Int J Software Informatics, Vol.3, No.2-3, June/September 2009, pp. 197–239 E-mail: ijsi@iscas.ac.cn
International Journal of Software and Informatics, ISSN 1673-7288 http://www.ijsi.org
 Tel: +86-10-62661040

Refinement-Based Guidelines for Algorithmic Systems*

Dominique Méry

(Université Henri Poincaré Nancy 1 & LORIA CNRS UMR 7503, France)

Abstract The *correct-by-construction* approach can be supported by a progressive and incremental process controlled by the refinement of models of programs. We explore the Event B modelling language to illustrate the expression of our methodological proposal using proof-based patterns called guidelines. The main objective is to facilitate the correct-by-construction approach for designing classical sequential algorithms. We address the description of guidelines for the design of programs and algorithms and the integration of proof-based aspects using the RODIN platform. More precisely, we introduce several methodological steps identified during the development of case studies, and we propose auxiliary notions, such as refinement diagrams, for guiding users during problem analysis. A general structure characterizes the relationship between the contract, the Event B, and the developed algorithm using a specific application of Event B models and refinement. We simplify the translation of Event B models into algorithmic elements by promoting the use of recursive constructs. The resulting algorithm is proved to be sound with respect to the pre/post specification, namely, the contract. Applications rely on a dynamic programming technique that illustrates the applicability of these patterns based on a call-as-event relationship. Each proof-based development is validated using the RODIN platform. Our paper contributes to the development of patterns for assisting the proof-based development of algorithmic systems.

Key words: Event B; modelling; refinement; algorithm; pattern; proof; formal method; proof-based development; correct by construction

Méry D. Refinement-Based guidelines for algorithmic systems. *Int J Software Informatics*, **2009, 3(2-3): 197–239. http://www.ijsi.org/1673-7288/3/197.htm**

1 Introduction

1.1 Overview

The development of programs is carried out using either bottom-up techniques or top-down techniques, and our main goal is to develop correct programs from specifications using refinement and proofs. More precisely, we explore the use of *proof-based patterns* for aiding and assisting *programmers* who are prepared to use formal techniques in the development of programs.

* This work is sponsored by grant No. ANR-06-SETI-015-03 awarded by the Agence Nationale de la Recherche.
Corresponding author: Dominique Méry, Email: mery@loria.fr
Manuscript received 2009-02-18; revised 2009-08-10; accepted 2009-08-15.

Program development is supported by Event B models related by semantical relationships that guarantee correctness properties. Modelling is a very challenging task, and its complexity increases when dealing with proof obligations, checking development steps, and refinement steps.

Proof-based patterns are intended to make the programmer's life easier. According to Alexander[10], in the context of architectural problems, *each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.* Gamma et al.[31] introduce *design patterns*, which systematically name, motivate, and explain a general design that addresses a recurring design problem in object-oriented systems. A design pattern describes the problem, the solution, and when to apply the solution, and gives implementation hints and examples. Design patterns structure and organize the development of object-oriented systems and are classified into several classes: creational, structural, and behavioral. Fowler[30] emphasizes the two following modelling principles.

- *Patterns are a starting point, not a destination*: applying a pattern puts you on a safe road to success, but the remaining way is painful; the pattern is not sufficient for constructing a model.
- *Models are not right or wrong; they are more or less useful*: in our case of formal models, it is clear that a model can supply hints for improving the development.

Considering the general problem of correct-by-construction programs (or systems), we limit our scope to those problems solved by sequential algorithms based on paradigms, leaving the case of distributed systems to further work. Our notion of a proof-based pattern integrates models, refinements, and proofs and is based on case studies producing *ideas*. A claim of Fowler[30] is that *a pattern is an idea that has been useful in one practical context and will probably be useful in others*, which shows the informal and imprecise definition of a pattern.

Considering our notion of a proof-based pattern, it remains an imprecise idea, and in the current paper, we are considering the development of an idea summarized mainly by the dual *(procedure) call/event*. Our goals are to illustrate the *call-as-event* principle for sequential algorithms and to provide *guidelines* expressed in the Event B modelling language. The models are Event B models supported by the RODIN environment, which is the reference platform for experimenting with our approach. However, we do not define patterns within an object-oriented framework. Finally, our paper aims to introduce *methodological guidelines* for developing Event B models and for producing programs.

1.2 Correct by construction

Our approach is directly related to the design of correct-by-construction programs. The main idea relies upon the development of *structured* programs following a top-down approach, which is well known from the earlier works of Dijkstra[28,40], and using refinement to control the correctness of the resulting program. It also relies on a more fundamental question related to the notion of the *problem to solve*. Polya[40] describes a set of techniques (or recipes) that can be used to solve problems. The

various steps advocated by Polya can be summarized as follows. The first step is to understand the problem and to list the data, conditions on the data, the unknown elements, and the feasibility of the requirements listed in the statement of the problem. It is clearly important to identify redundancy and possible inconsistencies. Elicitation of the requirements is clearly a very important step and can be driven by following an incremental and progressive methodology based on proof checking. The methodology can be based on *incremental proof-based* development. However, the link between the problem and the first model remains to be expressed, and refinement is a real help in justifying in a very progressive way the choices in design.

When a problem is stated, the incremental proof-based methodology of Event B[19] starts by expressing a very abstract model, with further refinement leading to finer-grain event-based models, which are used to derive an algorithm[4]. The main idea is to consider each *procedure call* as an *abstract event* in a model corresponding to the development of the *procedure*. In general, a procedure is specified by a pre/post specification. Then the refinement process leads to a set of events, which are finally combined to obtain the *body of the procedure*. The refinement process can be considered an *unfolding* of *call* statements under the preservation of invariants and termination. It means that the *abstraction* corresponds to the *terminating procedure call*, and the body is derived using the refinement process. The refinement process may also use recursive procedures, and it supports top-down refinement. The procedure call simulates the abstract event, and the refinement guarantees the correctness of the resulting algorithm.

A preliminary version[21,41,42] introduces the ideas using case studies and provides hints for teaching the design of algorithms using events. Abrial[4] first showed how to derive sequential programs from Event B models, and he used rules for transforming events into structured programs. The rules allow one to decide whether two complementary events (guarded by a boolean condition and its complement) correspond to an *IF* statement or to a *LOOP* statement. In our case, we apply only one rule for an *IF* statement that can be defining a recursive call. Transformations are made simpler and are based on diagrams called *refinement diagrams*, and we base our development on the inductiveness of data structures. Ideas for developing models and for identifying guidelines are borrowed from the analysis of inductive data structures and on programming paradigms such as dynamic programming. Another key aspect of the approach is the strong links with the proof process and proof-based

1.3 Proof-Based development

Proof-based development methods[1,11,40] 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 events. The relationship between two successive models in this sequence is that of *refinement*[1,11]. The essence of the refinement relationship is that it preserves already-proved *system properties* including safety properties and termination.

Development gives rise to a number of *proof obligations*, which guarantee its correctness. Such proof obligations are discharged by a proof tool, using automatic and interactive proof procedures supported by a proof engine[7].

At the most abstract level, the obligation is to describe the static properties of a model's data in terms 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 that are claimed to be invariant are preserved by the events 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, called *gluing invariants*, are used in the formulation of the refinement proof obligations.

The goal of an Event B development is to obtain a *proved model* and to implement the correct-by-construction[36] paradigm. Because 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 for logical theories, and we must distribute the complexity of proofs over the components of the current development, such as by refinement. Refinement has the potential to decrease the complexity of the proof process while allowing for the traceability of requirements.

Models based on B rarely need to make assumptions about the *size* of a system being modelled, such as the number of nodes in a network. This is in contrast to model-checking approaches[26]. The price is facing possibly complex mathematical theories and difficult proofs. The re-use of developed models and the structuring mechanisms available in B help to decrease the complexity. These solutions are, de facto, supporting the use of patterns that state general proof-based developments and are validating the expression of proof-based patterns.

1.4 Patterns for proof-based development

Patterns and design patterns[31] are very convenient aids to the design of object-oriented software. Originally, they were borrowed from architectural practices[10]. Recently, Abrial[6] suggested the introduction of a kind of pattern for proof-based development. Action/reaction patterns have been applied to a press case study by Abrial, and they improved the proof process. Another pattern, called a *re-usability pattern*, has been suggested by Abrial and by Cansell and Méry[8,20] and applied to the development of voting systems[18] and greedy algorithms[20]. In the context of time-sensitive systems[23], Rehm et al.[24] have identified patterns for introducing time in Event B models and for expressing time constraints in Event B models of the IEEE 1394 tree-identification protocol. Dealing with access-control-based problems, Benaissa et al.[13] have proposed a general pattern for handling access control policies and for integrating access control policies into systems. Clearly, activity in pattern modelling is growing and is addressing many kinds of case studies and domains of problems. No classification has yet been given, and there is no real repository of patterns validated for a specific modelling language based on proof-based transformations.

Important aspects of using patterns are evaluation and relevance to the various types of problems. The evaluation for current identified patterns depends upon several indicators, and this should be accepted. In our case, we have experimented with a plug-in for implementing the call-as-event principle. Our goal is to provide a language of patterns that is simple to use and that is supported by a plug-in for the RODIN[45]

environment. Finally, proof-based patterns should be robust with respect to the semantics of the languages used.

1.5 Organization of the paper

Section 2 introduces the modelling language called Event B. It introduces definitions for event, refinement, and model, and their corresponding proof obligations. Section 3 describes how problems can be stated in the Event B environment and provides a general methodology for designing a correct-by-construction algorithm using our guidelines. Refinement diagrams are defined, and we explain why they are useful for the systematic construction of the algorithm from a refinement model. We show how guidelines and structures organize the proof-based development of Event B models, with a simple case study illustrating the various guidelines. Section 4 contains non-trivial examples that use our framework based on the *call-as-event* relationship, namely, primitive recursive functions and Floyd's algorithm. We conclude our work in the final section.

Table 1 Event B B events and proof obligations

Event e	Before-after Predicate BA(e)(x,x')
BEGIN $x :\mid (P(x, x'))$ END	$P(x, x')$
WHEN $G(x)$ THEN $x :\mid (Q(x, x'))$ END	$G(x) \wedge Q(x, x')$
ANY t WHEN t WHERE $G(t, x)$ THEN $x :\mid (R(x, x', t))$ END	$\exists t \cdot (G(t, x) \wedge R(x, x', t))$

PROOF OBLIGATIONS

(INV1) $Init(x) \Rightarrow I(x)$

(INV2) $I(x) \wedge BA(e)(x, x') \Rightarrow I(x')$

(FIS) $I(x) \wedge \mathsf{grd}(e)(x) \Rightarrow \exists y.BA(e)(x, y)$

2 The Modelling Framework

Here, we will summarize the concepts of the Event modelling language developed by Abrial[2,6,19] and will indicate the links with the tool called RODIN[45]. We will also reorganize the Event B method to fit our problem, where the main goal is to use Event B without change and to add guidelines for the problem we want to tackle.

2.1 Modelling actions over states

The event-driven approach[3,6,19] is based on the B notation. It extends the methodological scope of basic concepts to take into account the idea of *formal models*. Briefly, a formal model is characterized by a (finite) list x of *state variables* possibly modified by a (finite) list of *events*, where an invariant $I(x)$ states properties that must always be satisfied by the variables x and *maintained* by the activation of the

events. In the following, we summarize the definitions and principles of formal models and explain how they can be managed by tools[7,27,45].

Generalized substitutions are borrowed from the B notation. They provide a means to express changes to state variable values. In its simple form $x := E(x)$, a generalized substitution looks like an assignment statement. In this construct, x denotes a vector built on the set of state variables of the model, and $E(x)$ denotes a vector of expressions. Here, however, the interpretation we shall give to this statement is not that of an assignment statement. We interpret it as a *logical simultaneous substitution* of each variable of the vector x by the corresponding expression of the vector $E(x)$. There exists a more general normal form of this, denoted by the construct $x : |(P(x, x'))$. This should be read as x *is modified in such a way that the value of* x *afterwards, denoted by* x'*, satisfies the predicate* $P(x, x')$, where x' denotes the *new value* of the vector and x denotes its *old value*. This is clearly nondeterministic in general.

An event has two main parts, namely, a *guard*, which is a predicate built on the state variables, and an *action*, which is a generalized substitution. An event can take one of three normal forms. The first form (BEGIN $x : |(P(x, x')$ END) shows an event that is not guarded, being therefore always enabled and semantically defined by $P(x, x')$. The second form (WHEN $G(x)$ THEN $x : |(Q(x, x'))$ END) and third form (ANY t WHERE $G(t, x)$ THEN $x : |(R(x, x', t))$ END) are guarded by a guard that states the necessary condition for these events to occur. The guard is represented by WHEN $G(x)$ in the second form, and by ANY t $WHERE$ $G(t, x)$ (for $\exists\, t \cdot G(t, x)$) in the third form. We note that the third form defines a possibly nondeterministic event where t represents a vector of distinct local variables. The *before–after* predicate $BA(x, x')$, associated with each of the three event types, describes the event as a logical predicate expressing the relationship linking the values of the state variables just before (x) and just after (x') the *execution* of EVENT evt. The second and the third forms are semantically equivalent to $G(x) \wedge Q(x, x')$ resp. $\exists\, t \cdot (G(t, x) \wedge R(x, x', t)$. Table 1 summarizes the three possible forms for writing a B event.

