Modelling Software-based Systems:
Correct by Construction Paradigm

Neeraj SINGH and Dominique MÉRY

November 8, 2024

Contents

Foreword

The document presents the Event-B modelling language for illutrating the correct by construction design methodology.

The Modelling Language Event-B

Dominique MÉRY[1]
[1] *LORIA, Telecom Nancy & Université de Lorraine*

0.1. Introduction

The Event-B (Abrial 2010 ; Cansell and Méry 2007) method is based on a modelling language used to describe state-based models and safety properties of those state-based models. The The originality of Event-B lies in its ability to enable incremental and proof-based modelling of *reactive systems*. The Event-B language contains both set notations and a first-order predicate calculus; it offers the possibility of defining models of reactive systems called machines and contexts and includes the refinement relationship that allows us to follow an incremental development methodology. An Event-B machine is used to describe reactive systems, i.e. systems that react to their environment and its stimuli. An important property of such machines is that they maintain an (inductive) invariant describing the set of reachable states of the current system. The Event-B language has been designed from the classical B (Abrial 1996a) language and proposes a general framework for developing reactive systems, using a progressive approach to model design by refinement. Refinement (Back 1979 ; Dijkstra 1976 ; Back and von Wright 1998 ; Back and Kurki-Suonio 1989) is a relation linking two machines (or models), expressing an enrichment of one model by another; the refinement of an abstract model by a concrete model means that the concrete model simulates the abstract model and that all invariance properties (inductive or not) of the abstract model are preserved in the concrete model. Event-B aims to express system models by an inductive invariant and by invariant properties, also called as safety properties. In the course of the presentation of this language, we will see that that liveness properties of the type $\leadsto$ *leads to* can also be implicitly integrated via the verification conditions for each $\leadsto$ property to be verified. We will also make a comparison with other state-based languages such as UNITY (Chandy and Misra 1988) or TLA$^+$ (Lamport 2002a, 1994). We will look at the induction principles used in the Event-B method and then describe the elements that make up the syntax and the verification conditions called proof obligations.

0.2. Modelling Reactive Systems

Here are some basic elements for modelling what we will call a reactive system or simply a system. Modelling a system is done using well-known recipes, which we will recall first.

Definition 1 (transition system)

A transition system $\mathcal{ST}$ is given by a set of states Σ, a set of initial states $Init$ and a binary relation $\mathcal{R}$ on Σ.

The set of terminal states $Term$ defines specific states, identifying particular states associated with a termination state; this set can be empty, in which case the transition system does not terminate; this aspect can be used to model operating system programs that do not and should not terminate. We use the term *system* rather than program because we can describe more general entities than programs. in the computer sense, but also because this

,

coordinated by The Modelling Language Event-B . © ISTE Editions 2019.

formalism can be used for interactive, concurrent, distributed or hybrid applications[1]. Our definition is general, but we will apply it first to discrete systems. The idea is to observe transformations on the states of the system. Before modelling a system by a transition system, we must observe what constitutes the state of the system and induce transformations that operate on that state. A transformation is caused by an event that updates a temperature from a sensor, or a computer updating a computer variable, or an actuator sending a signal to a controlled entity. An observation of a system S is based on the following points:

– a state $s \in \Sigma$ allows you to observe elements and reports on these elements, such as the number of people in the meeting room or the capacity of the room: $s(np)$ and $s(cap)$ are two positive integers.

– a relationship between two states s and s' observes a transformation of the state s into a state s' and we will note $s \xrightarrow{R} s'$ which expresses the observation of a relationship R: $R = s(np) \in 0..s(cap) - 1 \wedge s'(np) = s(np) + 1 \wedge s'(cap) = s(cap)$ is an expression of R observing that one more person has entered the room.

– a trace $s_0 \xrightarrow{R_0} s_1 \xrightarrow{R_1} s_2 \xrightarrow{R_2} s_3 \xrightarrow{R} \ldots \xrightarrow{R_{i-1}} s_i \xrightarrow{R_i} \ldots$ is a trace generated by the different observations $R_0, \ldots R_p, \ldots$

We insist that we observe changes of state that correspond either to physical or biological phenomena or to artefactual structures such as a program, a service or a platform. An observation generally leads to the identification of a few possible transformations of the observed state, and the closed-model hypothesis follows naturally. One consequence is that there are visible transformations and invisible transformations. These invisible transformations of the state are expressed by an identity relation called event skip (or stuttering (Lamport 1994) or time-stretching (Abrial 1996*b*)). Event-B modelling produces a closed model with a skip event modelling what is not visible in the observed state.

To express properties, a language of assertions $\mathcal{L}$ (or a language of formulae) is important. To simplify, we can take the assertion language $\mathcal{P}(\Sigma)$ (the set of parts of Σ) and $\varphi(s)$ (or $s \in \hat{\varphi}$) means that φ is true in s. The assertion language can be used to express state properties, but the assertion language in question may not be sufficiently expressive. In the context of program correctness, we will assume that assertion languages are sufficiently complete (in Cook's sense), which means that the (state) properties required for completeness can be expressed in the language in question. Properties of a system S which interest us are the state properties expressing that *nothing bad can happen*. In other words, we wish to express state properties such as *the number of people in the meeting room is always smaller than the maximum allowed by law* or *the computer variable storing the number of wheel revolutions is sufficient and no overflow will happen*. A. van Gasteren and G. Tel (van Gasteren and Tel 1990) make a very important comment in the definition of what is *always true* and what is *invariant* and we choose to refer to state properties that are always true as safety properties. Safety properties are, for example, the partial correctness (PC) of an algorithm A with respect to its pre/post specifications (PC), the absence of errors at runtime (RTE) . . . Properties are expressed in the language $\mathcal{L}$ whose elements are combined by logical connectors or by instantiations of variable values in the computer sense called flexible (Lamport 1994). We assume that a system S is modelled by a set of states Σ, and that $\Sigma \overset{def}{=} \mathtt{Var} \longrightarrow \mathrm{D}$ where Var is the variable (or list of variables) of the system S and D is the domain of possible values of variables. The assertion language $\mathcal{L}$ can be used to define first-order predicate calculus formulas using set-theoretic operations ($\in$, $\subseteq$, $\cup$, . . .) and operators ($\wedge$, $\vee$, . . .). The interpretation of a formula P in a state $s \in \Sigma$ is denoted $[\![P]\!](s)$ or sometimes $s \in (\hat{P}$. This hypothesis makes it possible to transfer from an assertion to the set of states validating this assertion. The definition of the validity of an assertion of $\mathcal{L}$ can be given in an inductive form of $[\![P]\!](s)$. The aim is not to give a complete definition but to give an idea of the interpretation of formulae, in order to expose elements specific to the Event-B language. A distinction is made between flexible variable symbols x and logical variable symbols v, and constant symbols c are used.

Example - 1 (interpretation of formulae)

1) $[\![\mathrm{x}]\!](s) = s(\mathtt{x}) = x$: x is the value of the variable x in s.

2) $[\![\mathrm{x}]\!](s') = s'(\mathtt{x}) = x'$: x' is the value of the variable x in s'.

3) $[\![c]\!](s)$ is the value of c in s, in other words the value of the constant c in s.

[1] We will see that so-called hybrid systems require special treatment in terms of mathematical properties involving discrete and continuous domains, in particular Hilbert spaces. These elements will be the subject of two dedicated chapters in the second book in this series, which will complete the presentation of event modelling.

4) $[\![\varphi(x) \wedge \psi(x)]\!](s)$ = $[\![\varphi(x)]\!](s)$ et $[\![\psi(x)]\!](s)$ where *and* is the classical interpretation of symbol $\wedge$ according to the truth table.

5) $[\![\mathtt{x} = 6 \wedge y = \mathtt{x} + 8]\!](s) \stackrel{def}{=} [\![\mathtt{x}]\!](s) = [\![6]\!](s)$ **and** $[\![y]\!](s) = [\![x]\!](s) + [\![8]\!](s) = (x = 6$ **and** $y = x + 8$ where y is a logical variable distinct of $\mathtt{x}$ and where $[\![\mathtt{x}]\!](s) = s(\mathtt{x}) = x$.

We use notations which simplify the reference to states; thus, $[\![\mathtt{x}]\!](s)$ is the value of $\mathtt{x}$ in s and its value will be distinguished by the font used: $\mathtt{x}$ is the $\mathtt{tt}$ font of LaTeX and x is the math font of LaTeX. In this way, we can use the name of the variable $\mathtt{x}$ as its current value, i.e. x and $[\![\mathtt{x}]\!](s')$ is the value of $\mathtt{x}$ in s' and will be noted x'. So $[\![x = 6]\!](s) \wedge [\![y = x + 8]\!](s')$ will be simplified to $x = 6 \wedge y' = x' + 8$. The consequence is that we can write the transition relation as a relation linking the state of the variables in s and the state of the variables in s' using the prime notation as defined by L. Lamport for TLA (Lamport 1994). We distinguish several types of variable depending on whether we are talking about the computer variable, its value or whether we are defining constants such as np, the number of processes, or π, which designates the constant π. In the Event-B approach, a current observation refers to a current state for both endurant and perdurant information data in the sense of the Dines Bjørner (Bjørner 2021) approach.

Definition 2 (flexible variable)

A flexible variable $\mathtt{x}$ is a name related to a perdurant information according to a state of the (current observed) system:

– x is the current value of $\mathtt{x}$ in other words the value at the observation time of $\mathtt{x}$.

– x' is the next value of $\mathtt{x}$ in other words the value at the next observation time of $\mathtt{x}$.

– x_0 is the initial value of $\mathtt{x}$ in other words the value at the initial observation time of $\mathtt{x}$.

A logical variable *x* is a name related to an endurant entity designated by this name.

