Computing and Informatics, Vol. 22, 2003, 221–256

FOUNDATIONS OF THE B METHOD

Dominique Cansell, Dominique Méry

LORIA
BP 239, Campus Scientifique
54506 Vandœuvre-lès-Nancy
France
e-mail: {cansell, mery}@loria.fr

Abstract. 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.

Keywords: Events, actions, systems, refinement, proof, validation, formal method

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 [2]. It is supported by the Atelier B tool [34] and by the B ToolKit [49].

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 [7]. 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 [12, 2, 54] 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* [12, 2, 32, 14]. 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 [34].

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 *glueing 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 tracability 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 [33]. 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 [53].

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 [16, 21, 41] but a wider scope for the method has emerged with the incorporation of the event based approach [1, 11, 6, 7, 28, 26, 59] 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 [11] 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 [16] for controlling train traffic, the PCI protocol [29], the IEEE 1394 Tree Identify Protocol [10]. Finally, B has been combined with CSP for handling communications systems [26, 25] and with action systems [28, 59].

The proposal can be compared to action systems [13], UNITY programs [32] and TLA [43] 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 [60], 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, wheras 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, wheras 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$^+$ [44] 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 [42] 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 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 [38, 23] 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 [61] of Java and the Java Virtual Machine.

1.5 Organization and Reading of the Document

The document is organized with respect to the development process supported by B. It details the event-based B approach and sketches the classical B approach. We introduce some notations and concepts with a simple case study of the *factorial* function. Section 2 presents the mathematics of B and uses the *factorial* function to illustrate how mathematical objects can be carefully described in B. Defined mathematical objects are used later in an abstract specification or model, from which an algorithm is developed using the event-based B approach. Section 3 details the semantics of events and operations and defines the language of actions that are used in B models. Section 4 introduces the B modelling of systems and the different clauses of a B model. Section 5 covers the refinement of B models and details the proof obligations required by the refinement process; we return to our factorial case study to illustrate *proof-based development*. This takes an abstract specification of factorial based on the mathematical definition discussed earlier, then uses an event based approach to derive an algorithm for factorial. In Section 6 we make some concluding remarks.

2 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 [8] develop a *structure language* which allows 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 [37], 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 sections; the complete notation is described in the B book of Abrial [2].

2.1 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 codomain $\mathsf{ran}(r)$. 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 $\mathsf{dom}(f)$.

In Figure 1, set-theoretical notations are summarized 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.

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

Fig. 1. Set-theoretical notations

2.2 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 (e.g. [2, 1, 6, 5, 3, 9, 29, 10]). When considering the definition of a function, we can use different 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; by default, it is supposed to be the least fixed-point. *factorial* is a B *constant* and has B *properties*:

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

In previous work on B [31], 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:

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

(**axioms or B properties**)

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

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

(**axioms or B properties**)

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*.

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

(**consequences or B assertions**)

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 [8]. 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:

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 quantifications 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. [8] 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. [8] or the doctoral thesis of Burdy [24] 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:

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

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

2.3 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
```

Fig. 2.

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 [34] 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**.

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*.

machine
  FACTORIAL_DEF
constants
  $factorial$
properties
  $factorial \in \mathbb{N} \leftrightarrow \mathbb{N} \wedge$
  $0 \mapsto 1 \in factorial \wedge$
  $\forall (n, fn).(n \mapsto fn \in factorial \Rightarrow n{+}1 \mapsto (n{+}1)\cdot fn \in factorial) \wedge$

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

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

3 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 3) contains syntactical structures for expressing different kinds of (states) transitions. Generalized substitutions of GSL allow us to write *operations* in the classical B approach [2]; a restriction over GSL leads to *events* in the so called event-based B approach [11, 7]. In the following subsections, we address the semantical issues of generalized substitutions and the differences between *operations* and *events*.

3.1 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. However, the interpretation we shall give here to this statement is *not* 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**), $S \mathbin{[\!]} T$ (bounded choice) (or **CHOICE** S_1 **OR** S_2 **END**), $P \Rightarrow S$ (guard)(or **SELECT** P **THEN** S **END**),

$@z.S$ (unbounded choice), $x :\in S$ (set choice), $x : R(x_0, x)$ (generalized assignement), $S_1; S_2$ (sequencing), **WHILE** B **DO** S **INVARIANT** J **VARIANT** V **END**, ...

Name	Generalized substitution: S	$[S]P$
Assignment	$x := E$	$P(E/x)$
Skip	*skip*	P
Parallel	$x := E \vert\vert y := F$	$[x, y := E, F]P$
Composition	$x := E \vert\vert y := F$	$[x, y := E, F]P$
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* $G \vert T$)	$G \wedge [T]P$
Generalized Guard	**any** t **where** G **then** T **end**	$\forall t \cdot (G \Rightarrow [T]P)$
Sequential Composition	$S; T$	$[S][T]P$
Iteration	**WHILE** B **DO** T **INVARIANT** J **VARIANT** V **END**	$J \wedge$ $\forall x \cdot (J \wedge B \Rightarrow [T]J) \wedge$ $\forall x \cdot (J \Rightarrow V \in \mathbb{N}) \wedge$ $\forall x \cdot (J \wedge B \Rightarrow [n := V][T](V < n)) \wedge$ $\forall x \cdot (J \wedge \neg B \Rightarrow P)$

Fig. 3. Definition of GSL and $[S]P$

The meaning of a generalized substitution S is defined in the weakest-precondition calculus [35, 36] 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 [35, 36] 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 3 contains the inductive definition of $[S]P$.

We say that two substitutioons 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 [2] 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.

Theorem 1 ([2])**.** For any substitution S, there exist 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 susbtitution 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: $\mathsf{abt}(S) \triangleq$ *for any predicate* $R, \neg[S]R$ and *it defines the set of states that cannot establish any predicate* R *and that are the non-terminating states.*
- Terminating computations: $\mathsf{trm}(S) \triangleq \neg\mathsf{abt}(S)$ and *it defines the termination condition for the substitution* S.
- Miraculous computations: $\mathsf{mir}(S) \triangleq$ *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: $\mathsf{fis}(S) \triangleq \neg\mathsf{mir}(S)$ *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. Figures 4 and 5 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 [2] 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 cannot 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. Figure 6 contains a list of definitions of the predicate with respect to the syntax.

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**	$\forall t \cdot (G \ \Rightarrow \ \mathsf{trm}(T))$

Fig. 4. Examples of definitions for $\mathsf{trm}(S)$

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* $G \| T$)	$G \ \Rightarrow \ \mathsf{fis}(T)$
any t **where** G **then** T **end**	$\exists t \cdot (G \ \wedge \ \mathsf{fis}(T))$

Fig. 5. Examples of definitions for $\mathsf{fis}(S)$

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 4 and Figure 6.

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

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.

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* $G \vert T$)	$G \ \Rightarrow \ \mathsf{prd}_x(T)$
any t **where** G **then** T **end**	$\exists\, t \cdot (\, G \ \wedge \ \mathsf{prd}_x(T)\,)$

Fig. 6. Examples of definitions for $\mathsf{prd}_x(S)$

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 subsections we introduce operations and events, which are two ways to use the B method.

3.2 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 section. However, we should explain the difference between those two notions. An (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:

$$I \wedge \mathsf{trm}(O) \Rightarrow [O]I. \qquad (1)$$

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 *genereous* 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 Figure 7). 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. Figure 8 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 [43] 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, events can get a semantics defined over primed and unprimed variables in Figure 7. The before-after predicate is already defined in the B book as the predicate $\mathsf{prd}_x(S)$ defined for every substitution S (see Subsection 3.1).

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

Fig. 7. Definition of events and before-after predicates of events

Any event e 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. Before introducing B models, we give the expression stating the preservation of a property by a given event e:

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

Fig. 8. Definition of events and guards of events

$$I(x) \;\Rightarrow\; [e]\, I(x) \tag{2}$$

or equivalently in a relational style

$$I(x) \;\wedge\; BA(e)(x,x') \;\Rightarrow\; I(x'). \tag{3}$$

$BA(e)(x,x')$ is the before-after relation of the event e and $I(x)$ is a state predicate over variables x. Equation 1 states the proof obligation of the operation O using the weakest-precondition operator and Equation 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. In the next section, we detail abstract machines and abstract models, which are using operations and events.

4 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 [2, 7] in B contains information for editing, proving, analysing, mapping and exporting models or components. A B component has two separate forms: the first form concerns the development of software models and B components are *abstract machine, refinement, implementation*; the 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 [34, 49] and the second one is partly supported by tools [34]. In the next subsections, we overview each approach based on the same logical and mathematical concepts.

4.1 Modelling Systems in the B Classical Approach

The B method [2] is historically applied to software systems and has helped in developing safe software controling trains [16]. 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.