Proof obligations (INV 1 and INV 2) are produced by the RODIN tool[45] from events to state that an invariant condition $I(x)$ is preserved. Their general form follows immediately from the definition of the before–after predicate $BA(e)(x, x')$ of each event e (see Table 1). Note that it follows from the two guarded forms of the events that this obligation is trivially discharged when the guard of the event is false. Whenever this is the case, the event is said to be *disabled*. The proof obligation FIS expresses the feasibility of the event e with respect to the invariant I.

2.2 Model refinement

The refinement of a formal model allows us to enrich the model via a *step-by-step* approach and is the foundation of our correct-by-construction approach[36]. Refinement provides a way to strengthen invariants and to add details to a model. It is also used to transform an abstract model to a more concrete version by modifying the state description. This is done by extending the list of state variables (possibly suppressing some of them), by refining each abstract event to a corresponding concrete version, and by adding new events. The abstract (x) and concrete (y) state variables are linked by means of a *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 takes control for ever, and (4) relative deadlock freedom is preserved. Details of the formulation of these proofs follows.

We suppose that an abstract model AM with variables x and invariant $I(x)$ is refined by a concrete model CM with variables y and gluing invariant $J(x,y)$. If $BA(e)(x,x')$ and $BA(f)(y,y')$ are respectively the abstract and concrete before–after predicates of the same event, e and f respectively, we have to prove the following statement, corresponding to proof obligation (1).

$$I(x) \wedge J(x,y) \wedge BA(f)(y,y') \;\Rightarrow\; \exists x' \cdot (BA(e)(x,x') \wedge J(x',y'))$$

Now, proof obligation (2) states that $BA(f)(y,y')$ must refine *skip* ($x' = x$), generating the following simple statement to prove (2).

$$I(x) \wedge J(x,y) \wedge BA(f)(y,y') \;\Rightarrow\; J(x,y')$$

In refining a model, an existing event can be refined by strengthening the guard and/or the before–after predicate (effectively reducing the degree of nondeterminism), or a new event can be added to refine the skip event. The feasibility condition is crucial to avoiding possible states that have no successor, such as division by zero. Furthermore, this refinement guarantees that the set of traces of the refined model contains (up to stuttering) the traces of the resulting model.

The Event B modelling language is supported by the RODIN platform[45] and has been introduced in publications[6,19], where there are many case studies and discussions about the language itself and the foundations of the Event B approach. The language of *generalized substitutions* is very rich, enabling the expression of any relation between states in a set-theoretical context. The expressive power of the language leads to a requirement for help in writing relational specifications, which is why we should provide guidelines for assisting the development of Event B models.

2.3 Structures for Event B models

The Event B modelling language provides a framework for supporting our methodology as applied to the development of sequential programs. Abrial[5] has demonstrated the possibility of developing sequential programs using Event B. The modelling process deals with various languages, as seen by considering the triptych of Bjoerner[14–17]: $\mathcal{D}, \mathcal{S} \longrightarrow \mathcal{R}$. Here, the domain $\mathcal{D}$ deals with properties, axioms, sets, constants, functions, relations, and theories. The system model $\mathcal{S}$ expresses a model or a refinement-based chain of models of the system. Finally, $\mathcal{R}$ expresses requirements for the system to be designed. Considering the Event B modelling language, we notice that the language can express *safety* properties, which are either *invariants* or *theorems* in a machine corresponding to the system. Recall that two main structures are available in Event B.

- Contexts express static information about the model.

- Machines express dynamic information about the model, invariants, safety properties, and events.

2.3.1 Contexts

The first structure is called a context (see Fig. 1), and it provides the definition of the sets, constants, axioms for sets and constants, and theorems that can be derived from the axioms of the context $\mathcal{D}$. The context $\mathcal{AD}$ is a previous context that has already been defined, and it extends the current context. A context is validated when sets $S_1, \ldots, S_n$, constants $C_1, \ldots, C_m$, and axioms $ax_1, \ldots, ax_p$ are well formed and when all theorems $th_1, \ldots, th_q$ are proved.

CONTEXT $\mathcal{D}$
EXTENDS $\mathcal{AD}$
SETS
$S_1, \ldots S_n$
CONSTANTS
$C_1, \ldots, C_m$
AXIOMS
$ax_1 : P_1(S_1, \ldots S_n, C_1, \ldots, C_m)$
$\ldots$
$ax_p : P_p(S_1, \ldots S_n, C_1, \ldots, C_m)$
THEOREMS
$th_1 : Q_1(S_1, \ldots S_n, C_1, \ldots, C_m)$
$\ldots$
$th_q : Q_q(S_1, \ldots S_n, C_1, \ldots, C_m)$

MACHINE $\mathcal{M}$
REFINES $\mathcal{AM}$
SEES $\mathcal{D}$
VARIABLES x
INVARIANTS
$inv_1 : I_1(x, S_1, \ldots S_n, C_1, \ldots, C_m)$
$\ldots$
$inv_r : I_r(x, S_1, \ldots S_n, C_1, \ldots, C_m)$
THEOREMS
$th_1 : SAFE_1(x, S_1, \ldots S_n, C_1, \ldots, C_m)$
$\ldots$
$th_s : SAFE_s(x, S_1, \ldots S_n, C_1, \ldots, C_m)$
EVENTS
EVENT initialisation
BEGIN
$x :|(P(x'))$
END
$\ldots$
EVENT e
ANY t
WHERE
$G(x, t)$
THEN
$x :|(P(x, x', t))$
END
$\ldots$
END

Figure 1. Context and machine

A context clearly states the static properties of the (system) model under construction. The *extends* construct enables re-use by extending a previously defined context.

The proof process is based on the management of sequents, with an associated environment for proof called $\Gamma(\mathcal{D})$. The proof environment includes axioms, prop-

erties, and theorems already proved. An environment is initially provided, but the intention is to add new theorems. This means that we intend to prove the following properties in the sequent calculus style:

$$\text{for any } j \text{ in } \{1..q\},\ \Gamma(\mathcal{D}) \vdash th_j : Q_j(S_1, \ldots, S_n, C_1, \ldots, C_m).$$

Theorems for the context are proved using the RODIN tool, but it is clear that the process for constructing the domain $\mathcal{D}$ is crucial to modelling the system, from consideration of the triptych of Bjoerner[14–17] and variations of this methodology.

The possibility of re-using former definitions is crucial, but we do not consider this point in this paper. Instead, we *simulate* the re-use of theories by manipulating the contexts directly. Among the requirements, we can list the theorems of the context, and we can, in fact, interpret the triptych as follows: for any j in $\{1..q\}$, $\mathcal{D} \longrightarrow th_j : Q_j(S_1, \ldots S_n, C_1, \ldots, C_m)$. Here, it appears that the system is not mentioned, and this is the case for static properties. Therefore, we have an interpretation of the triptych for the static information, which can be re-used later for any system.

2.3.2 Machines

The dynamic part of a model is expressed using the notion of the *machine* (see Fig. 1). A machine is either a basic machine or a refinement of a more abstract machine. A machine models a state via a list of variables x that are assumed to be modifiable by events listed in the machine. The view is assumed to be closed with respect to events. Each event maintains an assertion called an *invariant*, which is a conjunction of logical statements called inv_j. Each reached state satisfies properties of the theorem part called safety properties. Proof obligations are given in the last section, and they are generated and checkable by the RODIN framework. The validation of the machine M leads to the validation of the safety and invariance properties.

We can obtain a variation of the triptych ($\Gamma(\mathcal{D}, M)$ is an associated environment for proof) as follows.

- For any j in $\{1..r\}$,
  $\Gamma(\mathcal{D}, M) \vdash INITIALISATION(x') \Rightarrow I_j(x', S_1, \ldots, S_n, C_1, \ldots, C_m)$
- For any j in $\{1..r\}$, for any event e of M,
  $$\Gamma(\mathcal{D}, M) \vdash \left(\bigwedge_{j \in \{1..r\}} I_j(x, S_1, \ldots, S_n, C_1, \ldots, C_m) \right) \wedge BA(e)(x, x')$$
  $$\Rightarrow I_j(x', S_1, \ldots, S_n, C_1, \ldots, C_m)$$
- For any k in $\{1..s\}$,
  $$\Gamma(\mathcal{D}, M) \vdash \left(\bigwedge_{j \in \{1..r\}} I_j(x, S_1, \ldots, S_n, C_1, \ldots, C_m) \right)$$
  $$\Rightarrow SAFE_k(x, S_1, \ldots, S_n, C_1, \ldots, C_m)$$

We use temporal operators for expressing the safety and invariant properties.

- For any j in $\{1..r\}$, $\mathcal{D}, M \longrightarrow \Box I_j(x, S_1, \ldots, S_n, C_1, \ldots, C_m)$.
- For any k in $\{1..s\}$, $\mathcal{D}, M \longrightarrow \Box SAFE_k(x, S_1, \ldots, S_n, C_1, \ldots, C_m)$.

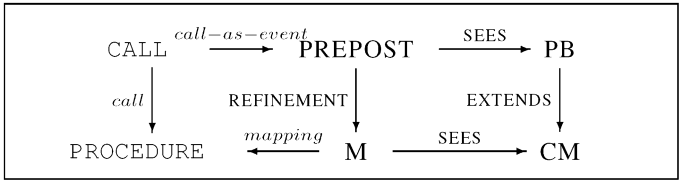
We summarize the requirements expressed by the machine M as follows.

$$\mathcal{D}, M \longrightarrow \Box \left(\begin{array}{l} \left(\bigwedge_{j \in \{1..r\}} I_j(x, S_1, \ldots, S_n, C_1, \ldots, C_m) \right) \\ \left(\bigwedge_{k \in \{1..s\}} SAFE_k(x, S_1, \ldots, S_n, C_1, \ldots, C_m) \right) \end{array} \right)$$

We will use the notation $\mathcal{I}(M)$ to stand for the invariant of the machine M and $\mathcal{SAFE}(M)$ to stand for the safety properties of the machine M. We have shown that requirements $\mathcal{R}$ are first expressed using the *always* temporal operator. To specify total correctness properties, we should extend the scope of the requirements language by adding *eventuality* properties. Eventuality properties will be defined in the next section and will be specific to our methodology.

3 Sequential Programming from Event B Models

This section introduces guidelines as proof-based constructs, and we will illustrate the application of guidelines in the next section. Our guidelines consider the *programming-from-specification*[40] principle in the Event B context. They are not simply taken from of the works of Morgan and Abrial[4], but they suggest a simpler way to program specifications stated as Event B models. We reformulate the programming-from-specification principle in a simpler way . The call-as-event principle is stated by the following diagram:



- `CALL` is the call of the `PROCEDURE`; we assume that `CALL` provides pre- and postconditions and define the problem to be solved by a sequential algorithm stated by `PROCEDURE`.
- PREPOST is the machine containing the events stating the pre- and postconditions of `CALL` and `PROCEDURE`, and M is the refinement machine of PREPOST, with events including control points defined in CM. CM extends the context PB, which expresses the problem to be solved in a mathematical way.
- The *call-as-event* transformation produces a model PREPOST and a context PB from `CALL`.
- The *mapping* transformation allows us to derive an algorithmic procedure that can be mechanized. A plug-in is developed that can be used to derive the algorithm from the component M.
- `PROCEDURE` is a node corresponding to a procedure derived from the refinement model M. `CALL` is an instantiation of `PROCEDURE` using parameters x and y.
- M is a refinement model of PREPOST, which is transformed into `PROCEDURE` by applying structuring rules. It may contain events corresponding to calls of other procedures.

The diagram summarizes the relationship between programming objects and modelling objects, where the *call-as-event* transformation relates the procedure call to a list of events, and the mapping transformation produces the procedure body. Abrial[4] proposes a list of transformation rules for producing sequential algorithms from Event B models, and the idea is to reduce two events, guarded by a condition and its complement, to a conditional statement or to a loop statement. Our diagram is based on well-defined relationships such as *refines* and *sees*, which are checked by discharging the proof obligations generated by the RODIN platform and by justifying the relationship between two languages, namely, a programming language and a modelling language. We now provide details about how to construct such a diagram.

3.1 Defining PB and PREPOST from `CALL`

Let the problem to be solved PB be stated by the pre/post specification of a procedure called `PROC`. The procedure can be called according to precondition P. y represents *call-by-reference* parameters, and x represents *call-by-value* parameters.

PROCEDURE `PROC` $(x;$ VAR $y)$
PRECONDITION $P(x)$
POSTCONDITION $Q(x,y)$

The modelling process starts by identifying the domain of the problem and is expressed using the concept of CONTEXT. We are using the notation $\tilde{R}$ to define the set of values satisfying R. The CONTEXT PB (see Fig. 2) states the theoretical notions required to express the problem statement in a formal way. The CONTEXT PB declares the following.