For a given system S, we will denote $\mathcal{V}(S)$ (resp. $\mathcal{V}ar(S)$) the set of logical (flexible) variables of the system S. The flexible variables are names used in writing the models and this set is used to distinguish these variables from other variables. When observing a system S, we wish to express relations between the flexible variables of this system and we will note such an expression in the form of an assertion of the form $P(x)$ where x is the current value of the flexible variable $\mathtt{x}$ (or a list of flexible variables). The set $\mathcal{V}ar(S)$ is put into the form x i.e. $x \in 1..n \rightarrow \mathcal{V}ar(S)$ where n is the number of flexible variables in the system S. We can write $x = x_1 \ldots x_n$. We have defined the flexible variables which allow us to link the values of the D domain of the system we are observing. Observing a system S means determining the observed values of the D domain. Moreover, if a flexible variable $\mathtt{x}$ is used for modelling a system S, we can use notations as x, x' or x_0. We assume that we can simplify our process by using the expression x to designate the flexible variable $\mathtt{x}$, since we have defined the two sets of variables namely logical and flexible.

Definition 3 (state property of a system)

Let be a system S whose flexible variables x are the elements of $\mathcal{V}ar(S)$. A property $P(x)$ of S is a logical expression involving ,freely the flexible variables x and whose interpretation is the set of values of the domain of x: $P(x)$ is true in x, if the value x satisfies $P(x)$.

For each property $P(x)$, we can associate a subset of D denoted $\hat{P}$ and, in this case, $P(x)$ is true in x. is equivalent to $x \in \hat{P}$.

Examples of property are listed as follow:

– $P_1(x) \overset{def}{=} x \in 18..22$: x is a value between 18 and 22 and $\hat{P}_1 = \{18, 19, 20, 21, 22\}$.

– $P_2(p) \overset{def}{=} p \subset PEOPLE \wedge card(p) \leq n$: p is a set of persons and that set has at most n elements and $\hat{P}_2 = \{p_1 \ldots p_n\}$. In this example, we use a logical variable n and a name for a constant $PEOPLE$.

In our last example, we used $PEOPLE$ which represents a set of people and which we will therefore use to write our expressions. Note the list of symbols $s_1, s_2, \ldots, s_p$ corresponding to the symbols of the sets that make up the domain D.

Definition 4 (basic set of a system S)

The list of symbols $s_1, s_2, \ldots, s_p$ corresponds to the list of basic set symbols in the D domain of S and $s_1 \cup \ldots \cup s_p \subseteq$ D.

Finally, to model a system S, you need symbols for constants and correspond to endurant information data.

Definition 5 (constants of system S)

The list of symbols $c_1, c_2, \ldots, c_q$ corresponds to the list of symbols for the constants of S.

In fact, we are not using the classical partition of logic languages, which separates constants symbols on one side from function symbols on the other. We will give a few examples to get the idea across.

Example - 2 (Examples of constant and set)

– $fred$ is a constant and is linked to the set $PEOPLE$ using the expression $fred \in PEOPLE$ which means that $fred$ is a person from $PEOPLE$.

– aut is a constant which is used to express the table of authorisations associated with the use of vehicles. the expression $aut \subseteq PEOPLE \times CARS$ where $CARS$ denotes a set of cars.

The constants of S cover the basic constants and the functions of S. Each constant c of S must be defined by a list of expressions called axioms.The list of basic sets (resp. constants) is denoted s (resp. c).

Definition 6 (axiom of system S)

An axiom ax(s,c) of S is a logical expression describing a constant or constants of S and can be defined as an expression depending on symbols of constants expressing a set-theoretical expression using symbols of sets and symbols of constants already defined.

We give a few examples of axioms that can be defined:

Example - 3 (Examples of axiom)

– $ax1(fred \in PEOPLE)$: *fred is a person from the set PEOPLE*

– $ax2(suc \in \mathbb{N} \rightarrow \mathbb{N} \wedge (!i.i \in \mathbb{N} \Rightarrow suc(i) = i + 1))$: *The function suc is the total function which associates any natural i with its successor. successor*

– $ax3(\forall A.A \subseteq \mathbb{N} \wedge 0 \in \mathbb{N} \wedge suc[A] \subseteq A \Rightarrow \mathbb{N} \rightarrow \subseteq A)$: This axiom states the induction property for natural numbers. It is an instantiation of the fixed-point theorem.

– $ax4(\forall x.x = 2 \Rightarrow x + 2 = 1)$: This axiom poses an obvious problem of consistency and care should be taken not to use this kind of statement as axiom.

We have numbered axioms and we will use this numbering to define axioms of the system S. One assumption is that axioms are consistent but it should be checked by the user. For any system, we will use a list of axioms to describe constants of S.

Definition 7 (axiomatics for S)

The list of axioms of S is called the axiomatics of S and is denoted $AX(S, s, c)$ where s denotes the basic sets and c denotes the constants of S.

Advice (Consistency of the axiomatisation of S)

Checking the consistency of $AX(S, s, c)$ is an important part of modelling a system. It is quite easy to introduce inconsistency and tools such as Rodin provide the ProB technique based on the discovery of a model in the logical sense. However, this technique has its limits and you need to be very careful.

We have defined an axiomatic system $AX(S, s, c)$ for the system S and we will now derive some properties from this system. These properties will be proved from this axiomatic system and will be the theorems for S.

Definition 8 (theorem for S)

A property $P(s, c)$ is a theorem for S, if $AX(S, s, c) \vdash P(s, c)$ is a valid sequent.

Theorems for S are denoted by $TH(S, s, c)$.

We have shown how definitions of basis sets, constants, axioms and theorems are organised. The flexible variables have an essential quality, since they allow us to account for the state of the system under observation.

Let s, s' be two states of S ($s, s' \in \mathcal{V}ar(S) \longrightarrow$ D). $s \underset{R}{\longrightarrow} s'$ will be written as a relation R(x, x') where x and x' designate values of x before and after the observation of R. We define what is an observed event on the *flexible variables* of the observed system S.

Definition 9 (event)

Let $\mathcal{V}ar(S)$ be the set of flexible variables of S. Let s be the basis sets and c the constants of S. An event e for S is a relational expression of the form $R(s, c, x, x')$ denoted $BA(e)(s, c, x, x')$.

This definition is borrowed directly from TLA (Lamport 1994) and simplifies the presentation of our preliminaries. On the other hand, we will not borrow the set-theoretic language of TLA$^+$ (Lamport 2002*a*). We give examples of events:

Example - 4 (Examples of event)

– $BA(e_1)(x,x') \overset{def}{=} x' = x+1$: e_1 observes the increase of x by one unit.

– $BA(e_2)(x,y,x',y') \overset{def}{=} x'+y' = x+y$: e_2 observes that the values of x and y evolve so that the sum of the two variables is constant.

Convention (meta-language of proofs)

We choose to use logical expressions of the form $P \Rightarrow Q$ or $\forall x.P(x)$ or $\forall x \in E.P(x)$ in the meta-language of proofs and formal expressions. This will allow us to present fundamental results on induction principles and verification conditions. We will then express the verification in a form closer to the logical tools used.

Following an observation of a system S, a set of events E is identified and an event-based model of the system S is obtained.

Definition 10 (event-based model of a system)

Let $\mathcal{V}ar(S)$ be the set of flexible variables of S denoted x. Let s be the list of basis sets of the system S. Let c be the list of constants of the system S. Let D be a domain containing sets s. An event-based model for a system S is defined by

$$(AX(s,c), x, \mathrm{D}, Init(x), \{e_0, \ldots, e_n\})$$

where

– $AX(s,c)$ is an axiomatic theory defining the sets, constants and static properties of these elements.

– $Init(x)$ defines the possible initial values of x.

– $\{e_0, \ldots, e_n\}$ is a finite set of events of S and e_0 is a particular event present in each event-based model defined by $BA(e_0)(x,x') = (x' = x)$.

The event-based model is denoted $EM(s,c,x,\mathrm{D},Init(x)\{e_0,\ldots,e_n\}) = (AX(s,c),x,\mathrm{D},Init(x),\{e_0,\ldots,e_n\})$.

From this structure, we can define a relationship $Next(x,x')$ as follows $Next(s,c,x,x') \overset{def}{=} BA(e_0)(s,c,x,x') \vee \ldots \vee BA(e_n)(s,c,x,x')$. Modelling a system involves giving the variable x, the predicate $Init(x)$ characterising the initial values of the variables and a relationship $Next(s,c,x,x')$ modelling the relationship between the values before and the values after. Safety properties express that *nothing bad can happen (Lamport 1980)*. For example, the value of the variable *x* is always between 0 and 567; the sum of the current values of the variables *x* and *y* is equal to the current value of the value z. To continue our descriptions, we need to introduce the transitive reflexive closure of the relation

$$Next^{\star}(s,c,x_0,x) \overset{def}{=} \begin{cases} \vee\ x = x_0 \\ \vee\ Next(s,c,x_0,x) \\ \vee\ \exists\, xi \in \mathrm{D}.Next^{\star}(s,c,x_0,xi) \wedge Next(s,c,xi,x) \end{cases}$$

Definition 11 (safety property)

A property $P(x)$ is a safety property for the system S, if

$$\forall x_0, x \in \mathrm{D}.Init(x_0) \wedge Next^{\star}(s, c, x_0, x) \Rightarrow P(x).$$

The safety property uses a universal expression to quantify the possible values of the variable *x*. To demonstrate such a property, we can either check the property for every possible value of *x* in the domain D,provided that this set is finite, or use an abstraction of the domain Dor use an induction principle. For the verification for each possible value, we use an algorithm to calculate the values accessible from an initial state. This technique of calculating accessible values is often used and is the basis of automatic verification techniques such as *model-checking* (McMillan 1993 ; Holzmann 1997 ; Clarke *et al*. 2000). We consider an inductive technique to prove safety properties. We observe the logical equivalence between the two equations:

$$\forall x_0, x \in \mathrm{D}.Init(x_0) \wedge Next^{\star}(s, c, x_0, x) \Rightarrow P(x) \qquad [1]$$

$$\forall x \in \mathrm{D}.(\exists x_0 \in \mathrm{D}.Init(x_0) \wedge Next^{\star}(s, c, x_0, x)) \Rightarrow P(x) \qquad [2]$$

Thus, the second equation expresses that the accessible values are safe values with respect to $P(x)$ and gives us the key to the induction principle to be implemented. In fact, the property $(\exists x_0 \in \mathrm{D}.Init(x_0) \wedge Next^{\star}(s, c, x_0, x))$ expresses that x is an accessible value with respect to the $Next(s, c, x_0, x)$ relationship and defines a least fixed point which is an inductive property.

