalakrishnan, Michael D. Jones, Dominique Méry, and Airy Weinzoepflen. Incremental proof of the producer/consumer property for the pci protocol. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB*, volume 2272 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2002.

[48] Dominique Cansell and Dominique Méry. Foundations of the b method. *Computers and Informatics*, 22, 2003.

[49] Dominique Cansell and Dominique Méry. Incremental parametric development of greedy algorithms. In S. Merz and T. Nipkow, editors, *AVOCS'O6 Sixth International Workshop on Automated Verification of Critical Systems*, 18-19 September 2006.

[50] N. Cariero and D. Gelernter. *How to write parallel programs: a first course*. The MIT Press, 1990.

[51] Jérémie Chalopin and Yves Métivier. A bridge between the asynchronous message passing model and local computations in graphs. In Joanna Jedrzejowicz and Andrzej Szepietowski, editors, *MFCS*, volume 3618 of *Lecture Notes in Computer Science*, pages 212–223. Springer, 2005.

[52] K. M. Chandy and J. Misra. *Parallel Program Design A Foundation*. Addison-Wesley Publishing Company, 1988. ISBN 0-201-05866-9.

[53] Michel Chaudron. Notions of Refinement for a Coordination Language for GAMMA. Technical report, Leiden University, The Netherlands, 1997.

[54] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[55] ClearSy, Aix-en-Provence (F). *Atelier B*, 2002. Version 3.6.

[56] ClearSy. Web site b4free set of tools for development of b models. `http://www.b4free.com/index.php`, 2004.

[57] John Cooke, Savi Maharaj, Judi Romijn, and Carron Shankland, editors. *Formal Aspects of Computing*, volume 14. Springer, April 2003.

[58] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.

[59] M. Devillers, D. Griffioen, J. Romin, and F. Vaandrager. Verification of a Leader Election Protocol: Formal Methods Applied to IEEE 1394. *Formal Methods in System Design*, 16:307–320, 2000. Kluwer Academic Publishers.

[60] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[61] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer Verlag, 1990.

[62] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science. Springer Verlag, w. brauer and r. rozenberg and a. salomaa edition, 1985.

[63] M. Frappier and H. Habrias, editors. *Software Specification Methods An Overview Using a Case Study*. Hermes Science Publishing, London, England, April 2006. ISBN: 1905209347.

[64] Olivier Galibert. YLC - Linda C++, 1997. 1997.

[65] Y. Gurevitch. *Specification and Validation Methods*, chapter "Evolving Algebras 1993: Lipari Guide", pages 9–36. Oxford University Press, 1995. Ed. E. Börger.

[66] H. Habrias, editor. *First Conference on the B Method*, Nantes, France, April 22-24 1996. IRIN-IUT de Nantes. ISBN 2-906082-25-2.

[67] H. Habrias. *Spécification formelle avec B*. Hermès, 2001.

[68] J. Hoare. The use of B in CICS. In *Applications of Formal Methods*, 1995.

[69] C. B. Jones. *Sytematic Software Development Using VDM*. Prentice-Hall International, 1986.

[70] J. B. Kruskal. On the shortest spanning subtree and the traveling salesman problem. *Proc. Am. Math. Soc.*, 7:48–50, 1956.

[71] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[72] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[73] K. Lano. *The B Language and Method - A Guide to rPactical Formal Development*. FACIT. Springer Verlag, 1996.

[74] Hung Ledang and Jeanine Souquières. Formalizing UML behavioral diagrams with B. In *Tenth OOPSLA Workshop on Behavioral Semantics : Back to Basics , Tampa Bay, Florida, USA*, Oct 2001.

[75] Hung Ledang and Jeanine Souquières. Modeling class operations in B : application to UML behavioral diagrams. In IEEE Computer Society, editor, *16th IEEE International Conference on Automated Software Engineering - ASE'2001, Loews Coronado Bay, San Diego, USA*, Nov 2001.

[76] Hung Ledang and Jeanine Souquières. Contributions for modelling UML state-charts in B. In Springer Verlag, editor, *Third International Conference on Integrated Formal Methods - IFM'2002, Turku, Finland*, May 2002.

[77] B-Core(UK) Ltd. *B-Toolkit User's Manual*, relase 3.2 edition, 1996.

[78] Z. Manna. *Mathematical Theory of Computation*. Mac Graw Hill, 1974.

[79] A. McIver, C. Morgan, and T. S. Hoang. Probabilistic termination in B. In D. Bert, J.-P. Bowen, S. King, and M. Waldén, editors, *ZB 2003: Formal Specification and Development in Z and B - Third International Conference of B and Z Users*, volume 2651 of *Lecture Notes in Computer Science*, Turku, Finland, January 2003. Springer Verlag.

[80] D. Méry. Requirements for a temporal B : Assigning Temporal Meaning to Abstract Machines ... and to Abstract Systems. In A. Galloway and K. Taguchi, editors, *IFM'99 Integrated Formal Methods 1999*, Workshop on Computing Science, YORK, June 1999.

[81] L. Mikhailov and M. Butler. An approach to combining B and Alloy. In D. Bert, J.-P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B - 2nd International Conference of B and Z Users*, volume 2272 of *Lecture Notes in Computer Science*, Grenoble, France, January 2002. Springer Verlag.

[82] L. Moreau. Distributed directory service and message routing for mobile agents. *Science of Computer Programming*, 39(2–3):249–272, 2001.

[83] C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice Hall, 1990.

[84] Carroll Morgan, Thai Son Hoang, and Jean-Raymond Abrial. The challenge of probabilistic *vent b* - extended abstract. In Treharne et al. [97], pages 162–171.

[85] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6:319–340, 1976.

[86] A. Papatsaras and B. Stoddart. Global and communicating state machine models in event driven b: A simple railway case study. In D. Bert, J.-P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B - 2nd International Conference of B and Z Users*, volume 2272 of *Lecture Notes in Computer Science*, Grenoble, France, January 2002. Springer Verlag.

[87] M.-L. Potet and Y. Rouzeau. Composition and refinement in the B method. In *B'98: Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[88] R. C. Prim. Shortest connection and some generalizations. *Bell Syst. Tech. J.*, 36, 1957.

[89] project RODIN. Rigorous open development environment for complex systems. http://rodin-b-sharp.sourceforge.net/, 2004. 2004–2007.

[90] H. Jr Rogers. *Theory of Recursive Functions and Effective Computability*. The MIT Press, 1967.

[91] S. Schneider and H. Treharne. Communicating B machines. In D. Bert, J.-P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B - 2nd International Conference of B and Z Users*, volume 2272 of *Lecture Notes in Computer Science*, pages 416–435, Grenoble, France, January 2002. Springer Verlag.

[92] Scientific Computing Associates inc, 246 Church Street, Suite 307 New Haven, CT 06510 USA. *Original LINDA C-Linda Reference manual*, 1990.

[93] E. Sekerinski and K. Sere, editors. *Program Development by Refinement - Cases Studies Using the B Method*. FACIT. Springer Verlag, 1998.

[94] J. M. Spivey. *The Z notation, A Reference Manual*. Prentice Hall, 1989.

[95] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer Verlag, 1998.

[96] H. Treharne and S. Schneider. How to drive a B machine. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *ZB 2000: Formal Specification and Development in Z and B - First International Conference of B and Z Users*, York,UK, August 29 - September 2 2000.

[97] Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider, editors. *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005, Proceedings*, volume 3455 of *Lecture Notes in Computer Science*. Springer, 2005.