- Two domains D_1 and D_2 that define the global set of possible values of the current system for its inputs and outputs.
- A list of constants x that specify the input of the system under development $\tilde{P}$, which is the set of values for x defining the precondition, and $\tilde{Q}$, which is a binary relation over $D_1 \times D_2$ defining the postcondition for the problem.
- A list of axioms that assigns types to constants and adds knowledge to the RODIN environment.
- The theorems that state the existence of solutions with respect to preconditions: for instance, theorem $th1.i$ states that there is always a solution y when the input value x satisfies the precondition P.

Theorems are discharged using the proof assistant of the RODIN tool. The underlying language is a set-theoretical language. When an expression E is given, a well-definedness condition is generated by the tool, which enables us to check that some side conditions are true. For instance, the expression $f(x)$ generates a condition $x \in dom(f)$. We formulate our first guideline as follows.

Guideline 1 PB
Problem analysis produces a CONTEXT PB, which includes sets, constants, axioms, and theorems related to the problem. Proof obligations express the well-definedness

of constants and axioms, and the existence of solutions for PB. Call-by-value parameters x are declared as constants, since the value of x is supposed to be given for the complete processing. The CONTEXT PB should be consistent.

CONTEXT PB

SETS

D_1, D_2

CONSTANTS

$x, n, \widetilde{P}, \widetilde{Q}$

AXIOMS

$axm1 : x \in D_1$

x belongs to a general set of the problem domain

$axm2 : n \in NAT1$

n is the number of different possible preconditions, n is not equal to 0 and it is generally 1

$axm3 : \widetilde{P} \in 1..n \longrightarrow \mathbb{P}(D_1)$

$\widetilde{P}$ *is a collection of sets defining the precondition*

$axm4 : \widetilde{Q} \subseteq D_1 \times D_2$

$\widetilde{Q}$ *is a binary relation over* $Q_1 \times Q_2$ *defining the postcondition*

$axm5 : x \in \bigcup_{i \in 1..n} \widetilde{P}(i)$

x is supposed to satisfy the precondition $P(i)$

THEOREMS

$th1.i : \forall a \cdot a \in \widetilde{P}(i) \Rightarrow (\exists b \cdot a \mapsto b \in \widetilde{Q})$

there is at least one solution for each data x satisfying the precondition P_i

END

Figure 2. Context for modeling the problem PB

Now, we consider the *dynamic* part of the problem to be solved. The pre/post specification states that y is the result of the computation of the procedure PROC. The precondition P is defined as the exclusive disjunction of predicates $P_1, \ldots, P_n$ with predicate P_i $(i \in 1..n)$ being attached to case i for the call. The theorem $th.i$ justifies the existence of a value y, when x satisfies P_i. We define a list of events corresponding to each possible call instance EVENT call_1, EVENT call_2, . . . , EVENT call_n, which express the various cases identified by the preconditions.

For each call CALL$_i$, y is set to a value satisfying $Q(x, y)$ when x satisfies $P(x)$. We consider that variable y is initialized by a value satisfying the assertion $Init(y, x, D_1, D_2)$. We can write the machine PREPOST, which sees and uses the data defined in the CONTEXT PB and which encapsulates the n cases corresponding to the n preconditions. Fig. 3 describes the machine PREPOST. The invariant is simply typing the variable y.

EVENT $\texttt{call}_i$
  WHEN
    $P_i(x)$
  THEN
    $y :|(Q(x, y'))$
  END

Guideline 2 PREPOST

Each case specified by a precondition is translated into a specific event defining the computed value according to the postcondition. Call-by-reference parameters are defined as variables of the Event B machine, and call-by-value parameters are defined as constants of the Event B machine.

MACHINE PREPOST
SEES PB
VARIABLES
  y
INVARIANTS
  $inv1 : y \in D_2$
EVENTS
INITIALISATION
  BEGIN
    $act1 : y :\in D_2$
  END
...
EVENT call_i
  WHEN
    $P_i(x)$
  THEN
    $y :|(Q(x, y'))$
  END
...
END

Figure 3. Machine PREPOST

The current status of the development can be represented as follows:

Proof obligations of PREPOST are very simple to discharge, but proofs of theorems may be more tedious. The statement of a given problem in the Event B modelling language is relatively direct, provided we are able to express the underlying mathematical theory using the mechanism of contexts. The existence of a solution y for each value x is proved by deriving the property as a theorem. This means that we should develop a way to validate axioms to ensure the consistency of the underlying theory.

The Event B modelling language allows us to express safety properties, and the machines define reactive systems. According to Abrial, there is no execution of Event B machines: a machine is valid with respect to a set of discharged proof obligations. Because we have a list of events for each machine, we can simulate reactions to events by extending the semantical scope of Event B properties. The definition of *liveness* properties requires the definition of traces for an Event B machine in an operational style. Although proof obligations of refinement integrate elements that take into account the introduction of new events, which can take control from the previous abstract events, we prefer to support our proofs by the TLA[33] framework, which integrates simple temporal modalities such as liveness or fairness.

Definition 1. *Let M be an Event B machine and C a context seen by M. Let y be the list of variables of M, let E be the set of events of M, and let $Init(y)$ be the predicate defining the initial values of y in M. The temporal framework of M is defined by the TLA specification denoted $SpecM$:*

$$Init(y) \wedge \Box[Next]_y \wedge WF_y(Next), \text{where} Next \equiv \exists e \in E.BA(e)(y, y').$$

Following Lamport[33,34], $SpecM$ is valid for the set of infinite traces simulating M with respect to the events of M and to fairness constraints. The set of traces for M is a subset of $Values^{\omega}$, which is the set of infinite words over the set of possible values of y in M, namely, *Values*. Liveness properties for M are, de facto, defined in TLA as follows.

M satisfies $P \leadsto Q$ when $\Gamma(M) \vdash SpecM \Rightarrow (P \leadsto Q)$. $\Gamma(M)$ is the proof context of M. Obviously, safety properties can be reformulated in the same framework. Considering liveness properties, we can also use the wp-based approach for defining the liveness properties under weak fairness. Abrial and Mussat[9] introduce specific constructs to state liveness properties as events, and they define mathematical semantics in a *wp* framework[25,37,38]. Now we can derive the following theorem.

Theorem 1. Suppose that PB is a context and PREPOST is a machine corresponding to a problem stating calls of a procedure. Suppose that the following diagram is validated:

We assume that the preconditions are defined by P, i.e., $P_1, \ldots, P_n$, and the postcondition is defined by Q. Then for any i in $1..n$, PREPOST satisfies $P_i(x, y) \leadsto Q(x, y)$.

Proof:

Suppose that PB is a context and PREPOST is a machine corresponding to a problem stating calls of a procedure. Suppose that both PREPOST and PB are validated: proof obligations are generated and are proved to be correct. This means that the following property holds for any i in $1..n$: $\forall x, y, y'.P_i(x) \wedge y \in D_2 \wedge BA(call_i)(y, y') \Rightarrow Q(x, y')$, which can be simply rewritten as $P_i \wedge [Next]_{<x,y>} \Rightarrow (P_i' \vee Q)$, $P_i \wedge \langle Next\rangle_{<x,y>} \Rightarrow Q'$ and $P \Rightarrow$Enabled $\langle Next\rangle_{<x,y>}$, where Enabled $\langle Next\rangle_{<x,y>}$ is defined by $\exists y'.Next(y, y')$ and $\exists y'.BA(call_i)(y, y') \Rightarrow \exists y'.Next(y, y')$: the condition of feasibility ensures that the call is terminating and it will ensure the final termination of the refinement machine. Applying the rule WF1 of the TLA logic gives $\Box[Next]_{<x,y>} \wedge \mathrm{WF}_y(Next) \Rightarrow (P_i \rightsquigarrow Q)$. Finally, we obtain $Init(x, y) \wedge \Box[Next]_{<x,y>} \wedge \mathrm{WF}_y(Next) \Rightarrow (P_i \rightsquigarrow Q)$. This is a very simple and strict application of the definition of the event $call_i$ and the definition of the $\rightsquigarrow$ properties. □

Therefore, the total correctness of the procedure is obtained easily, thanks to events, and it is discharged either automatically or interactively. The event-based expression is very similar to the expression of HOARE logic triples. Questions about the termination and then about the total correctness are also addressed in the following way. An abstract model PREPOST expresses what the various possibilities are, when calling the procedure. Moreover, the goal is to call successfully, which leads to guarding the event by a condition called the *precondition* in other languages, such as EIFFEL[39], JML[35], and Spec#[12]. The point is to interpret the call in the Event B framework as a guarded event, which is a defensive way of modelling the idea. Here, the feasibility condition states that, under the current environment, there is a possibility of termination of the CALL events, and we should keep this point as a condition of termination. The guard intends to capture conditions for the termination of the event, and Event B provides three simple constructs, thanks to Abrial, which are considered as *waiting entities*. We will see later that the issue is to produce an algorithm corresponding to the call, and conditions for termination should be clearly stated in the models.

3.2 Refining PREPOST into M

We have seen that we have obtained the expression of the pre/post specification by machine events. The next step is to *unfold* the events in order to reduce the nondeterminism. The idea is quite simple: using refinement reduces the nondeterminism by introducing a specific variable for the control of the algorithm. We extend the previous diagram by adding a new Event B machine. We use refinement for *unfolding events* considered as procedure calls, and based on refinement and proof, for deriving structured programs. The main goal is to introduce a way of controlling events by adding a control variable c. Possible values for the variable c are defined in the context CM that extends the context PB. The issue is to identify the various control points of the model M. We have used diagrams to help to decompose each call of the model PREPOST into several steps, corresponding to steps leading to the computation.

These diagrams are very similar to the proof lattices introduced by Owicki and Lamport[43] for representing proofs of liveness properties under fairness assumptions. They are used not for proving but for refining, and we call them *refinement diagrams*. We construct our refinement diagrams by applying the inference rules for the temporal

operators $\rightsquigarrow$. We do not consider induction-based rules in our case, because induction-related questions are hidden by the choice of an inductive data structure for solving the problem. For instance, the dynamic programming paradigm provides a framework based on a function for solving a given problem, and the main issue is stating carefully the function in the context PB. First we define our *refinement diagrams.*

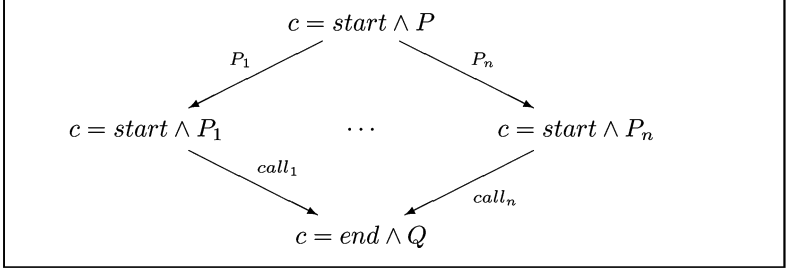
Definition 2. *Let L be a set of elements called control points, containing at least* start *and* end*. Let c be a variable of M called the control variable with values ranging across L. Let A be a finite set of assertions for M of the form $c = l \wedge P(x)$, where $l \in L$ and x are variables of M. Let G be a finite set of assertions for M called conditions of the form $g(c, x)$, where $c \in L$ and x are the variables of M. Let E be the set of events for M. We assume that P is a precondition of the form $c = start \wedge A(x)$, that it can be decomposed into n preconditions P_i, and that Q is a postcondition of the form $c = end \wedge B(x)$. ℓ is a one-to-one function assigning a unique label to each assertion of A and $I(M)$ is the invariant of* M.

A refinement diagram for M, P, and Q over L and A is an acyclic labelled graph over A with labels from G or E satisfying the following rules.

- There is a unique input node P with at least one outgoing arrow.
- There is a unique output node Q with no outgoing arrows.
- If R is related to S by a unique arrow labelled $e \in E$, then
  - It satisfies the property $R \rightsquigarrow S$
  - $\forall c, x, c', x'.R(c, x) \wedge I(M)(c, x) \wedge BA(e)(c, x, c', x') \Rightarrow S(c', x')$
  - $\forall c, x.R(c, x) \wedge I(M)(c, x) \Rightarrow \exists c', x'.BA(e)(c, x, c', x')$
  - If $R \equiv c = l1 \wedge A(x)$ and $S \equiv c = l2 \wedge B(x)$, then $l1 \neq l2$ and $\ell\mathrm{R} = l1$, $\ell(\mathrm{S}) = l2$.
- If R is related to $S_1, \ldots, S_p$, then
  - Each arrow R to S_i is labelled by a guard $g_i \in G$.
  - For any i in $1..p$ the following conditions hold.
    $$\begin{cases} R \wedge I(M) \wedge g_i(x) \Rightarrow S_i \\ \forall j.j \in 1..p \wedge j \neq i \wedge R \wedge I(M) \wedge g_i(x) \Rightarrow \neg g_j(x) \end{cases}$$
  - $R \wedge I(M) \Rightarrow \exists i \in 1..p.g_i$.
- For each $e \in E$, there is only one instance of e in the diagram.