Property 1 (induction principle)

A property $P(x)$ is a safety property for S if, and only if, there exists a property $I(x)$ such that

$$\forall x, x' \in \mathrm{D}. \begin{cases} (1) & Init(x) \Rightarrow I(x) \\ (2) & I(x) \Rightarrow P(x) \\ (3) & I(x) \wedge Next(s, c, x, x') \Rightarrow I(x') \end{cases}$$

The property $I(x)$ is called an (inductive) invariant and is a particular property that is stronger than the other safety properties. The justification for this induction principle is quite simple. This property justifies the method of proof by induction better known as the Floyd/Hoare method (Floyd 1967 ; Hoare 1969), devised by Turing in 1949 (Turing 1949). This property gives a form of induction that must be reduced to more familiar forms. P. and R. Cousot (Cousot 2000 ; Cousot and Cousot 1979, 1992 ; Cousot 1978) give a complete summary of the principles of induction equivalent to this principle of induction. We apply these results to the case of event-based models and obtain an expression for the definition of a safety property. If we transform the properties, we obtain a form closer to what we will use in the following and closer to the notions of event-based models.

Property 2 (equivalence of induction principles)

The following two statements are equivalent:

(I) There exists a state property $I(x)$ such that:

$$\forall x, x' \in \mathrm{D}. \begin{cases} (1) & Init(x) \Rightarrow I(x) \\ (2) & I(x) \Rightarrow P(x) \\ (3) & I(x) \wedge Next(s, c, x, x') \Rightarrow I(x') \end{cases}$$

(II) There exists a state property $I(x)$ such that:

$$\forall x, x' \in \mathrm{D}. \begin{cases} (1) & Init(x) \Rightarrow I(x) \\ (2) & I(x) \Rightarrow P(x) \\ (3) & \forall i \in \{0, \ldots, n\} : I(x) \wedge BA(e_i)(s, c, x, x') \Rightarrow I(x') \end{cases}$$

We have thus given an explanation of the induction rule which is used in Floyd (Floyd 1967 ; Turing 1949 ; Hoare 1969)'s method; this rule makes it possible to put on one side the so-called invariance properties, i.e. those which require an induction step, and the more general safety properties which require an invariance property, i.e. those which require an induction step. of invariance, i.e. inductive properties, in order to be proven. The Event B method implements these two types of property using the INVARIANTS clause for invariants and the THEOREMS clause for safety properties. We deduce that any invariance property is a safety property. We will describe the Event-B language and the method for incremental development of event-based models. The following verification conditions are derived from the above properties.

Definition 12 (Verification conditions for a system S and its invariance and safety properties)

Let $EM(S) = (AX(s,c), x, \mathsf{D}, Init(x), \{e_0, \ldots, e_n\})$ an event-based model for system S. $EM(S)$ is a valid event-based model for S, if the following verification conditions for a system S and its invariance properties I(x) and safety properties A(x) are valid:

– $AX(s,c) \vdash \forall x \in \mathsf{D} : Init(x) \Rightarrow I(x)$

– For any event e in S, $AX(s,c) \vdash \forall x, x' \in \mathsf{D} : I(x) \wedge BA(e)(x,x') \Rightarrow I(x')$

– $AX(s,c) \vdash \forall x \in \mathsf{D} : I(x) \Rightarrow A(x)$

We have used expressions of the form $AX(s,c) \vdash \forall x \in \mathsf{D} .P(x)$ and these expressions are equivalent to $AX(s,c) \vdash \forall x.x \in \mathsf{D} \Rightarrow P(x)$. Considering that x has no free occurrences in $AX(s,c)$, we can simplify the expression into the form $AX(s,c), x \in \mathsf{D} \vdash P(x)$ and meaning that x is a value of D. The expression $x \in \mathsf{D}$ constitutes a typing assumption for x and this information is made explicit directly or indirectly in the invariant $I(s,c,x)$ and in the initial conditions $Init(s,c,x')$. We assume that the information $x \in \mathsf{D}$ is expressed in the initial conditions and in the invariant. The conditions of definition 12 are simplified into the following new consitions:

– $AX(s,c) \vdash Init(x) \Rightarrow I(x)$

– For any event e in S, $AX(s,c) \vdash I(x) \wedge BA(e)(x,x') \Rightarrow I(x')$

– $AX(s,c) \vdash I(x) \Rightarrow A(x)$

Before continuing with this presentation dedicated to Event-B , we would like to remind you that Event-B is a language that supports an approach to developing and modelling systems that are correct by construction. Consequently, the event-based model of the system S $EM(S) = (AX(s,c), x, \mathsf{D}, Init(x), \{e_0, \ldots, e_n\})$ (definition 10) is characterised by the existence of fixed points on the complete lattice $(\mathbb{P}(D), \subseteq)$.

Property 3 (Fixed-point characterization of invariants and safety properties)

Let D be the value domain of the event model $EM(S) == (AX(s,c), x, \mathsf{D}, Init(x), \{e_0, \ldots, e_n\})$. Let $Next(s,c,x,x') \stackrel{def}{=} BA(e_0)(s,c,x,x') \vee \ldots \vee BA(e_n)(s,c,x,x')$. Let $INIT = \{d | d \in \mathsf{D} \wedge Init(d)\}$.

– $(\mathbb{P}(\mathsf{D}), \subseteq)$ is a complete lattice.

– The function $F = \lambda X \in \mathbb{P}(D).(INIT \cup Next[X])$ is monotonically increasing and the equation $X = F(X)$ has a non-empty set of solutions forming a complete lattice.

– Any solution I of the equation$X = F(X)$ satisfies the following property property: $I = F(I)$, $INIT \subseteq I$, $Next[I] \subseteq I, lfp(F) \subseteq I.$

– For any safety property $P(x)$ for system S, $lfp(F) \subseteq \{d | d \in \mathsf{D} \wedge P(d)\}$

A consequence of this property is that a *canonical* invariant can be associated with any event-based model and this invariant is operationally defined by the least fixed-point of the function F. This approach is generally used in the verification of a system. In the case of the Event-B language, we are interested in a correction-by-construction approach. Consequently, the problem is to define a model equipped with an invariant that will be verified and this method is carried out incrementally with the help of refinement. The solutions of the equation $X = F(X)$ correspond to inductive invariants and are useful for showing that a propertyt $P(x)$ is a safety property. The triptych $(EM(S), I, P)$ puts forward an event-based model $EM(S)$ whose (inductive) invariant allows the safety property $P(x)$ to be verified (by verifying the verification conditions necessary for the verification of the invariance of I and the safety of $P(x)$.

Before moving on to the presentation of structures for modelling reactive systems, we are interested in the notion of event and in the so-called feasibility conditions associated with model verification. J.-R. Abrial (Abrial 1996*a*) founds the verification of abstract machines on the calculus wp and uses the notation $[S]P$ (generalised substitution), to express that *S establishes P*.This expression also means that S terminates in *P* and the termination predicate is denoted trm(e)(s,c,x). In our case, S is an event e and we recall the definition $BA(\mathsf{e})(s, c, x, x') \overset{def}{=} \exists u.G(u, s, c, x) \wedge BAP(u, s, c, x, x')$ where $G(u, s, c, x)$ is a guard and $BAP(u, s, c, x, x')$ is a before-after relation$BA(\mathsf{e})(s, c, x, x')$.

Property 4 (*wp* and *relational* styles)

Let be a property $P(x)$.

1) $[\mathsf{e}]P(x) = \forall u.(G(u, s, c, x) \Rightarrow [x : |BAP(u, s, c, x, x')]P(x))$

2) $\mathsf{trm}(\mathsf{e})(s, c, x) = \forall u.(G(u, s, c, x) \Rightarrow \exists x' : BAP(u, s, c, x, x'))$

3) the two following properties are equivalent:

a) $I(s, c, x) \wedge \mathsf{trm}(e)(s, c, x) \Rightarrow [e]I(s, c, x)$ (*wp*)

b) $$\left\{\begin{array}{l} I(s, c, x) \wedge G(u, s, c, x) \Rightarrow \exists x'.BAP(u, s, c, x, x') \\ I(s, c, x) \wedge G(u, s, c, x) \wedge BAP(u, s, c, x, x') \Rightarrow I(s, c, x') \end{array}\right. \quad (\textit{relational})$$

The first two properties are a simple application of the results of J.-R. Abrial (Abrial 1996*a*). The third property is an equivalence between two expressions of the preservation of a state property by an event e. $[e]$ is a predicate transformer which is defining the weakest precondition of e for a given postcondition and is expressing both partial correctness of e and termination of e. J.-R. Abrial (Abrial 1996*a*) has given a complete study of $[e]$ and has given the foundational ideas of Event-B in his seminal talk at B96(Abrial 1996*b*). The Atelier-B (Cle 2002) platform uses verification conditions in the *wp* style and the Rodin platform (Abrial *et al.* 2010) uses a relational style. We are now sketching an explanation of the equivalence.