```
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
```

Fig. 9.

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, s/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 Subsection 4.3. An abstract machine has the 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 the 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.

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* (see 1) 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. Figure 10 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.

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

Fig. 10. An example of an abstract machine for the factorial computation

4.2 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 react 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.

An (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:

```
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
```

Fig. 11.

A model has the 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 the 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 parameterization 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.

	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:

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

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**	$I(x) \;\; \Rightarrow \;\; \exists x' \cdot P(x, x')$
select $G(x)$ **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 Figure 12. As you notice, the abstract machine fac and the abstrcat 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 Figure 10 is passive, but the event *computation* of the model in Figure 12 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 subsection classical mechanisms for structuring developed components of specification.

4.3 Structuring Mechanisms of the B Method

In the last two 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.

4.3.1 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 components, 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

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

Fig. 12. An example of an abstract model for the *factorial* computation

to specific conditions for use. Several authors [57, 15] 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 [55] 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.

4.3.2 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 cannot 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 cannot 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 Figure 13 implements (refines) the machine of Figure 10. 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.

4.3.3 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

```
implementation
  $FACTORIAL\_IMP$
refines
  $FACTORIAL\_MAC$
values
  $m = 5$
concrete_variables
  $result, x$
invariant
  $x \in 0..n \wedge$
  $result = factorial(x)$
assertions
  $factorial(5) = 120 \wedge$
  $result \leq 120$
initialisation
  $result := 1; x := 0$
operations
  $computation =$
                  while $x < m$ do
                        $x := x+1; fn := x \cdot fn$
                  invariant
                        $x \in 0..m$
                        $result = factorial(x)$
                        $result \leq factorial(m)$
                  variant
                        $m - x$
                  end
end
```

Fig. 13. An example of an implementation for the factorial computation

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 [45] illustrates an object-oriented approach of the B development and identifies the layered development paradigm that we have already mentioned through B primitives. 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.

4.3.4 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 [6, 7]. 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 section, we describe the mathematical foundations of B and we illustrate B concepts in the event-based B approach.

4.3.5 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 mechansims support 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 machanisms of classical B do not allow the sharing of variable, but in read mode. Our work on the feature interaction problem [30] 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 [43].

5 PROOF-BASED DEVELOPMENT IN B

5.1 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 the 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 glueing 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 cannot 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 [43] 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 [12], 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.

5.2 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.

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

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

```
progress = select $n \notin \mathsf{dom}(fac)$ then
               any $x$ where
                   $x \in \mathbb{N} \wedge x \in \mathsf{dom}(fac) \wedge x{+}1 \notin \mathsf{dom}(fac)$
               then
                   $fac(x{+}1) := (x{+}1) \times fac(x)$
               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 = select $n \in \mathsf{dom}(fac)$ 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:

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

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:

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

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: the variable i plays the role of index and the variable *fq* is an accumulator. A glueing invariant defines relations between old and new variables:

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

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

```
computation = select $i = n$ then
                      $fn := fq$
              end;
progress = select $i \neq n$ then
                  $i := i{+}1 \parallel fq := (i{+}1) \times fq$
           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
$fn := fq$
```

Case studies provide information on the development process; different domains have been considered for illustrating the event-based B approach: sequential programs [5, 9], distributed systems [3, 10, 29], circuits [4].

6 CONCLUSION

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

6.1 Work on B and with B

The series of conferences [39, 18, 22, 19, 20] on B (in association with the Z community) and books [2, 45, 40, 59] 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:

6.1.1 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 [56] 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 [28] 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 [11] introduce modalities into abstract systems and develop proof obligations related to liveness properties; Méry [51] shows how the B concepts can be easily used to deal with liveness and fairness properties. Bellegarde et al [17] 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 cannot maintain every liveness property.

Recently, McIver et al [50] 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.