A refinement diagram D for M, P, and Q over L and A is denoted by PDM $= (A, L, M, P, Q, G, E)$. We use $PRE(D)$ for P and $POST(D)$ for Q.

As an example, we can construct the following refinement diagram for a machine RPREPOST that refines PREPOST by adding $L = \{start, end\}$ and a variable $c \in L$.



The refinement diagram is very similar to the proof lattice proposed by Owicki and Lamport and to predicate diagrams[22], and it can be used to infer the total correctness of the resulting algorithm. In fact, we consider that each call terminates. It can be considered as an abstract way of executing the program under construction. The acyclicity provides a simplification by deriving the algorithm from the nodes. We will see that this property is crucial for making the translation into a programming language easier, with the goal being to avoid links that create cycles. Recursive development helps to achieve this property. The operator $\rightsquigarrow$ is transitive and confluent. Therefore, if a refinement diagram is built for a given problem, it is sound with respect to the requirements of the problem. Assertions of the refinement diagram contain control assertions defining steps of the algorithmic process.

Lemma 1. Let M be a machine and let $D = (A, C, M, P, Q, G, E)$ be a refinement diagram for M.

1. If M satisfies $P \rightsquigarrow Q$ and $Q \rightsquigarrow R$, it satisfies $P \rightsquigarrow R$.
2. If M satisfies $P \rightsquigarrow Q$ and $R \rightsquigarrow Q$, it satisfies $(P \vee R) \rightsquigarrow Q$.
3. If I is invariant for M and if M satisfies $P \wedge I \rightsquigarrow Q$, then M satisfies $P \rightsquigarrow Q$.
4. If I is invariant for M and if M satisfies $P \wedge I \Rightarrow Q$, then M satisfies $P \rightsquigarrow Q$.

The $\rightsquigarrow$ relation is transitive and confluent. These properties are derived from the definitions in TLA.

Lemma 2. Let M be a machine and let $D = (A, C, M, P, Q, G, E)$ be a refinement diagram for M. If $P \xrightarrow{e} Q$ is a link of D for the machine M, then M satisfies $P \rightsquigarrow Q$.

Proof: By the rules of construction of D for M, and rule WF1 of TLA, M satisfies the property $P \rightsquigarrow Q$. □

Lemma 3. Let M be a machine, and let $D = (A, C, M, P, Q, G, E)$ be a refinement diagram for M. If P and Q are two nodes of D such that there is a path in D from P to Q and any path from P can be extended in a path containing Q, then M satisfies $P \rightsquigarrow Q$.

Proof: We proceed by induction on the path from P to Q.
⟨1⟩1. The length of path from P to Q is 1.

Proof: By the second constraint on the path from P, there is only one possible successor from P, namely, Q. Either the label is a guard, or the label is an event e. □

⟨2⟩1. The label is a guard g.

Proof: From the definition of the refinement diagram D, we derive the valid property $P \wedge I(M) \wedge g(x) \Rightarrow Q$ and $P \wedge I(M) \Rightarrow g(x)$. Therefore, $P \wedge I(M) \Rightarrow Q$. From the previous lemma, we derive that M satisfies $P \rightsquigarrow Q$. □
⟨2⟩2. The label is an event e.

Proof: From the construction of D, M satisfies the property $P \rightsquigarrow Q$. □
⟨2⟩3. Q.E.D.
Proof: From cases ⟨2⟩1 and ⟨2⟩2, M satisfies the property $P \rightsquigarrow Q$. □
⟨1⟩2. The length of path from P to Q is n, and we assume that the property to be proved holds for paths of length $n - 1$.

Proof: Because the path from P to Q is $n > 0$, either there exists an assertion R and a link from P to R labelled by an event e, or there are assertions $R_1, \ldots R_k$ and guards $g_1, \ldots, g_k$ such that links $P \xrightarrow{g_i} R_i$ exist for any $i \in 1..k$. □
⟨2⟩1. M satisfies $P \rightsquigarrow S$, where S is either $\bigvee_{i \in 1..k} R_i$ or R.
Proof:
⟨3⟩1. There exists an assertion R and a link from P to R labelled by an event e.

Proof: From the previous lemma, M satisfies $P \rightsquigarrow R$. □
⟨3⟩2. There are assertions $R_1, \ldots R_k$ and guards $g_1, \ldots, g_k$ such that links $P \xrightarrow{g_i} R_i$ exists for any $i \in 1..k$.
Proof: From the previous lemma, M satisfies $P \rightsquigarrow \bigvee_{i \in 1..k} R_i$. □
⟨3⟩3. Q.E.D.
Proof: From ⟨3⟩1 and ⟨3⟩2, M satisfies $P \rightsquigarrow S$, where S is either $\bigvee_{i \in 1..k} R_i$ or R. □
⟨2⟩2. M satisfies $S \rightsquigarrow Q$, where S is either $\bigvee_{i \in 1..k} R_i$ or R.
Proof: If we consider the two possible cases, we notice that each R_i is related to Q by a path of length smaller than n and that each path leads to Q. We can apply the induction hypothesis to derive that M satisfies $R_i \rightsquigarrow Q$ for any $i \in 1..k$. By applying the confluence property of $\rightsquigarrow$, we derive that M satisfies $\bigvee_{i \in 1..k} R_i \rightsquigarrow Q$, and by applying the transitivity rule, we derive that M satisfies $P \rightsquigarrow Q$. □
⟨2⟩3. Q.E.D.
Proof: By induction and induction steps ⟨1⟩1 and ⟨1⟩2, we derive that M satisfies $P \rightsquigarrow Q$. □

In addition, these diagrams can be used for deriving new liveness properties, and the following lemma shows how these diagrams are simply used. The refinement diagram is built on a set of control labels, and there are two distinctive nodes corresponding to the precondition and the postcondition of the procedure under development.

Lemma 4. Let M be a machine, and let $D = (A, L, M, P, Q, G, E)$ be a refinement diagram for M. If I,U, V, P, and Q are assertions such that

- I is the invariant of M,
- $P \wedge I \Rightarrow U$,
- $V \Rightarrow Q$,
- and there is a path from U to V and each path from U leads to V,

then M satisfies $P \rightsquigarrow Q$.

A very direct application of the lemma proves the soundness of the methodology based on refinement diagrams.

Theorem 2. Let M be a machine and let $D = (A, C, M, P, Q, G, E)$ be a refinement diagram for M.

Then M satisfies $(c = start \wedge PRE(D)) \rightsquigarrow (c = end \wedge POST(D))$.

This theorem tells us that a refinement diagram ensures that the machine M describes a process that satisfies the eventuality property, expressing correctness with respect to the pre/post specifications. The dual issue is being able to attach a refinement diagram to a given machine M that corresponds to the refinement of a machine PREPOST.

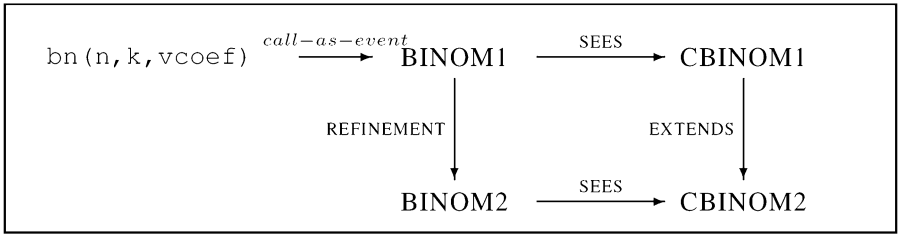
Guideline 3 REFINEMENT

The refinement of PREPOST into M is driven by a refinement diagram that provides an algorithmic technique for solving the problem. The main idea is to use the inductive definition of the computed values, which are in the context PB. The problem is decomposed into sub-problems, which are solved by events, and the refinement guarantees the simulation of PREPOST by the resulting machine M. Each event solves a subproblem of the target problem.

At this point, we have not yet produced an algorithm or a procedure. Therefore, we will illustrate the use of the methodology already identified via a simple example.

3.3 The problem of Pascal's triangle

We consider a very simple problem to be solved, which belongs to the class of dynamic programming problems. The goal is to complete the following structure.



3.3.1 Analyzing the context

Context of the calculations Pascal's triangle provides a geometrical way of understanding the calculation of binomial coefficients. It assigns to the calculation of binomial coefficients an underlying algorithm based on dynamic programming principles. We follow Guideline 1 to produce the context of the problem by the translation of the mathematical definitions.

n \ $k\rightarrow$	0	1	2	3	4
$\downarrow$ 0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1

The connection between Pascal's triangle and binomial coefficients is based on the fact that a binomial is a polynomial expression of two variables, such as 1, $x+y$, $x^2+2xy+y^2$, and $x^3+3x^2y+3xy^2+y^3$. More precisely, the expansion of $(x+y)^n$ is simply given by the general formula.

$$\sum_{k=0}^{k=n} \binom{n}{k} x^k y^{n-k} \tag{1}$$

The interpretation of the first line of the triangle is the expansion of $(x+y)^0$, namely 1, the expansion of $(x+y)^1$ is $x+y$, and so on. Finally, Pascal's triangle provides a simple way to compute a binomial coefficient by following the relation between binomial coefficients of the $k-1$ and k lines as follows.

$$\forall k \in \{1, \ldots n-1\}. \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \tag{2}$$

$$\binom{n}{n} = 1 \tag{3}$$

$$\binom{n}{0} = 1 \tag{4}$$

The *dynamic-programming* style is explicitly inherited from the relation between the line k and the line $k-1$. We have now to suggest a way to compute the value of $\binom{n}{k}$. Mathematical definitions of the binomial coefficients are integrated in the Event B environment by defining them in a context. The constant c stands for coefficients of Pascal's triangle.

CONTEXT CBINOM1

CONSTANTS n, k, c

AXIOMS

$axm1 : n \in \mathbb{N}$

$axm2 : k \in \mathbb{N}$

$axm3 : k \leq n$

$axm4 : c \in 0 \mathinner{..} n \times 0 \mathinner{..} n \nrightarrow \mathbb{N}$

$axm5 : dom(c) = \{i \mapsto j | i \in 0 \mathinner{..} n \wedge i \in 0 \mathinner{..} n \wedge j \leq i\}$

$axm6 : \forall j \cdot j \in 0 \mathinner{..} n \Rightarrow c(j \mapsto 0) = 1$

$axm7 : \forall j \cdot j \in 0 \mathinner{..} n \Rightarrow c(j \mapsto j) = 1$

$$axm8 : \forall i, j \cdot \left(\begin{array}{l} i \in 0 \mathinner{..} n \\ \wedge\ j \in 0 \mathinner{..} n \\ \wedge\ i < j \end{array} \right) \Rightarrow c(j \mapsto i) = c(j-1 \mapsto i) + c(j-1 \mapsto i-1)$$

END

Specification of binomial coefficient computations We have to define a new model called BINOM1, which defines an event corresponding to the action of calling the procedure. We apply Guideline 2, which produces a single call instance ($vcoef := c(n \mapsto k)$), obtaining the following structure.

MACHINE BINOM1
SEES CBINOM1
VARIABLES
$vcoef$
INVARIANTS
$inv1 : vcoef \in \mathbb{N}$
EVENTS
EVENT INITIALISATION
BEGIN
$act1 : vcoef :\in \mathbb{N}$
END
EVENT computing
BEGIN
$act1 : vcoef := c(n \mapsto k)$
END
END

The machine BINOM1 contains only one event, which expresses the assignment of a mathematically defined value to the variable $vcoef$. Having stated the *what*, we now have to explain the *how*. The procedure call is `bn(n,k,vcoef)`.

3.3.2 Building a refinement diagram for the binomial coefficients

The model BINOM1 is derived from the inductive definition of the binomial coefficients, which is in the context CBINOM1. The refinement of BINOM1 to BINOM2 introduces control points and a control variable called l. We need to add new control point values in the context CBINOM1. The process is divided into three main steps, according to the value of k. When k is 0 or n, the result is 1. When the value of k is neither 0 nor n, then the process is decomposed into two calls. A refinement diagram (see Fig. 4) provides details of the various events and the dependency between these events.

Finally, the model BINOM 2 is derived from the refinement diagram and contains the various steps of the computation. The definition of c is crucial, but the number of proof obligations is less critical. The two interactive proof obligations are proved by an instantiation of an axiom of c.

The context CBINOM2 extends the context CBINOM1 by defining control points.

3.3.3 Events of BINOM2

Stating the invariant The model BINOM2 refines the model BINOM1 by intro-

ducing three new variables.