Explanation (Proof sketch of the property)

$$(1) \left\{\begin{array}{l} I(s, c, x) \wedge G(u, s, c, x) \Rightarrow \exists x'.BAP(u, s, c, x, x') \\ I(s, c, x) \wedge G(u, s, c, x) \wedge BAP(u, s, c, x, x') \Rightarrow I(s, c, x') \end{array}\right.$$

is equivalent by transforming the logical connectors.

$$\left\{\begin{array}{l} I(s, c, x) \wedge G(u, s, c, x) \Rightarrow \exists x'.BAP(u, s, c, x, x') \\ I(s, c, x) \wedge G(u, s, c, x) \Rightarrow \forall x'.(BAP(u, s, c, x, x') \Rightarrow I(s, c, x')) \end{array}\right.$$

is equivalent by conjunction of the goals

$$I(s, c, x) \wedge G(u, s, c, x) \Rightarrow \left\{\begin{array}{l} \exists x'.BAP(u, s, c, x, x') \\ \forall x'.(BAP(u, s, c, x, x') \Rightarrow I(s, c, x')) \end{array}\right.$$

is equivalent to the transformation of logic connectors

$$I(s, c, x) \wedge G(u, s, c, x) \Rightarrow \left\{\begin{array}{l} \exists x'.BAP(u, s, c, x, x') \\ \forall x'.(BAP(u, s, c, x, x') \Rightarrow I(s, c, x')) \end{array}\right.$$

is equivalent to the property of the calculation wp

$$I(s, c, x) \wedge G(u, s, c, x) \Rightarrow [x : |BAP(u, s, c, x, x')]I(s, c, x)$$

is equivalent by transforming the logical connectors

$$I(s,c,x) \Rightarrow (G(u,s,c,x) \Rightarrow [x :|BAP(u,s,c,x,x')]I(s,c,x)$$

is equivalent by internalising the quabtification on u.

$$I(s,c,x) \Rightarrow \forall u.(G(u,s,c,x) \Rightarrow [x :|BAP(u,s,c,x,x')]I(s,c,x)$$

is equivalent to the property of the calculation wp

$$I(s,c,x) \Rightarrow \forall u.[G(u,s,c,x) ==> x :|BAP(u,s,c,x,x')]I(s,c,x)$$

is equivalent to the property of the calculation wp

$$\begin{cases} I(s,c,x) \Rightarrow [@u.G(u,s,c,x) ==> x :|BAP(u,s,c,x,x')]I(s,c,x) \\ I(s,c,x) \Rightarrow (G(u,s,c,x) \Rightarrow \exists x'.BAP(u,s,c,x,x')) \end{cases}$$

is equivalent to the property of the calculation wp

$$\begin{cases} I(s,c,x) \Rightarrow [@u.G(u,s,c,x) ==> x :|BAP(u,s,c,x,x')]I(s,c,x) \\ I(s,c,x) \Rightarrow \mathsf{trm}(\mathsf{e})(x) \end{cases}$$

is equivalent to the property of the calculation wp

$$I(s,c,x) \wedge \mathsf{trm}(\mathsf{e})(x) \Rightarrow [\mathsf{e}](s,c,x)$$

A consequence of this result is to allow a definition of invariant preservation according to two modes of implementation (Atelier-B (Cle 2002) and Rodin (Abrial *et al.* 2010)). We will break down the definition of preservation in the form that separates verification into an induction step and a proof of feasibility. In particular, it defines verification conditions (PO(e)) using these elements as follows.

Definition 13 (verification condition init)

init is an initialisation event and we assume that it is defined as follows: init $\overset{def}{=}$ begin x : $|(initBAP(s,c,x')$ end. The verification condition for init (initialisation) denoted PO(init) is defined by $\begin{cases} (FIS)\ AX(s,c) \vdash \exists x'.initBAP(s,c,x') \\ (INV)\ AX(s,c), initBAP(s,c,x) \vdash I(s,c,x) \end{cases}$ '

Definition 14 (verification condition e)

The verification condition for e (event e) denoted PO(e) is defined by $AX(s,c) \vdash I(x) \wedge \mathsf{trm}(e) \Rightarrow [e]I(s,c,x)$.

$$\begin{cases} (FIS)\ AX(s,c), I(s,c,x), G(u,s,c,x) \vdash \exists x'.BAP(u,s,c,x,x') \\ (INV)\ AX(s,c), I(s,c,x), G(u,s,c,x), BAP(u,s,c,x,x') \vdash I(s,c,x') \end{cases}$$

This definition puts forward two conditions INV and FIS which must be verified and which ensure that $I(s,c,x)$ is preserved. We have specified the elements we are going to use to present the implementation in the Event-B language.

Convention (label for formal texts)

In the definition of the Event-B language, the implementation is based on a labelling of each formal text (axiom, theorem, guard, action) and this labelling is noted $\ell(text)$. This means that the expression $\ell(text) : text$ appears in the modelling text. This labelling makes it possible to refer to elements of the model.

0.3. Contexts and Machines in Event-B

0.3.1. *Modelling sets, constants, axioms and theorems in a context* D

Event-B is organised according to context and machine structures. These structures form the basic structures for the definition of an experimental model. In a separate chapter, we will look at the theory structure implemented in the *Theory* plug-in. Schematically, the relationship between a machine AM; and a context D is expressed by the link sees which is expressed in the machine AM.

Machine AM
SEES
Context D

An Event-B context brings together the definitions of the enduring entities of the system S to be developed and therefore modelled. Now we refer to the document (Métayer and Voisin 2009) presenting the Event-B mathematical language constituting the core of the Event-B language.

```
CONTEXT D
EXTENDS   AD
SETS
  $S_1, \dots S_n$
CONSTANTS
  $C_1, \dots, C_m$
AXIOMS
  $ax_1 : P_1(S_1, \dots S_n, C_1, \dots, C_m)$
  $\dots$
  $ax_p : P_p(S_1, \dots S_n, C_1, \dots, C_m)$
THEOREMS[a]
  $th_1 : Q_1(S_1, \dots S_n, C_1, \dots, C_m)$
  $\dots$
  $th_q : Q_q(S_1, \dots S_n, C_1, \dots, C_m)$
```

– Sets s $(S_1, \dots S_n)$ are declared in the SETS clause.

– Constants c $(C_1, \dots, C_m)$ are declared in the CONSTANTS clause.

– Axioms are listed in the clause AXIOMS clause and define properties of constants.

– Theorems are properties declared in the clause THEOREMS and must be proved from axioms.

– The context defines a logical-mathematical theory which must be consistent.

– The clause EXTENDS extends the context and therefore extends the theory defined by the context of this clause.

– $AX(s, c)$ designates the list of axioms corresponding to the sets s and constants c.

– $TH(s, c)$ designates the list of theoremscorresponding to the sets s and constants c.

Each Event-B expression $expr$ must be well defined, and a verification condition is systematically produced from the text of the property to be proved *expr/WD*; the verification condition for establishing that *expr* is a well-defined theorem, is denoted *expr/WD*. Intuitively, an expression *expr* is well defined, if it obeys certain rules of use, and it is denoted *WD(expr)*. We have given the tables that define the predicate *WD(expr)* according to the syntax of *expr* and according to the classes of expressions that are predicates, relations . . .

From the tables 1, 2 and 3, we can simply establish the verification condition to be verification condition to be proved, denoted expr/WD.

[1] The xlause THEOREMS does not exist in the current verison, but it does make it possible to separate the a xiomes from the theorems. The Rodin platform uses a theorem or axiom indication and produces an italicised version for statements that are theorems, but axioms and theorems are placed under the same AXIOMS clause. This representation may panic the young user.

Expression e	WD(e)
$P \wedge Q, P \Rightarrow Q$	$WD(P) \wedge (P \Rightarrow WD(Q))$
$P \vee Q$	$WD(P) \wedge (WD(P) \wedge (P \vee WD(Q))$
$P \Leftrightarrow Q$	$WD(P) \wedge WD(Q)$
$\neg P$	$WD(P)$
$\forall L.P\ , \exists L.P$	$\forall L.WD(P)$
$\top\ , \bot$	$\top$
$\mathrm{finite}(E)$	$WD(E)$
$partition(E_1, E_2, \ldots, E_n)$	$WD(E_1) \wedge WD(E_2) \wedge \ldots \wedge WD(E_n)$
E op F with op $\in \{=, \neq, \in, \notin, \subset, \not\subset, \subseteq, \not\subseteq\}$	$WD(E) \wedge WD(F)$

Table 1. *WD for predicates*

Expression e	WD(e)
$F(E)$	$\left(\begin{array}{c} WD(E) \wedge WD(F) \\ E \in dom(F) \wedge F \in S ⇸ T \\ F \subseteq S \times T \end{array}\right.$
$E[F], E \mapsto F, E \leftrightarrow F, E F, E F$ $E F, E \rightarrow F, E ⇸ F, E ⤔ F, E \rightarrowtail F$ $E \twoheadrightarrow F, E ⤀ F, E ⤖ F, E \cup F, E \cap F$ $E \backslash F, E \times F, E \otimes F, E \parallel F, E\ ;F, E \circ F$ $E \lhd F, E ⩤ F, E \rhd F, E ⩥ F, E F$ $E..F, E + F, E - F, E * F$	$WD(E) \wedge WD(F)$
E / F, E mod F	$WD(E) \wedge WD(F) \wedge F \neq 0$
E $\widehat{\ }$ F	$WD(E) \wedge 0 \leq E \wedge WD(F) \wedge 0 \leq F$
$-E, E^{-1}, \mathbb{P}(E), \mathbb{P}_1(E)$ $\mathrm{dom}(E), \mathrm{ran}(E), \mathrm{union}(E)$	$WD(E)$
$\mathrm{card}(E)$	$WD(E) \wedge \mathrm{finite}(E)$
$\mathrm{inter}(E)$	$WD(E) \wedge E \neq \varnothing$
$\min(E)$	$WD(E) \wedge E \neq \varnothing \wedge (\exists v.\forall u.u \in E \Rightarrow v \leq u)$
$\max(E)$	$WD(E) \wedge E \neq \varnothing \wedge (\exists v.\forall u.u \in E \Rightarrow v \geq u)$