6.1.2 Combining B with Another Formalism

The limited expressivity of the B language has inspired work on several proposals. Butler [26] investigates a mixed language including B AMN and CSP; CSP is used to structure abstract machines; the idea is exploited by Schneider and Treharne [62, 58] who control B machines.

Since diagrammatic formalisms offer a visual representation of models, another integration of B with UML is achieved by Butler [27] and by Le Dang et al [47, 46, 48]; 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 [52] 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.

6.2 Final Remarks

The design of (software) systems is an activity based on logic-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 construct safer systems and it provides a realistic framework for developing a pragmatic engineering. Mathematical theories [8] 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 [10]. Tools are already very helpful and will evolve towards a toolset 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. In [7], Abrial describes the new B method mainly related to B events.

Acknowledgements

We are grateful to Stephan Merz and to Bill Stoddart for detailed comments on draft versions; both have kindly suggested improvements of our French English version and have pointed out technical problems. We thank anonymous referees for helpful comments. Thanks to Dines Björner for his encouragements. Merci Dines.

REFERENCES

[1] Abrial, J.-R.: Extending B without Changing It (for Developing Distributed Systems). In H. Habrias, editor, *1st Conference on the B method*, pp. 169–190, November 1996.

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

[3] Abrial, J.-R.: Event Driven Distributed Program Construction. Internal note, Consultant, August 2001. `jr@.abrial.org`.

[4] Abrial, J.-R.: Event Driven Electronic Circuit Construction. Internal note, Consultant, August 2001. `jr@.abrial.org`.

[5] Abrial, J.-R.: Event Driven Sequential Program Construction. Internal note, Consultant, August 2001. `jr@.abrial.org`.

[6] Abrial, J.-R.: Discrete System Models. Internal note, Consultant, February 2002. `jr@.abrial.org`.

[7] Abrial, J.-R.: 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.

[8] Abrial, J.-R.—Cansell, D.—Laffitte, G.: 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, pp. 370–393. Springer Verlag, January 2002.

[9] Abrial, J.-R.—Cansell, D.—Méry, D.: 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.

[10] Abrial, J.-R.—Cansell, D.—D. Méry: A Mechanically Proved and Incremental Development of IEEE 1394 Tree Identify Protocol. Formal Aspects of Computing, Vol. 14, 2003, No. 3.

[11] Abrial, J.-R.—Mussat, L.: 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.

[12] Back, R.: On Correct Refinement of Programs. Journal of Computer and System Sciences, Vol. 23, 1979, No. 1, pp. 49–68.

[13] Back, R.-J.: A Calculus of Refinements for Program Derivations. Acta Informatica, Vol. 25, 1998, pp. 593–624.

[14] Back, R.-J.—von Wright, J.: Refinement Calculus. Springer Verlag, 1998.

[15] Büchi, M.—Back, R.: 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.

[16] Behm, P.—Benoit, P.—Faivre, A.—Meynadier, J.-M.: 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, pp. 369–387, 1999.

[17] Bellegarde, F.—Darlot, C.—Julliand, J.—Kouchnarenko, O.: 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.

[18] Bert, D. 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.

[19] Bert, D.—Bowen, J.-P.—Henson, M. C.—Robinson, K. 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.

[20] Bert, D.—Bowen, J.-P.—King, S.—Waldén, M. 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.

[21] Bicarregui, J.—Clutterbuck, D.—Finnie, G.—Haughton, H.—Lano, K.—Lesan, H.—Marsh, W.—Matthews, B.—Moulding, M.—Newton, A.—Ritchie, B.—Rushton, T.—Scharbach, P.: Formal Methods Into Practise: Case Studies in the Application of the B Method. Internal report, BUT Project, 1995.

[22] Bowen, J. P.—Dunne, S.—Galloway, A.—King, S. 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.

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

[24] Burdy, L.: Traitrement des Expressions Dépourvues de Sens de la Théorie des Ensembles Application à la méthode B. PhD thesis, CNAM, 2000.

[25] Butler, M.: Stepwise Refinement of Communicating Systems. Science of Computer Programming, Vol. 27, 1996, pp. 139–173.

[26] Butler, M.: csp2b: A Practical Approach to Combining csp and b. Formal Aspects of Computing, Vol. 12, 2000, pp. 182–196.

[27] Butler, M.—Snook, C.: 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.