- l states the current control: $inv1 : l \in LOC$.
- $vtcoefx$ is an intermediate variable containing the value of $c(n-1 \mapsto k-1)$ when the control is at *cally*.
- $vtcoefy$ is an intermediate variable containing the value of $c(n-1 \mapsto k)$ when the control is at *endcalling*.

model	Total	Auto	Manual	Reviewed	Undischarged
CBINOM1	3	3	0	0	0
BINOM1	4	4	0	0	0
BINOM2	42	40	2	0	0
Global	49	47	2	0	0

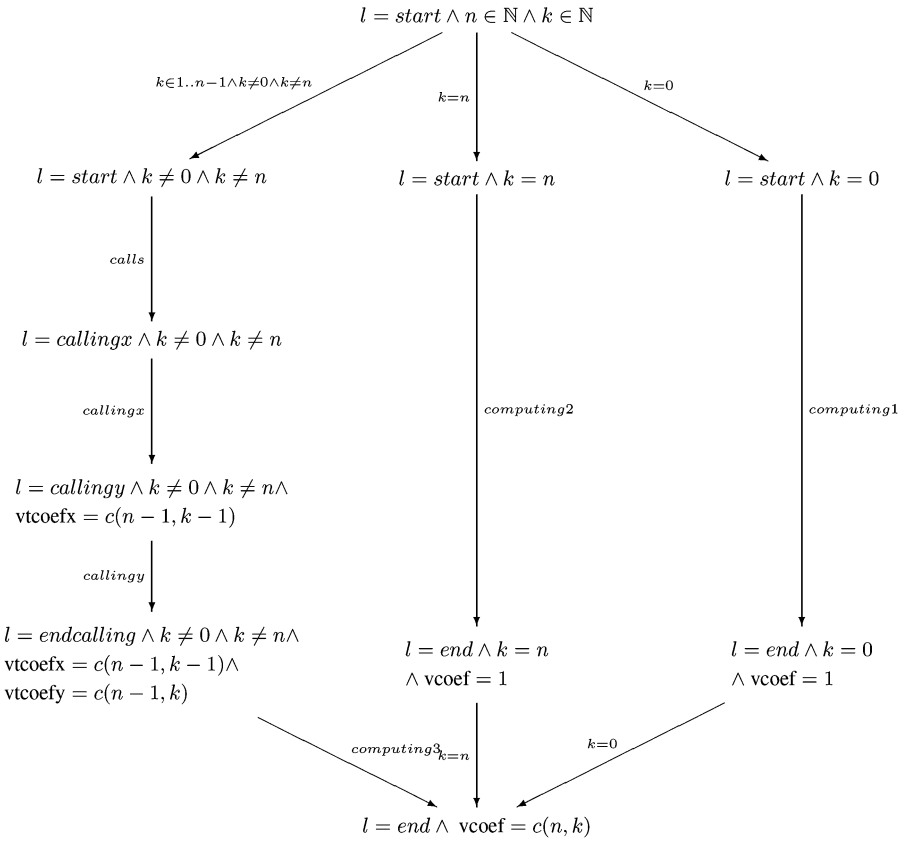


Figure 4. Refinement diagram for the binomial coefficients computing

The invariant is very easy to derive, because we infer it from the global organization of the computations. The control points *callx*, *cally*, and *endcalling* are central to expressing the results of the procedure calls.

- $inv5 : l \in \{callx, cally, endcalling\} \Rightarrow k \neq 0 \wedge n \neq 0 \wedge k < n$: the definition of c is expressed using the value n when k is in $1..n-1$. Otherwise, the values are clearly equal to one.

- $inv6 : l = cally \Rightarrow vtcoefx = c(n-1 \mapsto k-1)$: $vtcoefx$ contains the result of the recursive call for $n-1$ and $k-1$.
- $inv7 : l = endcalling \Rightarrow vtcoefy = c(n-1 \mapsto k) \wedge vtcoefx = c(n-1 \mapsto k-1)$: at this point, $vtcoefy$ contains the result of the recursive call for $n-1$ and k.
- $inv8 : l = end \Rightarrow vcoef = c(n \mapsto k)$: the final result is assigned to $vcoef$ from the definition of the binomial coefficients.

Defining controlled events The INITIALISATION assigns l to $start$, with other variables being set to natural numbers. Following the refinement diagram 4 for binomial coefficient computing, we consider three events that refine the event COMPUTING.

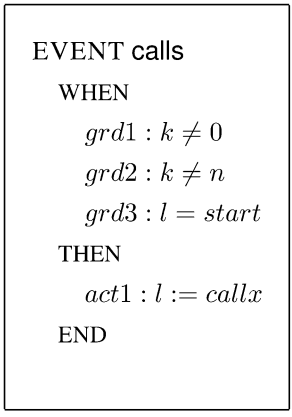
According to c, if k is 0 or n, then the value of $c(n \mapsto k)$ is 1 and the event sets 1 to $vcoef$:

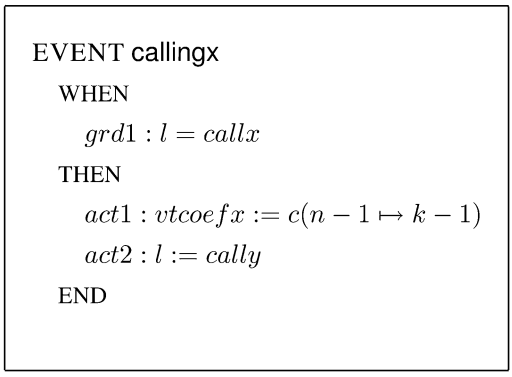
EVENT computing1
REFINES computing
WHEN
$grd1 : k = 0$
$grd2 : l = start$
THEN
$act1 : vcoef := 1$
$act2 : l := end$
END

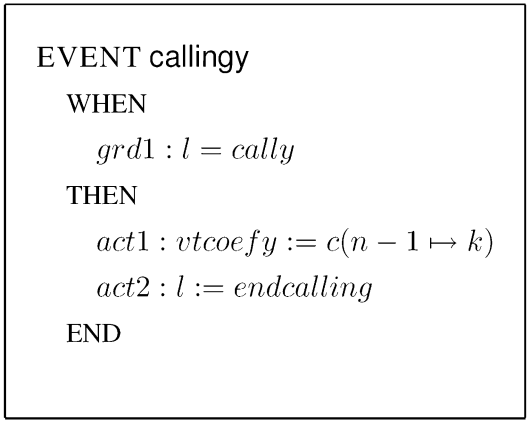
EVENT computing2
REFINES computing
WHEN
$grd1 : l = start$
$grd2 : k = n$
THEN
$act1 : vcoef := 1$
$act2 : l := end$
END

The EVENT computing 3 deals with the general case and is triggered when the value of k is in $1..n-1$. Considering the refinement diagram for binomial coefficient computing, this expresses that the control is at $endcalling$.

EVENT computing3
REFINES computing
WHEN
$grd1 : l = endcalling$
THEN
$act1 : vcoef := vtcoefx + vtcoefy$
$act2 : l := end$
END







The next three events complete the refinement diagram for binomial coefficient computing by adding the links from *start* to *endcalling*. The EVENT calls transfers the control to the label *callx*. When the control is at *callx*, the EVENT callingx solves the problem for $n-1$ and $k-1$, meaning that $c(n-1 \mapsto k-1)$ is assigned to $vtcoefx$. The control is transferred to *cally*.

The next step is to apply the mapping transformation to obtain the code of the procedure. The transformation of the model M into an algorithm is based on the refinement diagram.

3.4 Producing an algorithm from the model M

We have indicated that the mapping transformation is mechanized, but this is not the case for the *refinement* step and the *sees* step, because some problem analysis is required. The implementation should offer help with these manipulations, and we have an initial part that defines the mapping transformation via a plug-in for RODIN. When applying the *call-as-event* principle, we produce a refinement machine that simulates the control flow of the program and that expresses the correctness of the program with respect to labels and events. We can build a refinement diagram for the problem, without introducing cycles. The *mapping* transformation produces an algorithm from a model M.

We assume that variables of M are x, y, z, and c, and the set of events is E. Each event $e \in E$ is transformed into a term $proc(e)(x, y, z)$ that simulates the call of a procedure $proc(e)$ assigned to e. We consider several cases according to the control flow of M, which is the same control flow as for D.

- If P is the input node, then $\{P\}$ is a comment at label *start*.
- If Q is the output node, then $\{Q\}$ is a comment at label *end*.
- If R is related to S by a unique arrow labelled $e \in E$, then the fragment of code produced at the label of R is the following:
  - $\{R\}$ is a comment at label $\ell(R)$.
  - Add the conditional statement from this point.

$$\begin{array}{lll} \{R\} & & \\ \textbf{IF} & guard(e)(x,y,z) & \textbf{THEN} \\ & trans(e)(x,y,y) & \\ \{S\}, & & \end{array}$$

    where $guard(e)(x, y, z)$ is the guard of e transformed into the programming language by a procedure call, and $trans(e)(x, y, y)$ is the procedure call corresponding to the specification of the transformation of variables x, y, and z according to e.

- If R is related to $S_1, \ldots, S_p$, then the fragment of code produced at the label of R is the following.
  - $\{R\}$ is a comment at label ℓR.
  - Add the conditional statement from this point.

$$\begin{array}{lll} \{R\} & & \\ \textbf{IF} & guard(g_1)(x,y,z) & \textbf{THEN} \\ \{S_1\} & & \\ \textbf{IF} & guard(g_2)(x,y,z) & \textbf{THEN} \\ \{S_2\} & & \\ \ldots & \ldots & \ldots \\ \textbf{IF} & guard(g_n)(x,y,z) & \textbf{THEN} \\ \{S_n\}, & & \end{array}$$

    where $guard(g_i)(x, y, z)$ is the expression of g_i in a programming language and is a procedure call.

Any event e is defined by an expression such as $x : |(R(x, x'))$, stating that the next value of x satisfies $R(x, x')$. It has a guard g(x) and may not correspond to any programming assignment such as $x := f(x)$. The requirement is to produce the code corresponding to the event, which is written by $guard(g_i)(x, y, z)$, $guard(e)(x, y, z)$, or $trans(e)(x, y, y)$, and we simply reapply the same technique. A new development is started by a new PREPOST machine that includes the event in developing into a new procedure. A special case is when the event is an instance of the event under

development. This means that we have to translate this instance by a recursive call of the procedure under development in the current model M.

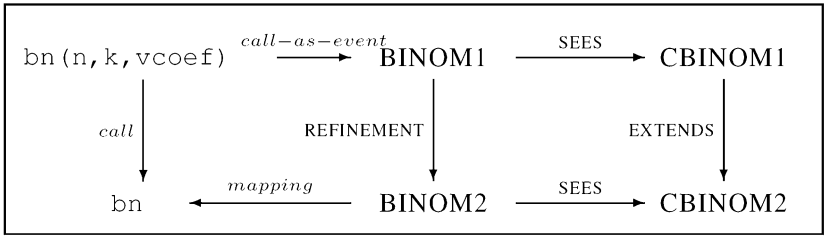
The procedure header is `bn(k,n,vcoef)`, and the text of the procedure is given by the algorithm 3.4. This process has been mechanized by a plug-in added to the RODIN platform. The main idea is to analyze the various cases. Either the procedure exists or it is a new problem to be solved. In the case of binomial coefficient computing, we have no new procedure to develop, but we translate EVENT callingx and EVENT callingy as recursive calls of the procedure *bn*.

Algorithm 1: `bn(k,n; var vcoef)`

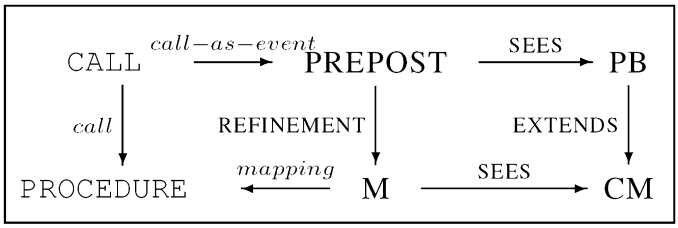
precondition : $k \in 1..n$
postcondition : $vcoef = c(n \mapsto k)$
local variables: $vtcoefx, vtcoefy \in \mathbb{N}$

/* $vcoef \in \mathbb{N} \wedge vtcoefx \in \mathbb{N} \wedge vtcoefy \in \mathbb{N}$ */
if $k = 0$ **then**
  /* $k = 0 \wedge vcoef \in \mathbb{N} \wedge vtcoefx \in \mathbb{N} \wedge vtcoefy \in \mathbb{N}$ */
  $vtcoef := 1$;
else
  if $k = n$ **then**
    /* $k = n \wedge vcoef \in \mathbb{N} \wedge vtcoefx \in \mathbb{N} \wedge vtcoefy \in \mathbb{N}$ */
    $vtcoef := 1$;
  else
    /* $k \neq 0 \wedge k \neq n \wedge vcoef \in \mathbb{N} \wedge vtcoefx \in \mathbb{N} \wedge vtcoefy \in \mathbb{N}$ */
    $bn(k-1, n-1, vtcoefx)$;
    /* $k \neq 0 \wedge k \neq n \wedge vcoef \in \mathbb{N} \wedge vtcoefx \in \mathbb{N} \wedge vtcoefy \in \mathbb{N} \wedge vtcoefx = c(n-1 \mapsto k-1)$ */
    $bn(k, n-1, vtcoefy)$;
    /* $k \neq 0 \wedge k \neq n \wedge vcoef \in \mathbb{N} \wedge vtcoefx \in \mathbb{N} \wedge vtcoefy \in \mathbb{N} \wedge vtcoefy = c(n-1 \mapsto k) \wedge vtcoefx = c(n-1 \mapsto k-1)$ */
    $vcoef := vtcoefx + vtcoefy$;
    /* $k \neq 0 \wedge k \neq n \wedge \wedge vcoef = c(n \mapsto k)$ */
;
/* $k \in 0..n \wedge vcoef = c(n \mapsto k)$ */

Finally, the global structure for the case study is as follows.