Table 2. *WD for unary and binary expressions*

Expression e	WD(e)
$\lambda P.Q \vert E$	$\forall \mathcal{F}_Q.WD(P) \wedge (Q \Rightarrow WD(E))$
$\bigcup L.P \vert E$ $\{L.P \vert E\}$	$\forall L.WD(P) \wedge (P \Rightarrow WD(E))$
$\bigcup E \vert P$ $\{E \vert P\}$	$\forall l\mathcal{F}_E.WD(P) \wedge (P \Rightarrow WD(E))$
$\bigcap L.P \vert E$	$\forall L.WD(P) \wedge (P \Rightarrow WD(E))$ $\wedge$ $\exists L.WD(P)$
$\bigcap E \vert P$	$\forall l\mathcal{F}_E.WD(P) \wedge (P \Rightarrow WD(E))$ $\wedge$ $\exists L.WD(P)$
$\mathrm{bool}(P)$	$WD(P)$
$\{E_1, \ldots, E_n\}$	$WD(E_1) \wedge \ldots \wedge WD(E_n)$
$I, \mathbb{Z}, \mathbb{N}, \mathbb{N}_1, pred, succ, \mathrm{BOOL}$ $\mathrm{TRUE}, \mathrm{FALSE}, \varnothing, \mathrm{prj}_1, \mathrm{prj}_2, \mathrm{id}, n$	$\top$

Table 3. *WD for other expressions*

Proof Obligation exp/WD

$AX(s,c) \vdash WD(expr)$

A few examples will illustrate the $WD(expr)$ notation and enable you to understand the verification conditions produced by the tool Rodin.

Exemple 1 (Examples for $WD(expr)$)

– $E1 = \forall k \cdot k \in \mathbb{N} \Rightarrow 2 * s(k) = k * k + k$

We simply apply the transformations of the tables , and :

$WD(\forall k \cdot k \in \mathbb{N} \Rightarrow 2 * s(k) = k * k + k)$

$=$

$\forall k.WD(k \in \mathbb{N} \Rightarrow 2 * s(k) = k * k + k)$

$\forall k.WD(k \in \mathbb{N}) \wedge (k \in \mathbb{N} \Rightarrow WD(2 * s(k) = k * k + k))$

$\forall k.WD(k \in \mathbb{N}) \wedge (k \in \mathbb{N} \Rightarrow WD(2 * s(k)) \wedge WD(k * k + k))$

$\forall k.WD(k \in \mathbb{N}) \wedge (k \in \mathbb{N} \Rightarrow WD(2) \wedge WD(s(k)) \wedge WD(k * k) \wedge WD(k))$

$\forall k.WD(k) \wedge WDS(\mathbb{N}) \wedge (k \in \mathbb{N} \Rightarrow WD(2) \wedge WD(s(k)) \wedge WD(k * k) \wedge WD(k))$

$\forall k.\top \wedge \top \wedge (k \in \mathbb{N} \Rightarrow \top \wedge WD(s(k)) \wedge WD(k * k) \wedge WD(k))$

$\forall k.(k \in \mathbb{N} \Rightarrow WD(s(k)) \wedge WD(k * k) \wedge WD(k))$

$\forall k.(k \in \mathbb{N} \Rightarrow WD(s(k)) \wedge WD(k) \wedge WD(k) \wedge WD(k))$

$\forall k.(k \in \mathbb{N} \Rightarrow WD(s(k)) \wedge \top \wedge \top \wedge \top)$

$\forall k.(k \in \mathbb{N} \Rightarrow WD(s(k)))$

$\forall k.(k \in \mathbb{N} \Rightarrow WD(s) \wedge WD(k) \wedge k \in dom(s) \wedge s \in \mathbb{Z} ⇸ \mathbb{Z} \wedge s \subseteq \mathbb{Z} \times \mathbb{Z}))$

$\forall k.(k \in \mathbb{N} \Rightarrow \top \wedge \top \wedge k \in dom(s) \wedge s \in \mathbb{Z} ⇸ \mathbb{Z} \wedge s \subseteq \mathbb{Z} \times \mathbb{Z}))$

$\forall k.(k \in \mathbb{N} \Rightarrow k \in dom(s) \wedge s \in \mathbb{Z} ⇸ \mathbb{Z} \wedge s \subseteq \mathbb{Z} \times \mathbb{Z}))$

We obtain the condition $WD(E1)$:

$WD(\forall k \cdot k \in \mathbb{N} \Rightarrow 2 * s(k) = k * k + k) = \forall k.(k \in \mathbb{N} \Rightarrow k \in dom(s) \wedge s \in \mathbb{Z} ⇸ \mathbb{Z} \wedge s \subseteq \mathbb{Z} \times \mathbb{Z}))$

– $E2 \overset{def}{=} s(0) = 0$

$WD(s(0) = 0)$

$=$

$WD(s(0)) \wedge WD(0)$

$WD(s(0)) \wedge \top$

$WD(s(0))$

$WD(s) \wedge WD(0) \wedge 0 \in dom(s) \wedge \wedge s \in \mathbb{Z} ⇸ \mathbb{Z} \wedge s \subseteq \mathbb{Z} \times \mathbb{Z})$

$\top \wedge \top \wedge 0 \in dom(s) \wedge \wedge s \in \mathbb{Z} ⇸ \mathbb{Z} \wedge s \subseteq \mathbb{Z} \times \mathbb{Z})$

$0 \in dom(s) \wedge \wedge s \in \mathbb{Z} ⇸ \mathbb{Z} \wedge s \subseteq \mathbb{Z} \times \mathbb{Z})$

We derive the condition $WD(E2)$ using the same rules:

$WD(s(0) = 0) = 0 \in dom(s) \wedge \wedge s \in \mathbb{Z} ⇸ \mathbb{Z} \wedge s \subseteq \mathbb{Z} \times \mathbb{Z})$

The verification condition th/TH is quite simple and sometimes requires interaction with the proof assistant.

Proof Obligation th/TH

$AX(s,c) \vdash th$

To conclude this presentation, we present an example of a context in the field of arithmetic. This allows us to pose a problem that will be used to illustrate the different notations and concepts.

Example - 5 (context for the sum of even or odd natural numbers)

The problem is to calculate the sum $s(n)$ of the natural numbers between 0 and a given integer n, and this sum $s(n)$ is fairly easy to calculate using the formula classically used in elementary maths books: $\forall n.n \in \mathbb{N} \Rightarrow s(n) = \sum_{k=0}^{k=n} = n*(n+1)/2$ or $2*s(n) = n*(n+1)$. The problem is to propose an algorithm which calculates this sum and which respects the correctness property explicitly stated by the relation $2*s(n) = n*(n+2)$. We will also calculate the sum of the even integers $os(n)$ and the sum of the odd integers $es(n)$. The problem is to calculate the functions s, es, os, but we need to define these functions and establish a number of inductive properties.

The first step is to define the axiomatic properties (axm1, axm7) of the necessary sets and constants. The axiom axm7 is an axiomatic expression of induction for the domain of naturals and it will facilitate our proofs by induction that we will have to establish in the section THEOREMS.

AXIOMS

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

$axm2 : s \in \mathbb{N} \rightarrow \mathbb{N} \wedge os \in \mathbb{N} \rightarrow \mathbb{N} \wedge es \in \mathbb{N} \rightarrow \mathbb{N}$

$axm3 : es(0) = 0 \wedge os(0) = 0 \wedge s(0) = 0$

$axm4 : \forall i, l \cdot i \in \mathbb{N} \wedge l \in \mathbb{N} \wedge i = 2*l$
$\Rightarrow s(i+1) = s(i) + i + 1 \wedge es(i+1) = es(i) \wedge os(i+1) = os(i) + i + 1$

$axm5 : \forall i, l \cdot i \in \mathbb{N} \wedge l \in \mathbb{N} \wedge i = 2*l+1$
$\Rightarrow s(i+1) = s(i) + i + 1 \wedge es(i+1) = es(i) + i + 1 \wedge os(i+1) = os(i)$

$axm6 : suc \in \mathbb{N} \rightarrow \mathbb{N} \wedge (\forall i \cdot i \in \mathbb{N} \Rightarrow suc(i) = i+1)$

$axm7 : \forall A \cdot A \subseteq \mathbb{N} \wedge 0 \in A \wedge suc[A] \subseteq A \Rightarrow \mathbb{N} \subseteq A$

THEOREMS

$th1 : \forall i \cdot i \in \mathbb{N} \Rightarrow s(i+1) = s(i) + i + 1$

$th2 : \forall u, v \cdot u \in \mathbb{N} \wedge v \in \mathbb{N} \wedge 2*u = v \Rightarrow u = v/2$

$th3 : \forall k \cdot k \in \mathbb{N} \Rightarrow 2*s(k) = k*k + k$

$th4 : \forall k \cdot k \in \mathbb{N} \Rightarrow s(k) = (k*k+k)/2$

$th5 : \forall k \cdot k \in \mathbb{N} \Rightarrow es(2*k) = 2*s(k)$

$th6 : \forall k \cdot k \in \mathbb{N} \Rightarrow es(2*k+1) = 2*s(k)$

$th7 : \forall k \cdot k \in \mathbb{N} \wedge k \neq 0 \Rightarrow os(2*k) = k*k$

$th8 : \forall k \cdot k \in \mathbb{N} \Rightarrow os(2*k+1) = (k+1)*(k+1)$

Theorems (th1, th5, th6, th7, th8) are based on fairly elementary properties. These theorems are proved using the Rodin tool with the help of the axiom axm7 expressing the induction rule over naturals. They are an important element in linking the inductive definition of sequences and the property expected for this sequence. Thus, the inductive definition of the sequence s and the property are linked as follows: $\forall i.i \in \mathbb{N} \Rightarrow s(i) = i*(i+1)/2$; the role of the inductive definition of s is to give a method of calculation. We will use this method in the illustration of refinement.