[28] Butler, M.—Walden, M.: Parallel Programming with the B Method. In Program Development by Refinement Cases Studies Using the B Method, Volume 59 of *FACIT*, pp. 183–195. Springer Verlag, 1998.

[29] Cansell, D.—Gopalakrishnan, G.—Jones, M.—Méry, D.—Weinzoepflen, A.: 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.

[30] Cansell, D.—Méry, D.: Abstraction and Refinement of Features. In Ryan Stephen, Gilmore et Mark, editor, Language Constructs for Designing Features. Springer Verlag, 2000.

[31] Cansell, D.—Méry, D.: Développement de Fonctions Définies Récursivement en B: Application du B Événementiel. Rapport de recherche, LORIA UMR 7503, January 2002.

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

[33] Clarke, E. M.—Grumberg, O.—Peled, D. A.: Model Checking. The MIT Press, 2000.

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

[35] Dijkstra, E. W.: A Discipline of Programming. Prentice-Hall, 1976.

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

[37] Ehrig, H.—Mahr, B.: 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.

[38] Gurevitch, Y.: Specification and Validation Methods. Chapter "Evolving Algebras 1993: Lipari Guide", pp. 9–36, Ed. E. Börger, Oxford University Press, 1995.

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

[40] Habrias, H.: Spécification Formelle Avec B. Hermès, 2001.

[41] Hoare, J.: The Use of B in CICS. In *Applications of Formal Methods*, 1995.

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

[43] Lamport, L.: The Temporal Logic of Actions. ACM Transactions on Programming Languages and Systems, Vol. 16, 1994, No. 3, pp. 872–923.

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

[45] Lano, K.: The B Language and Method – A Guide to Practical Formal Development. FACIT. Springer Verlag, 1996.

[46] Ledang, H.–Souquières, J.: Formalizing UML Behavioral Diagrams with B. In Tenth OOPSLA Workshop on Behavioral Semantics: Back to Basics, Tampa Bay, Florida, USA, Oct 2001.

[47] Ledang, H.—Souquières, J.: 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, November 2001.

[48] Ledang, H.—Souquières, J.: 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.

[49] B-Core(UK) Ltd. B-Toolkit User's Manual, relase 3, 2 edition, 1996.

[50] McIver, A.—Morgan, C.—Hoang, T. S.: 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.

[51] Méry, D.: 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.

[52] Mikhailov, L.—Butler, M.: 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.

[53] Moreau, L.: Distributed Directory Service and Message Routing for Mobile Agents. Science of Computer Programming, Vol. 39, 2001, Nos. 2–3, pp. 249–272.

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

[55] Owicki, S.—Gries, D.: An Axiomatic Proof Technique for Parallel Programs I. Acta Informatica, Vol. 6, 1976, pp. 319–340.

[56] Papatsaras, A.—Stoddart, B.: 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.

[57] Potet, M.-L.—Rouzeau, Y.: 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.

[58] Schneider, S.—Treharne, H.: 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, pp. 416–435, Grenoble, France, January 2002, Springer Verlag.

[59] Sekerinski, E.—Sere, K. editors: Program Development by Refinement – Cases Studies Using the B Method. FACIT, Springer Verlag, 1998.

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

[61] Stärk, R.—Schmid, J.—Börger, E.: Java and the Java Virtual Machine. Springer Verlag, 1998.

[62] Treharne, H.—Schneider, S.: 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.

Dominique Cansell is researcher at LORIA (Nancy) and lecturer at the University of Metz in eastern France and has been working at the INRIA research establishment for the last two years. He works in the area of Formal Methods on the use and development of the B method of formal system construction (using refinement) and its associated tool support. In particular he has worked recently with Jean-Raymond Abrial, the originator of the B method, on a new style of proof tool to support formal development. Known as "Click and Prove" this tool simplifies the process of finding the proofs that are used to ensure that software and systems are reliably constructed.

Dominique Méry is professor of computing science at Université Henri Poincaré Nancy 1 and is the scientific leader of the team MOSEL of LORIA laboratory. He works in the area of Formal Methods especially on the proof-based development of software systems. He has worked on temporal logic and proof methods for the verification of concurrent and distribued programs and has developed projects in cooperation with industrial partners in telecommunications and embedded systems. Current interest topics include security and reliability of software-based systems.