We have obtained a complete structure that describes the various parts of the whole development. Such a structure is a *pattern*, because it is defined by a diagram.



We can summarize the elements of the general structure:

- `CALL` and `PROCEDURE` are elements of the programming language.
- PREPOST, M, CM, and PB are elements of the modelling language.
- The call-as-event transformation produces a model PREPOST and a context PB from `CALL`.
- The mapping transformation enables us to derive an algorithmic procedure, and this can be mechanized. A plug-in is developed and can be used to derive the algorithm from the component M.
- `PROCEDURE` is a node corresponding to a procedure derived from the refinement model M. `CALL` is an instantiation of `PROCEDURE` using parameters x and y.
- M is a refinement model of PREPOST, which is transformed into `PROCEDURE` by applying structuring rules. It may contain events corresponding to calls of other procedures, and those events are defined nondeterministically. Each call is solved by the same principle, and we apply it for as long as there are no unresolved events.

Abrial[4] proposes a list of transformations rules for producing sequential algorithms from Event B models, and the idea is to reduce two events, guarded by a condition and its complement, into a conditional statement or a loop statement. Our technique does not produce loop statements, because we base our analysis on the use of inductive data structures, and refinement diagrams are acyclic structures by definition. Our translation is systematic. We have described it by using the refinement diagram and it is clear that we can also use the control flow induced by the events and the control variable to get an algorithm.

We have shown that a refinement diagram for a refinement diagram for a machine M, P, and Q over L and A is an acyclic labelled graph over A with labels from G or E satisfying a given set of rules. Let us consider the question of the construction of a refinement diagram from a machine M as described in the definition of these diagrams. M has a variable c which is modelling the control flow and other variables. From events of M, one can generate a graph over control points and one suppose that there is only one node with no incoming link, namely start and only one node with no outcoming link, namely end. If an event event EVENT e modifies control variable c from l to l', there is a link from l to l'. We assume that the resulting graph is acyclic and connected. We have obtained the skeleton of a refinement diagram. If P and Q are the pre/post conditions for M such that M satisfies $P \leadsto Q$, the fair weakest-precondition $FWP(M)(Q)$ defines the weakest precondition leading to Q under fairness assumption for M. By construction, $P \wedge I(M) \Rightarrow FWP(M)(Q)$. The last step for the construction of the refinement diagram is to attach to each label

l an assertion denoted A_l and defined by $A_l \equiv c = l \wedge FWP(M)(Q)$. By construction, the resulting graph is a refinement diagram for M, P, and Q over L and A. The short explanation shows that it is always possible to build a refinement diagram, when the machine M satisfies conditions listed above.

4 Applications

We illustrate the use of patterns by considering classical algorithmic problems.

4.1 Primitive recursive functions

The first application is the development of an algorithm from the definition of primitive recursive functions. We have already used this example[19] for illustrating the development of Event B models, but we now apply the current methodology, and we re-use elements of Ref. [19].

The class of primitive recursive functions In computability theory[46], the primitive recursive functions constitute a strict subclass of general recursive functions, which is also called the class of computable functions. Many computable functions are primitive recursive, including addition, multiplication, exponentiation, and sign. In fact, a primitive function corresponds to a bounded (for) loop, and we show how to derive the (for) algorithm from the definition of the primitive recursive functions.

The primitive recursive functions are defined by initial functions (the 0-place zero function ζ, the k-place projection function ${\pi_i}^k$, the successor function σ) and by two combining rules, namely, the composition rule and the primitive recursive rule. More precisely, we give this definition of functions and rules.

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

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

  then f is obtained from g and h_1, ..., h_ℓ by composition.

- If g is a ℓ-place function, if h is a $(\ell + 2)$-place function, and if the function f is defined by
  F $\forall x_1, \ldots, x_\ell, x \in \mathbb{N}$,

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

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

A function f is primitive recursive if it is an initial function or can be generated from initial functions by some finite sequence of the operations of composition and primitive recursion. A primitive recursive function is computed by an iteration, and

we define a general framework for stating the development of functions defined by primitive recursion using predicate diagrams.

A first Event B component is derived from the definitions of these functions.

CONTEXT CPRIMREC1
CONSTANTS
f, g, h, a, b
AXIOMS $axm1 : g \in \mathbb{N} \to \mathbb{N}$
$axm2 : h \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
$axm3 : f \in \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
$axm4 : \forall x \cdot x \in \mathbb{N} \Rightarrow f(x \mapsto 0) = g(x)$
$axm5 : \forall x, y \cdot x \in \mathbb{N} \wedge y \in \mathbb{N} \Rightarrow f(x \mapsto y+1) = h(x \mapsto y \mapsto f(x \mapsto y))$
$axm6 : a \in \mathbb{N}$
$axm7 : b \in \mathbb{N}$
END

We have simplified the definitions of primitive recursive functions, because we had to add details about the fact that the resulting function is the fixed point of the set of equations. Our choice is to define them in the list of axioms, and this means that we guarantee the consistency of the definitions according to our reference manual[46].

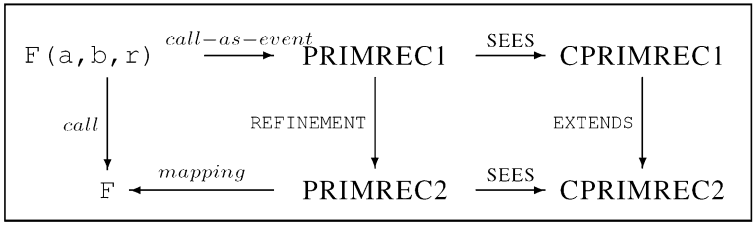
Specification of the computing We apply the following schema, producing a first specification related to the issue of computing the value $f(a, b)$.

Model	Total	Auto	Manual	Reviewed	Undischarged
CPRIMREC1	2	2	0	0	0
PRIMREC1	5	3	2	0	0
PRIMREC2	26	11	15	0	0
Global	33	16	17	0	0

The machine PRIMREC1 is simply defined as follows.

MACHINE PRIMREC1
SEES CPRIMREC1
VARIABLES
r
INVARIANTS
$inv1 : r \in \mathbb{N}$
EVENTS
EVENT INITIALISATION
BEGIN
$act1 : r :\in \mathbb{N}$
END
EVENT computing
BEGIN
$act1 : r := f(a \mapsto b)$
END
END

Clearly, we have another expression that uses a mathematical expression on the left side of the assignment. This defines the *what*. We consider the problem for deriving a procedure for computing a primitive recursive function f defined by the classical equations from two other functions g and h.



We have mainly used the automatic procedures provided by the RODIN platform. The context CPRIMREC2 is, in fact, the context CPRIMREC1.

The decomposition of the computation of f is based on its definitions, and we derive the refinement diagram (see Fig. 5) and the model PRIMREC2, defined as follows.

MACHINE PRIMREC2
REFINES PRIMREC1
SEES CPRIMREC2
VARIABLES
$r, c, tempr$
INVARIANTS
$inv1 : c \in LOC$
$inv2 : tempr \in \mathbb{N}$
$inv4 : c = callingf \Rightarrow b \neq 0$
$inv3 : c = callingh \Rightarrow b \neq 0$
$inv5 : c = callingh \Rightarrow tempr = f(a \mapsto b-1)$

EVENTS
EVENT INITIALISATION
BEGIN
$act1 : r :\in \mathbb{N}$
$act2 : c := start$
$act3 : tempr :\in \mathbb{N}$
END
EVENT computing1
REFINES computing
WHEN
$grd1 : c = start$
$grd2 : b = 0$
THEN
$act1 : r := g(a)$
$act1 : c := end)$
END
EVENT calling
WHEN
$grd1 : c = start$
$grd2 : b \neq 0$
THEN
$act1 : c := callingf$
END

EVENT callingf
WHEN
$grd1 : c = callingf$
THEN
$act1 : tempr := f(a \mapsto b-1)$
$act2 : c := callingh$
END
EVENT callingh
REFINES computing
WHEN
$grd1 : c = callingh$
THEN
$act1 : r := h(a \mapsto b-1 \mapsto tempr)$
$act2 : c := end$
END
END

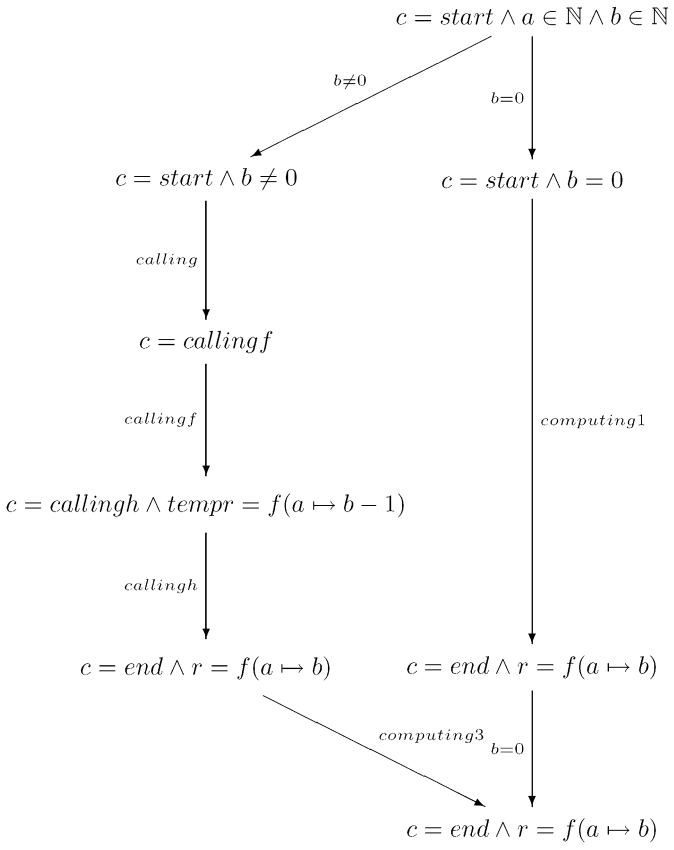


Figure 5. Refinement diagram for computing f from g and h

In this case, both events EVENT computing1 and EVENT callingh are translated into two calls corresponding to the procedure computing g and to the procedure computing h. Next, we have to develop the two procedures using the same pattern. This case study shows that we have a hierarchical and structured development when using the structure of context and the Event B machine. The algorithm of the procedure is easily derived from these events.

Algorithm 2: `F(a,b; var r)`

precondition : $a \in \mathbb{N} \wedge b \in \mathbb{N}$
postcondition : $r = f(a \mapsto b)$
local variables: $tempr \in \mathbb{N}$

if $b = 0$ **then**
  $G(a, r)$;
else
  $F(a, b - 1, tempr)$;
  $H(a, b, tempr, r)$;
;

- Event COMPUTING1 is translated by the procedure call `G(x,y)`.
- Event CALLINGH is translated by the procedure call `H(x,y,z,t)`.
- Event CALLING is translated by the procedure call `F(x,y,z)`.

The invariant is not very difficult to discover, because it is based on inductive analysis. This is the main advantage of this technique. G and H are developed by reapplying the same pattern, as long as it remains events which are not implemented.

4.2 Floyd's algorithm

Summary of the problem Floyd's algorithm[29] computes the shortest distances in a graph and is based on an algorithmic design technique called dynamic programming, where simpler sub-problems are first solved before the full problem is solved. It computes a distance matrix from a cost matrix, where the cost of the shortest path between each pair of vertices is $O(|V|^3)$. The set of nodes N is $1..n$, where n is a constant value, and the graph is simply represented by the distance function d ($d \in N \times N \times N \pfun \mathbb{N}$). When the function is not defined, it means that there is no vertex between the two nodes. The relation of the graph is defined as the domain of the function d. n is clearly greater than 1, meaning that the set of nodes is not empty.

The distance function d is defined inductively from bottom to top according to the principle of dynamic programming, and the following axioms define this function.