0.3.2. *Modelling states and events in an abstract machine* AM

0.3.2.1. *The structure of abstract machine*

In Section **??**, we introduce the notion of an event-based model of a system by the definition 10. Event-B has the abstract machine structure to represent such a state-based model. We give the syntax of an abstract machine AM which uses a context D and which describes a state observed by the variable x. This state variable x is characterised by a set of invariants $I_j(s,c,x), j \in 1..r$ and by a set of safety properties $SAFE_i(s,c,x), i \in 1..t$. We continue the description of this structure in the list of points on the right-hand side below.

```
MACHINE AM
REFINES   M
SEES   D
VARIABLES   $x$
INVARIANTS
  $inv_1 : I_1(s, c, x)$
  ...
  $inv_r : I_r(s, c, x)$
THEOREMS
  $th_1 : SAFE_1(s, c, x)$
  ...
  $th_n : SAFE_n(s, c, x)$
VARIANTS
  $var_1 : varexp_1(s, c, x)$
  ...
  $var_t : varexp_t(s, c, x)$          $i$
EVENTS
  Event initialisation
    begin
      $x : |(Init(s, c, x'))$
    end
  ...
  Event e
    any $u$ where
      $G(u, s, c, x)$
    then
      $x : |BAP(u, s, c, x, x')$
  end
  ...
END
```

– The machine AM is a model describing a set of events modifying the variable x declared in the clause VARIABLES and x is a flexible variable allowing to use notations as x and x'.

– A clause REFINES indicates that the machine AM refines a machine M which is more abstract; however, we will return to this refinement relationship and its role in the development process.

– A particular event defines the initialisation of the variable x according to the relationship $Init(s, c, x')$.

– A clause INVARIANTS describes the inductive invariant that this machine is supposed to respect provided that the associated verification conditions are shown to be valid in the theory induced by the context mentioned by the SEES clause.

– The clause THEOREMS introduces the list of safety properties derived in the theory induced by the context and the invariant; these properties relate to the variables and must be proved valid. It is possible to add theorems about sets and constants; this can help the proofs to be made during the verification process.

– To conclude this description, we would like to add that events can carry very important information for the proof process, in particular for proposing witnesses during event refinement. Furthermore, each event has a status (ordinary, convergent, anticipated) which is important in the production of verification conditions. The clause VARIANTS is linked to events of convergent status.

We will complete this presentation in the remainder of this section and we will add to it when we describe the refinement.

Here is an example of a machine that is important for understanding the difference between an inductive invariant and a safety property.

Example - 6 (abstract machine SAFETY)

```
MACHINE SAFETY
VARIABLES   $x$
INVARIANTS
  $inv : x = -1$
THEOREMS
  $th : x \leq 0$
EVENTS
  Event initialisation
    begin
      $x : |(x' = -1)$
    end
  Event e
      $grd : x \geq 0$
    then
      $act : x : |(x' = x + 1)$
  end
END
```

The variable x is initialised at -1 and, as a result, the event e is never observed. Since $x = -1$, the invariant *inv*:$x = -1$ is true and so is the theorem *th*:$x \leq 0$ which is also true and which is a property of safety property. This machine observes a system with a state variable that remains constant. The use of Rodin and ProB confirms our point. A reading of the short paper by Van Gasteren and Tel (van Gasteren and Tel 1990) clearly reports the difference between *inv* and *th*.

An abstract machine AM is a structure modelling an observed system from the point of view of its state variables but also from the point of view of the domain or domains concerned. Based on the results of section **??** and in particular on the definition 12 we will detail the verification conditions produced to ensure the correctness of this machine. These verification conditions will allow us to exploit the 6 example. The verification conditions for a system S and its invariance and safety properties are as follows:

– $AX(s, c) \vdash \forall x \in \mathsf{D} : Init(s, c, x) \Rightarrow I(s, c, x)$

– $AX(s, c) \vdash \forall x \in \mathsf{D} : I(s, c, x) \Rightarrow A(s, c, x)$

– For any event e of S, $AX(s, c) \vdash \forall x, x' \in \mathsf{D} : I(s, c, x) \wedge BA(e)(s, c, x, x') \Rightarrow I(s, c, x')$

We have explained that we can simplify these conditions by assuming that x does not occur in $AX(s, c)$, as follows:

– $AX(s, c) \vdash Init(s, c, x) \Rightarrow I(s, c, x)$

– $AX(s, c) \vdash I(s, c, x) \Rightarrow A(s, c, x)$

– For any event e of S, $AX(s, c) \vdash I(s, c, x) \wedge BA(e)(s, c, x, x') \Rightarrow I(s, c, x')$

These verification conditions are translated very directly into a list of more basic verification conditions called *proof obligations*. We will proceed in three steps corresponding to the three conditions. From the AM machine, the following notations are derived:

– s: sets *seen* from the context D .

– c: constants *seen* from the context D .

– x: flexible variables define the observed state.

– $AX(s, c)$: axioms *seen* from the context D .

– Let $i \in 1..t$. $TH_i(s, c)$: theorems *seen* from the context D and *located* before the theorem th_i; in particular, $TH_i(s, c)$ is empty.

– $I(s, c, x)$: expression of the invariant of AM defined by $I(s, c, x) \overset{def}{=} I_1(s, c, x) \wedge \ldots \wedge I_r(s, c, x)$

– Let $j \in 1..r$. $I_j(s, c, x)$: jth component of the $I(s, c, x)$ of AM .

– $A(s, c, x)$: expression of safety properties of AM defined as $A(s, c, x) \overset{def}{=} SAFE_1(s, c, x) \wedge \ldots \wedge SAFE_n(s, c, x)$

– Let $i \in 1..t$. $SAFE_i(s, c, x)$: expression of safety properties.

0.3.2.2. *Proof obligation th/TH*

The verification condition $AX(s, c) \vdash I(s, c, x) \Rightarrow A(s, c, x)$ is reduced to a simplified form $AX(s, c), I(s, c, x) \vdash A(s, c, x)$. Then we apply the rule for introducing the conjunction to produce the following conditions for all $i \in 1..t$, $AX(s, c), I(s, c, x) \vdash SAFE_i(s, c, x)$. The proof can then use properties proved in the previous steps and we have noted them $Th_i(s, c)$ and we can thus add these properties to the hypotheses of the sequent and derive the following verification condition for all $i \in 1..t$, $AX(s, c), Th_i(s, c), I(s, c, x) \vdash SAFE_i(s, c, x)$. The following property is therefore demonstrated.

Property 5 (Safety)

If for any $i \in 1..t$, $AX(s, c), Th_i(s, c), I(s, c, x) \vdash SAFE_i(s, c, x)$,
then $AX(s, c) \vdash I(s, c, x) \Rightarrow \bigwedge_{i \in 1..t} SAFE_i(s, c, x)$.

This property is translated into the following verification condition for $i \in 1..t$:

Proof Obligation th$_i$/TH

$$AX(s, c), Th_i(s, c), I(s, c, x) \vdash SAFE_i(s, c, x)$$

An example of a theorem is the case of the SAFETY machine, which is very simple and has a theorem th : $x \leq 0$ and the verification condition is $x \in \mathbb{Z}, x = -1 \vdash x \leq 0$ which is quite trivially deduced. We will return to this example when we have given the verification conditions that ensure the preservation of the invariant $I(s,c,x)$.

0.3.2.3. *Proof obligation INITIALISATION/inv/INV*

The verification condition $AX(s,c) \vdash Init(s,c,x) \Rightarrow I(s,c,x)$ is reduced to the condition $AX(s,c), Init(s,c,x) \vdash I(s,c,x)$. Finally, we can apply a second reduction transformation by deriving the following conditions: for all $i \in 1..r$, $AX(s,c), x \in \mathrm{D}, Init(s,c,x) \vdash I_i(s,c,x)$. The following property can be deduced.

Property 6 (INITIALISATION)

If for any $i \in 1..r$, $AX(s,c), Init(s,c,x) \vdash I_i(s,c,x)$, then $AX(s,c) \vdash Init(s,c,x) \Rightarrow I(s,c,x)$.

Before formulating the verification conditions actually generated by the tool, we recall that the invariant $I(s,c,x)$ is written as a conjunction $I(s,c,x) \equiv \bigwedge_{i \in \{1..r\}} I_i(s,c,x)$. Each element $I_i(s,c,x)$ is labelled inv_i : $I(s,c,x)$. This property is translated into the following verification condition for $i \in 1..r$:

Proof Obligation INITIALISATION/inv$_i$/INV

$AX(s,c), Init(s,c,x) \vdash I_i(s,c,x)$

A second check (FIS) is dedicated to the feasibility of the which is $AX(s,c) \vdash \exists x.initBAP(s,c,x)$. We assume that the expression $initBAP(s,c,x)$ is written in the action part of the event INITIALISATION in the form $act_1 : x_1 : |initBAP_1(s,c,x'_1), \ldots, act_p : x_p : |initBAP_1(s,c,x'_p)$, with $x = x_1 \ldots x_p$ (x is partitioned as p list of variables of x) and $initBAP_i(s,c,x'_i)$ is constructed from $initBAP(s,c,x')$. We therefore assume that $initBAP(s,c,x') \equiv \bigwedge_{i \in \{1..p\}} initBAP(s,c,x'_i$. The decomposition depends on the initial values and one could advise using a normalised form with $p = 1$ but we give the general form. This condition is important to prove as it ensures that the model exists at least in its first state.