- $axm1 : d \in N \times N \times N \pfun \mathbb{N}$: d is a partial function, which assigns a value representing the distance v from a node i to a node j while intermediate nodes are smaller than k. It is defined by $d(k \mapsto i \mapsto j) = v$. The function is partial, because the value may not exist. Other axioms define the function d inductively.
- $axm5$: $\forall i \cdot i \in N \Rightarrow 0 \mapsto i \mapsto i \in dom(d) \land\ d(0 \mapsto i \mapsto i) = 0$.
- $axm6 : \forall i, j, k \cdot$

$$\left(\begin{array}{l} \left(\begin{array}{l} k-1 \mapsto i \mapsto j \in dom(d) \\ \land\ (k-1 \mapsto i \mapsto k \notin dom(d) \lor k-1 \mapsto k \mapsto j \notin dom(d)) \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} k \mapsto i \mapsto j \in dom(d) \land\ d(k \mapsto i \mapsto j) = d(k-1 \mapsto i \mapsto j) \end{array} \right) \end{array} \right).$$

- $axm7 : \forall i, j, k \cdot$

$$\begin{array}{l} \left(\begin{array}{l} k-1 \mapsto i \mapsto j \in dom(d) \\ \land\ k-1 \mapsto i \mapsto k \in dom(d) \\ \land\ k-1 \mapsto k \mapsto j \in dom(d) \\ \land\ d(k-1 \mapsto i \mapsto j) \leqslant d(k-1 \mapsto i \mapsto k) + d(k-1 \mapsto k \mapsto j) \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} k \mapsto i \mapsto j \in dom(d) \\ \land\ d(k \mapsto i \mapsto j) = d(k-1 \mapsto i \mapsto j) \end{array} \right). \end{array}$$

- $axm8 : \forall i, j, k \cdot$

$$\left(\begin{array}{l} k-1 \mapsto i \mapsto j \in dom(d) \\ \wedge\ k-1 \mapsto i \mapsto k \in dom(d) \\ \wedge\ k-1 \mapsto k \mapsto j \in dom(d) \\ \wedge\ d(k-1 \mapsto i \mapsto j) > d(k-1 \mapsto i \mapsto k) + d(k-1 \mapsto k \mapsto j) \end{array}\right)$$
$$\Rightarrow$$
$$\left(\begin{array}{l} k \mapsto i \mapsto j \in dom(d) \\ \wedge\ d(k \mapsto i \mapsto j) = d(k-1 \mapsto i \mapsto k) + d(k-1 \mapsto k \mapsto j) \end{array}\right).$$

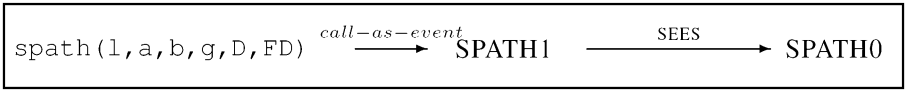
- $axm9 : \forall i, j, k \cdot$

$$\left(\begin{array}{l} \left(\begin{array}{l} k-1 \mapsto i \mapsto j \notin dom(d) \\ \wedge\ k-1 \mapsto i \mapsto k \in dom(d) \\ \wedge\ k-1 \mapsto k \mapsto j \in dom(d) \end{array}\right) \\ \Rightarrow \\ k \mapsto i \mapsto j \in dom(d) \end{array}\right).$$

The optimality property is derived from the definition of d itself, because it starts by defining the bottom elements and applies an optimal principle summarized as follows: $D_{i+1}(a,b) = Min(D_i(a,b), D_i(a,i+1) + D_i(i+1,b))$. This means that the distances in D_i represent paths with intermediate vertices smaller than i. D_{i+1} is defined by comparing new paths including $i+1$. D_i is defined by a partial function over $N \times N \times N$. The partiality of d leads to some possible problems in computing the minimum, and when at least one term is not defined, we should define a specific definition for the resulting term. Floyd's algorithm provides an algorithmic process for obtaining a matrix of all shortest possible paths with respect to a given initial matrix that represents links between nodes together with their distance.

Our first attempt was based on the computation of a shortest path between two given nodes a and b. The resulting matrix is called R, and a Boolean variable FD indicates whether the shortest path exists. In fact, this first attempt was not the strict Floyd's algorithm but used the same principle of computation for the resulting matrix R. In the case of Floyd's algorithm, the computation is global over all possible pairs of nodes.

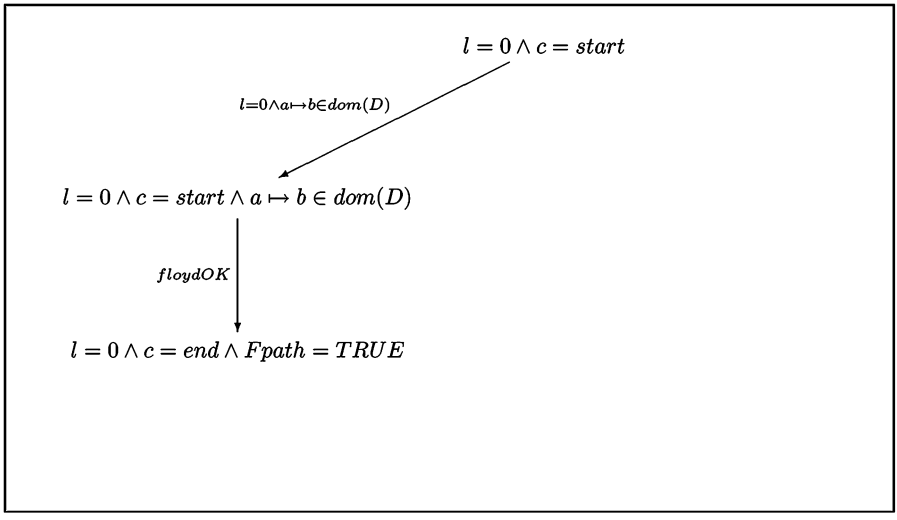
The first step defines the *context* of the problem, which is validated by the RODIN platform[45]. We design an algorithm that computes the value of the shortest path between two given nodes but uses the same principle as Floyd's algorithm.



The new model SPATH1 (see Fig: 6) uses the definitions of the context SPATH0. The event EVENT FLOYDKO models the fact that the call of `floyd` returns the value FALSE for FD, namely, that there is no path between a and b. The event FLOYDOK returns the value TRUE for FD and the value of the minimal path from a to b. The two events are also interpreted by a procedure that is called with respect to the existence of a path.

Next, we have two events, defined using the constant d, that are to be computed. Floyd's algorithm is not as simple as the previous example, but we have developed it after providing a complete definition of d. The main difficulty lies in the definition of d, which models the partiality of the result. The non-existence of a path is expressed by the variable FD.

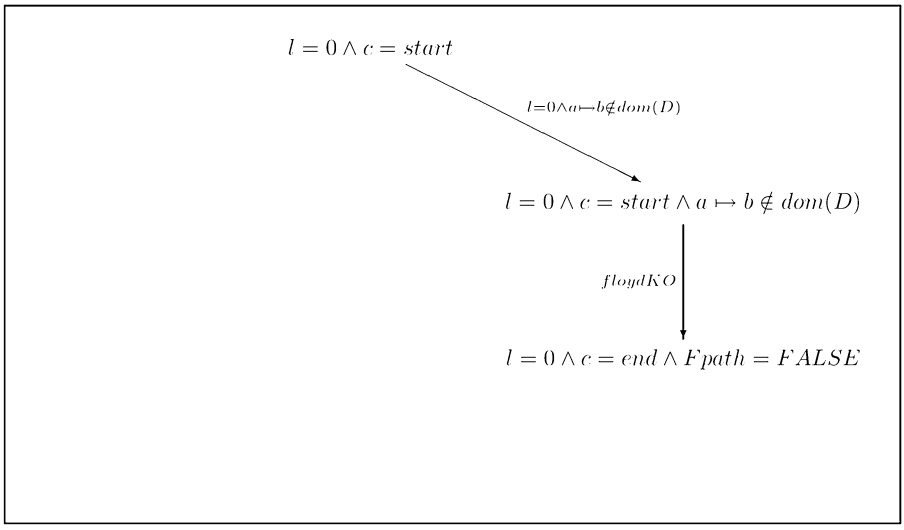
Producing the refinement diagram We construct two parts of the refinement diagram for analyzing the various steps and events that should be added to the refinement model. Consider a simply described case.



EVENT shortestpathOK
REFINES shortestpathOK
WHEN
$grd2 : l = 0$
$grd1 : a \mapsto b \in dom(D)$
$grd3 : c = start$
THEN
$act2 : FD := TRUE$
$act3 : c := end$
END

When the value of l is 0 and D is defined for the pair $a \mapsto b$, it means that there is a path between a and b without any intermediate node. This is the basic case, and

the value TRUE for FD is returned. The control is set to *end*, because the procedure is complete.



EVENT shortestpathKO
REFINES shortestpathKO
WHEN
$grd2 : l = 0$
$grd1 : a \mapsto b \notin dom(D)$
$grd3 : c = start$
THEN
$act1 : FD := FALSE$
$act2 : c := end$
END

When the value of l is 0 and when D is not defined for the pair $a \mapsto b$, it means that there is no elementary path between a and b. It is the basic case and one returns the value FALSE for FD. The control is set to *end*, since the procedure is completed. We have considered the basic case when l is 0.

Completing the development The structure should now be completed, and we obtain a model SPATH2 from the refinement diagram. The invariant is very long but not very difficult to compute. A theorem prover is very useful for this task and we inject new assertions from unproved proof obligations in the invariant. The invariant is defined by interacting with the prover and by considering each possible control state. The main idea is to provide events that refine the events of the abstraction and that take into account the various possible cases for the values $D1$, $D2$, and $D3$

containing respectively $d(l-1 \mapsto a \mapsto b)$, $d(l-1 \mapsto a \mapsto l)$, and $d(l-1 \mapsto l \mapsto b)$, when they are defined. This undefined status produces a number of cases for the invariant. We sketch the events of the refinement model.

MACHINE SPATH1
SEES SPATH0
VARIABLES D
FD
INVARIANTS $inv1 : D \in N \times N \nrightarrow \mathbb{N}$
$inv2 : FD \in BOOL$
EVENTS
INITIALISATION
BEGIN
$$act1 : D :| \left(\begin{array}{l} D' \in N \times N \nrightarrow \mathbb{N} \\ \wedge\ (\forall i, j \cdot 0 \mapsto i \mapsto j \in dom(d) \\ \Rightarrow \\ i \mapsto j \in dom(D') \\ \wedge\ D'(i \mapsto j) = d(0 \mapsto i \mapsto j))) \end{array} \right)$$
$act2 : FD := FALSE$
END
EVENT shortestpathOK
WHEN
$grd1 : l \mapsto a \mapsto b \in dom(d)$
THEN
$act1 : D(a \mapsto b) := d(l \mapsto a \mapsto b)$
$act2 : FD := TRUE$
END
EVENT shortestpathKO
WHEN
$grd1 : l \mapsto a \mapsto b \notin dom(d)$
THEN
$act1 : FD := FALSE$
END
END

Figure 6. Machine SPATH1

- EVENT floydcall-1 transfers the control to step1, when l is not 0.
- EVENT floydcall-2ok and EVENT floydcall-2ko assign $d(l-1 \mapsto a \mapsto b)$ to $D1$, when it is defined ($FD1$ is true).
- EVENT floydcall-3ok and EVENT floydcall-3ko assign $d(l-1 \mapsto a \mapsto l)$ to $D2$, when it is defined ($FD2$ is true).

- EVENT floydcall-4ok and EVENT floydcall-4ko assign $d(l-1 \mapsto l \mapsto b)$ to $D3$, when it is defined ($FD3$ is true).
- EVENT floydcallOK-ppmin, EVENT floydcallOK-ppmax, EVENT floydOK-mp, EVENT floydOKpm, and EVENT floydKOelse refine REFINES floydOK and take into account the value of the Boolean variables $FD1$, $FD2$, and $FD3$ for evaluating the shortest path from a to b through states smaller than l.

The events are controlled by the variable c, which drives the control from *start* to *step*1, *step*2, *step*3, *finalstep*, and *end*.

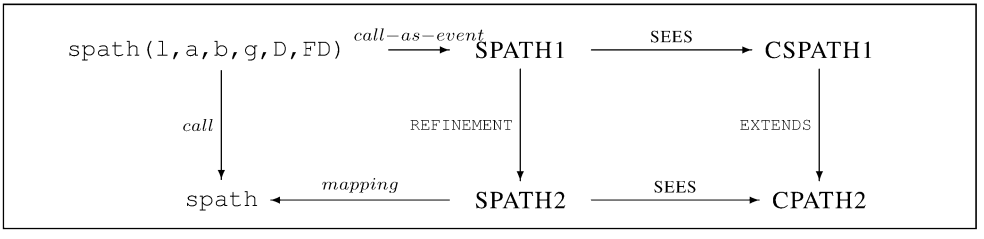
The number of proof obligations is very high, because it corresponds to the defined or undefined status of d for two nodes with respect to a possible set of intermediate nodes. Proofs are not very complex, and the invariant is incrementally built by adding forgotten cases.

Model	Total	Auto	Manual	Reviewed	Undischarged
CSPATH1	8	8	0	0	0
SPATH1	5	4	1	0	0
SPATH2	493	317	176	0	0
Global	506	329	177	0	0

We derive the algorithm from the complete refinement diagram built from the events and omitted because of its size.

The frame (see Fig. 8) contains the C code produced for the algorithm (see Fig. 7). We have produced the C code by hand, forgetting that C array indices start at 0. This means that our initial calls were wrongly written, making it clear that we need a mechanized way to produce code. Moreover, there are some conditions to check and some interactions to manage, with users able to obtain help with their choices.

The pattern is completed:



Others case studies have been developed using the pattern. we have completely developed a searching algorithm during a tutorial session of master students: no preparation was done previously and the example was suggested by students. It appears that the pattern structures the different steps to follow and it helps to discover the invariant by injection of unproved proof obligations. The recursive nature of the analysis makes easier the discovery of invariant. The sortin by insertion is also a developed example and we are developing the CYK algorithm based on dynamic programming techniques: the main effort is in the modelling phase in the context.

Algorithm 3: spath(l,a,b;var D,FD)

precondition : $l \in 1..n \wedge a \in \mathbb{N} \wedge b \in \mathbb{N}$
postcondition : D, FD
local variables: $FD1, FD2, FD3 \in BOOL; D1, D2, D3 \in \mathbb{Z}$

$FD := FALSE;$
$FD1 := FALSE;$
$FD2 := FALSE;$
$FD3 := FALSE;$
if $l = 0$ **then**
  if $(a, b) \in dom(D)$ **then**
    $FD := TRUE;$
    $R := D[a, b];$
  else
    $FD := FALSE;$
else
  $spath(l - 1, a, b, D1, FD1);$
  $spath(l - 1, a, l, D2, FD2);$
  $spath(l - 1, l, b, D3, FD3);$
  case $FD1 \wedge FD2 \wedge FD3$
    if $D1 < D2 + D3$ **then**
      $R := D1;$
    else
      $R := D2 + D3;$
    ;
    $FD := TRUE;$
  ;
  case $FD1 \wedge (\neg FD2 \vee \neg FD3)$
    $R := D1;$
    $FD := TRUE;$
  ;
  case $\neg FD1 \wedge (FD2 \wedge FD3)$
    $R := D2 + D3;$
    $FD := TRUE;$
  ;
  case $\neg FD1 \wedge (\neg FD2 \vee \neg FD3)$
    $FD := FALSE;$
  ;
;

Figure 7. Floyd's algorithm

```
/*  N = 1..n-1 */
void spath (int l, int a, int b,int g[][n],
            int *D, int *FD)
{
  int D1,D2,D3,FD1,FD2,FD3;
  *FD = 0; FD1=0;FD2=0;FD3=0;
  if (l==0)
    {
      if (g[a][b] != NONE)
       { *FD = 1; *D = g[a][b];}
    }
  else
    {
       spath(l-1,a,b,g,&D1,&FD1);
       spath(l-1,a,l,g,&D2,&FD2);
       spath(l-1,l,b,g,&D3,&FD3);
      if (FD1 == 1 && (FD2==1 &&  FD3==1))
        {  if (D1 < D2+D3)
             {*D= D1;}
           else
             {*D=D2+D3;};
           *FD = 1;
        }
      else if (FD1==1 && ( FD2==0 ||  FD3==0))
        {*D= D1;*FD = 1;}
      else if (FD1==0 &&  ( FD2 == 1 && FD3==1))
        {*D=D2+D3; *FD=1;}
      else /* (FD1==0 && ( FD2==0 || FD3==0)) */
        { *FD = 0;}
    }
}
```

Figure 8. C code for the algorithm Fig. 7

5 Conclusion and Perspectives

In this paper, we combine concepts of the Event B method and programming paradigms by defining a proof-based pattern based on the call-as-event principle. Proof-based development constitutes a very powerful framework for constructing procedures, programs, and systems, but it requires the use of formal techniques and formal languages. Consequently, the introduction of patterns or general structures for helping users to obtain correct systems with minimal effort is a very important direction. We have proposed a pattern as a means to structure refinement-based development. We suggest the definition of the pattern as a structure to fill with models or with actions to perform. We consider that it is like a design calculus[32], being based on validation through logical actions. However, the issue is to provide the basic concepts of pattern design and pattern use.

Our main goal is to facilitate the use of the Event B modelling language by proposing techniques and tools. Proofs were improved, even though RODIN has unexpected features. The general structure states the link between programming objects and modelling objects and provides a way to map an Event B model to a sequential program. Refinement is the step that introduces control points and guarantees the correctness of the resulting algorithm. We have formalized, generalized, and illustrated

the technique introduced by Cansell and Méry in Ref. [21], and unspecified details in their paper have been made more precise. The technique of development is a top-down approach, which is clearly well known from the earlier works of Dijkstra[28,40] and uses refinement to control the correctness of the resulting algorithm. It relies on a more fundamental issue related to the notion of the *problem to solve*. A specific diagram is used to organize the refinement, and we call it a refinement diagram. It is very similar to a proof lattice and provides a way of deriving the total correctness of the resulting algorithm. The translation of Event B models into sequential programs has already been proposed by Abrial, but our models correspond to acyclic refinement diagrams, as opposed to those implicitly used by Abrial. We have an automatic transformation of the refinement model into an algorithm.

We have used the technique in the teaching of the design of algorithms. Surprisingly, students discovered invariants using the theorem prover of the RODIN platform. In fact, we think that it is the recursive nature of the development that made the discovery of invariants easier. Proof obligations were not very difficult because we inherit the recursiveness of the structure of the problem. Floyd's algorithm is not a trivial algorithm, but proof obligations were not difficult to prove. It now remains to implement the pattern and to extend it to handle distributed systems.

Finally, we have introduced our work by referring to Eiffel, JML, and Spec#. The translation from an expression in one of these languages to the Event B landscape would be an interesting test of the usability of our method and of the general structure stating the links between the various elements in the development. Further work will integrate the development of distributed algorithms by listing new patterns for the design of correct-by-construction distributed algorithms.

Acknowledgements

Dominique Méry is grateful to Dines Björner for his encouragements. Anonymous referees provide comments on the presentation and on the technique: they have kindly suggested improvements of the french-english version and have pointed out technical problems. This work is supported by grant No. ANR-06-SETI-015-03 awarded by the Agence Nationale de la Recherche.

References

[1] Abrial JR. The B book-Assigning Programs to Meanings. Cambridge University Press, 1996.

[2] Abrial JR. B#: Toward a synthesis between Z and B. In: Bert D, Bowen JP, King S, Wald MA. én, eds, ZB, volume 2651 of Lecture Notes in Computer Science, Springer, 2003. 168–77.

[3] Abrial JR. B#: Toward a synthesis between z and b. In: Bert D, Walden M, eds, 3nd International Conference of B and Z Users-ZB 2003, Turku, Finland, Lectures Notes in Computer Science. Springer, June 2003.

[4] Abrial JR. Event based sequential program development: Application to constructing a pointer program. In: FME 2003, 2003. 51–74.

[5] Abrial JR. Event-based sequential program development: Application to constructing a pointer program. In: Araki K, Gnesi S, Mandrioli D, ed, FME, volume 2805 of Lecture Notes in Computer Science, Springer, 2003. 51–54.

[6] Abrial JR. Modeling in Event-B: System and Software Engineering. Cambridge University Press, 2009.

[7] Abrial JR, Cansell D. Click'n prove: Interactive proofs within set theory. In: TPHOL 2003, 2003. 1–4.

[8] Abrial JR, Hallerstede S. Refinement, decomposition, and instantiation of discrete models: Application to event-b. Fundam. Inform., 2007, 77(1-2): 1–8.
[9] Abrial JR, Mussat L. Introducing Dynamic Constraints in B. In: B98, 1998. 83–128.
[10] Alexander C, Ishikawa S, Silverstein M. A pattern language: towns, buildings, construction. Oxford University Press, 1977.
[11] Back R. On correct refinement of programs. Journal of Computer and System Sciences, 1979, 23(1): 49–68.
[12] Barnett M, Leino R, Schulte W. The Spec# programming system: An overview. In: CASSIS 2004, volume 3362. Springer, 2004.
[13] Benassa N, Cansell D, Méry D. Integration of security policy into system modeling. In: Julliand J, Kouchnarenko O, eds, B, volume 4355 of Lecture Notes in Computer Science, Springer, 2007. 232–247.
[14] Bjorner D. Software Engineering 1 Abstraction and Modelling. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006. ISBN: 978-3-540-21149-5.
[15] Bjorner D. Software Engineering 2 Specification of Systems and Languages. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006. ISBN: 978-3-540-21150-1.
[16] Bjorner D. Software Engineering 3 Domains, Requirements, and Software Design. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006. ISBN: 978-3-540-21151-8.
[17] Bjorner D, Henson MC, eds. Logics of Specification Languages. EATCS Textbook in Computer Science. Springer, 2007.
[18] Cansell D, Gibson JP, Méry D. Refinement: A constructive approach to formal software design for a secure e-voting interface. Electr. Notes Theor. Comput. Sci., 2007, 183: 39–55.
[19] Cansell D, Méry D. The event-B Modelling Method: Concepts and Case Studies, Springer, 2007. 33–40. See [17].
[20] Cansell D, Méry D. Incremental parametric development of greedy algorithms. Electr. Notes Theor. Comput. Sci., 2007, 185: 47–62.
[21] Cansell D, Méry D. Proved-patterns-based development for structured programs. In: Diekert V, Volkov MV, Voronkov A, eds, CSR, volume 4649 of Lecture Notes in Computer Science, Springer, 2007. 104–114.
[22] Cansell D, Méry D, Merz S. Diagram refinements for the design of reactive systems. J. UCS, 2001, 7(2): 159–174.
[23] Cansell D, Méry D, Proch C. System-on-chip design by proof-based refinement. STTT, 2009, 11(3): 217–238.
[24] Cansell D, Méry D, Rehm J. Time Constraint Patterns for Event B Development. In: Julliand OKJ, ed, The 7th International B Conference-B 2007, volume 4355 of Lecture Notes in Computer Science, Besancon, France, Springer. 2007. 140–154.
[25] Chandy KM, Misra J. Parallel Program Design A Foundation. Addison-Wesley Publishing Company, 1988. ISBN 0-201-05866-9.
[26] Clarke EM, Grumberg O, Peled DA. Model Checking. The MIT Press, 2000.
[27] ClearSy, Aix-en-Provence (F). B4FREE, 2004. http://www.b4free.com
[28] Dijkstra EW. A Discipline of Programming. Prentice-Hall, 1976.
[29] Floyd RW. Algorithm 97: Shortest path. Commun. ACM, 1962, 5(6): 345.
[30] Fowler M. Analysis Patterns Reusable Object Models. Addison-Wesley, 1997.
[31] Gamma E, Helm R, Johnson R, Vlissides R, Gamma P. Design Patterns: Elements of Reusable Object-Oriented Software design Patterns. Addison-Wesley Professional Computing, 1994.
[32] Jaehnichen S, Hussain FA, Weber M. Program development by transformation and refinement. In: An international workshop on Advanced programming environments, London, UK, Springer-Verlag. 1986. 471–486.
[33] Lamport L. The temporal logic of actions. ACM Trans. Program. Lang. Syst., 1994, 16(3): 872–923.
[34] Lamport L. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, 2002.
[35] Leavens G. JML web page. http://www.cs.iastate.edu/leavens/JML
[36] Leavens GT, Abrial JR, Batory D, Butler M, Coglio A, Fisler K, Hehner E, Jones C, Miller

D, Peyton-Jones S, Sitaraman M, Smith DR, Stump A. Roadmap for enhanced languages and methods to aid verification. In: Fifth Intl. Conf. Generative Programming and Component Engineering (GPCE 2006), ACM, October 2006. 221–235.

[37] Méry D. A proof system to derive eventuality properties under justice hypothesis. In: Gruszka J, ed, Mathematical Foundations of Computer Science, volume 233 of Lecture Notes in Computer Science, Bratislava, Springer. August 1986.

[38] Méry D. Requirements for a temporal B : Assigning Temporal Meaning to Abstract Machines ... and to Abstract Systems. In: Galloway A, Taguchi K, eds, IFM99 Integrated Formal Methods 1999, Workshop on Computing Science, YORK, June 1999.

[39] Meyer B. Eiffel: The Language. Prentice Hall International Ltd., 1992.

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

[41] Méry D. A simple refinement-based method for constructing algorithms. In: Davies J, Gibbons J, Hinchey M, Taguchi K, eds, First International Workshop on Formal Methods Education and Training, Report GRACE-TR-2008-03. GRACE Center, National Institute of Informatics, 2008.

[42] Méry D. Teaching programming methodology using event b. In: Conference The B Method: from Research to Teaching, Nantes, 2008.

[43] Owicki S, Lamport L. Proving liveness properties of concurrent programs. ACM Trans. Program. Lang. Syst., 1982, 4(3): 455–495.

[44] Polya G. How to Solve It. Princeton University Press, 2nd edition, 1957. ISBN 0-691-08097-6.

[45] Project RODIN. Rigorous open development environment for complex systems. 2004. 2004–2007. http://rodin-bsharp.sourceforge.net/

[46] Rogers HJ. Theory of Recursive Functions and Effective Computability. The MIT Press, 1967.