Property 7 (feasibility of initial conditions)

Let the following action $act_1 : x_1 : |initBAP_1(s,c,x'_1), \ldots, act_p :: x_p : |initBAP_1(s,c,x'_p)$ defining the conditions for initialising disjoint sub-lists of $x_1, \ldots x_p$ variables whose union constitutes the list x. The verification condition defined by the sequent $AX(s,c) \vdash \exists x.initBAP(s,c,x)$ is equivalent to the list of sequences $AX(s,c) \vdash \exists x_i.initBAP_i(s,c,x_i)$ for $i \in \{1..p\}$.

Proof Obligation INITIALISATION/act$_i$/FIS

$AX(s,c) \vdash \exists x_i.initBAP_i(s,c,x_i)\ i \in \{1..p\}$.

Example - 7 feasibility for the SAFETY machine

In the example SAFETY, the initialization is established, by proving $x \in \mathbb{Z} \vdash \exists x'.x' = -1$ which is in fact trivially true for $x' = -1$.

Example - 8 (feasibility with a witness)

We can also initialize a variable c that must satisfy an invariant of the type invariant of the type $c \in P \to B \wedge c \subseteq a$ where a is a constant modelling a table of access authorisations for people (P) to buildings (B), i.e. $a \subseteq P \times B$. The initialisation event is written very simply in the form $c : |(c' \in P \to B \wedge c' \subseteq a)$ which expresses the fact that the initial value of c must satisfy the invariant but does not explicitly give a value $c0$. The verification condition is then obtained as follows: $a \subseteq P \times B \vdash \exists c'.c' \in P \to B \wedge c' \subseteq a$. The solution is to declare a constant $c0$ which can be used as a witness in the proof.

Example - 9 (example of an assignment)

A final example is the assignment of an expression e to x, i.e. $x := e$. to x, i.e. $x := e$, and in this case the existence is fairly simple to derive simple to derive, but it will undoubtedly be necessary to demonstrate that the expression e makes sense according to the *WD* predicate.

0.3.2.4. *Proof obligations e/I/INV et e/I/FIS*

In the definition 14, we give these two verification conditions which are intended to ensure the preservation of the property $I(s,c,x)$ but also to ensure the feasibility of the event e when the invariant $I(s,c,x)$ is valid. The expressions for these verification conditions are given below:

$$\begin{cases} (FIS)\ AX(s,c), I(s,c,x), G(u,s,c,x) \vdash \exists x'.BAP(u,s,c,x,x') \\ (INV)\ AX(s,c), I(s,c,x), G(u,s,c,x), BAP(u,s,c,x,x') \vdash I(s,c,x') \end{cases}$$

We give an initial example which illustrates the central role of these conditions. We take the SAFETY machine and the event e defined by Event e when $grd : x \geq 0$ then $act : x : |(x' = x+1)$ end and we consider the invariant $I(x) \overset{def}{=} inv : x = -1$. e/inv/INV is defined by the following expression: $I(x), x \geq 0, x' = x+1 \vdash I(x')$ and which is reduced to this expression $x = -1, x \geq 0, x' = x+1 \vdash x' = -1$ which is trivially true since the hypotheses are inconsistent. A first attempt might have been to replace this invariant by $J(x) \overset{def}{=} x \leq 0$ and, in this case, the expression to be proved would have been $J(x), x \geq 0, x' = x+1 \vdash J(x')$ which simplifies to $x \leq 0, x \geq 0, x' = x+1 \vdash x' \leq 0$, then to $x \leq 0, x \geq 0, x = 0, x' = x+1 \vdash 1 \leq 0$! Obviously, it should be stressed that $I(x)$ is an inductive invariant and that $J(x)$ is a weaker invariant property which we will call a safety property or a theorem. So our SAFETY machine is valid since we can prove that $I(x) \vdash J(x)$. This example shows the difference between an inductively invariant property ($x = -1$) and an invariant or always true or safety property ($x \leq 0$); this difference was pointed out by A. J. M. van Gasteren and G. Tel(van Gasteren and Tel 1990). The feasibility condition is obvious and therefore poses no particular problem. We will now describe the verification conditions generated by the Rodin tool and the designation of the verification conditions. A property of the sequent calculus is used to break the verification conditions and thus divide the proof effort.

Property 8 (Introduction/Elimination of connector $\wedge$)

Let be the following sequent $\Gamma \vdash \bigwedge_{i \in \{1..t\}} P_i$.

The proof of this sequence amounts to proving the sequences $\Gamma \vdash P_i$, for all indices $i \in \{1..t\}$.

We apply this property to the case of the invariant $I(s,c,x)$ equivalent to $\bigwedge_{i \in \{1..r\}} I_i(s,c,x)$ and with the labelling of each $I_i(s,c,x)$ assigning it the label inv_i. This property is translated into the following verification condition for $i \in 1..r$:

Proof Obligation e/inv$_i$/INV

$$AX(s,c), I(s,c,x), G(u,s,c,x), BAP(u,s,c,x,x') \vdash I_i(s,c,x')$$

A second checking (FIS) is dedicated to the feasibility of the event e which is $AX(s,c), I(s,c,x), G(u,s,c,x) \vdash \exists x'.BAP(u,s,c,x,x')$.

Proof Obligation e/act/FIS

$$AX(s,c), I(s,c,x), G(u,s,c,x) \vdash \exists x'.BAP(u,s,c,x,x').$$

0.3.2.5. *Proof obligations e/act/WD et INITIALISATION/act/WD*

The verification conditions (INV) and (FIS) ensure that the invariant is preserved and that the events are feasible. However, there are still to be verified in order to guarantee the validity of formalised objects. The division by zero is an element which testifies to this need to avoid the *silly* expressions mentioned by L. Lamport(Lamport 1994, 2002*b*). Writing of formal expressions is therefore perilous and must be framed by the verification conditions WD applied for guards and for actions.

Proof Obligation le/act$_i$/WD

$$AX(s,c), I(s,c,x), G(u,s,c,x) \vdash WF(BAP_i(s,c,x_i)).$$

Proof Obligation INITIALISATION/act$_i$/WD

$$AX(s,c) \vdash WD(initBAP_i(s,c,x_i))$$

We have presented the three kinds of verification conditions : WD, INV and FIS for validating the inductive property of the assertion $I(s,c,x)$. We are continuing our gradual discovery of the elements that make it possible to model a set of events, with a particular focus on events.

0.3.2.5.1. Status of an event

An event e in Event-B is defined by its parameters u, its guards $G(u,s,c,x)$ and its actions $x :| BAP(u,s,c,x,x')$ but it can have one of three possible *status ordinary*, *convergent* or *anticipated* depending on its role. *convergent* or *anticipated* depending on the role it plays in the current machine. In principle, an Event-B machine preserves an invariant and safety properties called theorems. However, it is possible to use a *variant* to model an assertion indicated by a natural integer value or a set value. To understand this point, we need to remember that the induction rules relating to termination are expressed by a sequence of assertions $P_n(x)$ such that for each $n \in \mathbb{N}$, P_{n+1} *leads to* P_n by the observation of at least one *convergent* event. In fact, the problem is to show that the property $existsn \in \mathbb{N}.P_n(x)$ *leads* to a property $Q(x)$ which is linked by the implication $P_0(x) \Rightarrow Q(x)$. Readers familiar with temporal logics will have recognised a property of liveness expressed with the operator *leads to* denoted $\leadsto$ (Owicki and Lamport 1982 ; Méry 1986 ; Chandy and Misra 1988 ; Méry and

Mokkedem 1992 ; Lamport 1994 ; Méry 1999). Thus, a liveness property of the form $P(x) \leadsto Q(x)$ requires the use of an induction rule similar to that in Hoare logic such as:

If
– $I(x)$ is the invariant of the current machine,
– $R_n(x)$ is a sequence of assertions satisfying the property $R_{n+1}(x) \leadsto R_n(x)$, for all $n \in \mathbb{N}$,
– $P(x) \wedge I(x) \Rightarrow \exists n \in \mathbb{N}.R_n(x)$
– $R_0(x) \Rightarrow Q(x)$

then $P(x) \leadsto Q(x)$.

In the case of our machine, the sequence $R_n(x)$ is defined by $R_n(x) eqdef I(x) \wedge variant = n$. In fact, we find the Floyd-Hoare method and a way of showing that a machine converges to a stable state, if the variant decreases it by observing convergent events. We will now define the verification conditions associated with the proof of convergence.

The variant $var : varexp(s, c, x)$ is assumed to be defined for the current machine. Two conditions are required:
– The variant $var : varexp(s, c, x)$ is defined for any convergent event : $AX(s, c), I(s, c, x), G_e(u, s, c, x) \vdash varexp(s, c, x) \in \mathbb{N}$ (NAT)
– The variant $var : varexp(s, c, x)$ varies for any event convergent event : $AX(s, c), I(s, c, x), G_e(u, s, c, x), BAP(u, s, c, x, x') \vdash varexp(s, c, x') < varexp(s, c, x)$ (VAR)

In the case of inclusion relationship, we have the condition verification $AX(s, c), I(s, c, x), G_e(u, s, c, x), BAP(u, s, c, x, x') \vdash varexp(s, c, x') \subset varexp(s, c, x)$ (VAR). To sum up, we have two new verification conditions produced when there are convergent events and variants.

Proof Obligation e/var/NAT

$$AX(s, c), I(x), G_e(u, s, c, x) \vdash varexp(s, c, x) \in \mathbb{N}$$

Proof Obligation e/var/VAR1

$$AX(s, c), I(s, c, x), G_e(u, s, c, x), BAP(u, s, c, x, x') \vdash varexp(s, c, x') < varexp(s, c, x)$$

Proof Obligation e/var/VAR2

$$AX(s, c), I(s, c, x), G_e(u, s, c, x), BAP(u, s, c, x, x') \vdash varexp(s, c, x') \subset varexp(s, c, x)$$

We have devised a small example Fig. 1 which brings together the verification conditions with variant.

Example - 10 (Modelling an adder)

Consider two events evt2 and evt3 which decrease by one. unit respectively the variables x and y initialised respectively at $x0$ and $y0$ two natural integer values. These two events converge and *help* to reach a state where the two variables x and y are zero, leaving only the event evt1 observable. Each time an event decreases x or y, it increases z by one unit. Subject to an implicit assumption of weak global fairness over the events, we can guarantee that z will be worth the sum $x0 + y0$. This example introduces the notion of a *convergent* event, and it remains to

```
CONTEXT $ADDC$

 CONSTANTS $x0\ y0\ f0$

 AXIOMS
  $@axm1\ x0 \in \mathbb{N}$
  $@axm2\ y0 \in \mathbb{N}$
  $@axm3\ f0 \in \mathbb{N} \times \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$
  $@axm4\ \forall\ a, b, c.$
      $a \in \mathbb{N} \land b \in \mathbb{N} \land c \in \mathbb{N}$
      $\land\ a \leq b \land \land\ c \in a..b$
      $\Rightarrow f0(a \mapsto b)(c) = c$
 $end$
```

```
MACHINE $ADDM$ SEES $ADDC$

 VARIABLES $x\ y\ z\ ok$

 INVARIANTS
  $@inv1\ x \in 0..x0$
  $@inv2\ y \in 0..y0$
  $@inv4\ ok \in BOOL$
  $@inv3\ z = x0 - x + y0 - y$
  $@inv5\ ok = TRUE \Rightarrow z = x0 + y0$

 VARIANT $\{v \mid v \in \mathbb{N} \land v \leq x + y\}$
 $y + x$

 EVENTS
  EVENT $INITIALISATION$
     then
     $@act1\ x := x0$
     $@act2\ y := y0$
     $@act3\ z := 0$
     $@act4\ ok := FALSE$
   end
```

```
EVENT $evt1$
    where
   $@grd1\ x = 0 \land y = 0$
   $@grd2\ ok = FALSE$
    then
   $@act1\ ok := TRUE$
  end

$convergent$ EVENT $evt2$
    where
   $@grd1\ ok = FALSE$
   $@grd2\ x > 0$
    then
   $@act1\ z := z + 1$
   $@act2\ x := x - 1$
  end

$convergent$ EVENT $evt3$
    where
   $@grd1\ ok = FALSE$
   $@grd2\ y > 0$
    then
   $@act1\ z := z + 1$
   $@act2\ y := y - 1$
  end

$anticipated$ EVENT $evt4$
    where
   $@grd1\ x > 1 \land y > 1$
    then
   $@act1\ x, y, z, ok :\mid$
       $( z' = x0 - x' + y0 - y'$
       $\land\ x' \in 0..x0 \land y' \in 0..y0$
       $\land\ ( (x' < x \land y' = y)$
       $\lor (y' < y \land x' = x))$
       $\land\ (ok' = TRUE \Rightarrow z' = x0 + y0))$
  end
$end$
```

Figure 1. *Example of convergent and anticipated events*

comment on the status of event *anticipated* to give all the possible forms of events in a basic machine. Thus, the event evt4 is a form of anticipation of what could be added later in this machine. In fact, we are going to present the refinement which will enable us to add and specify elements due to the basic machine. In this case, we could imagine adding events which compute more quickly but which must respect and not disturb convergence.

0.4. Refinement of Event-B machines

The refinement relation is a mechanism which allows models (Event-B machines) to be developed in an incremental and progressive way, starting from a very abstract model and, following a chain of refinements, reaching a concrete expression of the system model. The sequence of machines is an exercise combining modelling and proof. The main idea of refinement is to diffuse the proof effort and to allow a description on several levels of abstraction of a system to be modelled. For example, a communication protocol can be modelled by a one-step communication action, expressing only the expected service. Then we can give details of the more elementary actions which contribute to this service and then we can propose a model which makes it possible to locate the actions. The modelling of the IEEE 1394 protocol (Abrial *et al.* 2003) is an example of development by refinement of a distributed algorithm which is seen as a transformation of a forest into a tree using important properties of a connected non-oriented graph. The chapter**??** is devoted to the incremental development of distributed systems. The diagram below describes the general framework of the use of refinement and the relationship EXTENDS with CM which refines AM and E which is an extension of the initial domain D . We will explain some general properties of refinement and then present the verification conditions for the refinement REFINES.

0.4.1. *Elements on the refinement*

The refinement relationship can refer to several aspects of the relationship between models of the same system. To some extent, a model M1 refines a model M2, if M1 is more accurate than M2 and if it gives more information about the system being modelled. This enrichment of the current model must not call into question what has already been acquired. The refinement of Event-B machines is a practice that makes it possible to make an abstract model closer to the observed system. It proceeds by successive and increasingly precise observations, by detailing elements in the concrete model. Refinement defines a series of levels of increasingly concrete observations. The definition is based on the refinement of events and on the definition of of a glueing invariant establishing the relationship between the variables of the abstract Event-B machine and the concrete concrete variables of the concrete Event-B machine. Before defining Event-B refinement, we recall some elements linking the evolution of B to Event-B . During the first B conference (Habrias 1996) organised by H. Habrias in 1996, J.-R.Abrial (Abrial 1996*b*) gave a demonstration of the transition from B to Event-B while preserving the tools implemented in Atelier-B. A B machine contains operations and an Event-B machine contains events. Michael Leuschel (Leuschel 2021) has written a detailed document outlining the differences between the B notation and the Event-B notation. Our comparison does not take into account the ratings but the intention of the events. The difference between an operation and an event is important when modelling a system and must be used with care and precision. Jean-Raymond Abrial distinguishes between two types of model: *abstract machines* and *abstract systems*. We take up the initial definitions of event refinement and describe the various verification conditions that follow from them. We give the initial definition of the refinement of two events expressed in the context of language B.

Definition 15 (refinement between two events (I))

Let x be the abstract variable (or list of variables) and $I(s,c,x)$ the abstract invariant, y the concrete variable (or list of variables) and $J(s,c,x,y)$ the concrete invariant.

Let c be a concrete event observing the variable y and a an event observing the variable x and preserving $I(s,c,x)$.

Event c refines event a with respect to x, $I(s,c,x)$, y and $J(s,c,x,y)$, if

$$AX(s,c) \vdash I(s,c,x) \wedge J(s,c,x,y) \Rightarrow [c](\neg[a](\neg J(s,c,x,y)))$$

Definition 15 is exactly the same as the definition of the refinement of *operations* in the B language (Abrial 1996*a*), which are replaced by *events*. We remind the reader that an event is observed and an operation is called, but that the refinement relation remains the same relation expressed by the use of wp-calculus. The two events are defined by the following relationships:

$$a \stackrel{def}{=} \left\{ \begin{array}{l} \textsf{any } u \textsf{ where} \\ \qquad G(u,s,c,x) \\ \textsf{then} \\ \qquad x :| ABAP(u,s,c,x,x') \\ \textsf{end} \end{array} \right. \qquad c \stackrel{def}{=} \left\{ \begin{array}{l} \textsf{any } v \textsf{ where} \\ \qquad H(v,s,c,y) \\ \quad \textsf{witness} \\ \qquad u : WP(u,s,c,v,y) \\ \qquad x' : WV(v,s,c,y',x') \\ \textsf{then} \\ \qquad y :| CBAP(v,s,c,y,y') \\ \textsf{end} \end{array} \right.$$

The two events a and c are normalised by a relationship called BA(e)(s,c,x,x'), which simplifies the notations used. The two events a and c are equivalent to events of the following normalized form:

– a is equivalent to begin $x :| (\exists u.G(u,s,c,x) \land ABAP(u,s,c,x,x'))$ end

– c is equivalent to begin $y :| (\exists v.H(v,s,c,y) \land CBAP(v,s,c,y,y'))$ end

From this new formulation, we can deduce a new expression for the refinement of events. The witnesses $u : WP(u,s,c,v,y)$ (*Witness Parameter*) and $x' : WV(v,s,c,y',x')$ (*Witness Variable*) are effective indications for the proof tool in solving an existential quantifier and will be useful at this point. We can conduct the following reasoning.

Explanation (Characterisation of refinement)

(Hypothesis)

(1) $AX(s,c) \vdash I(s,c,x) \land J(s,c,x,y) \Rightarrow [c](\neg[a](\neg J(s,c,x,y)))$

equivalent to

(Definition of $[a]$: $[a](\neg J(s,c,x,y)) \equiv \forall x'.(\exists u.G(u,s,c,x) \land ABAP(u,s,c,x,x')) \Rightarrow \neg J(s,c,x',y)))$

(2) $AX(s,c) \vdash I(s,c,x) \land J(s,c,x,y) \Rightarrow [c](\neg(\forall x'.(\exists u.G(u,s,c,x) \land ABAP(u,s,c,x,x')) \Rightarrow \neg J(s,c,x',y))$

equivalent to
(Transformation by simplification of logical connectives)

(3) $AX(s,c) \vdash I(s,c,x) \land J(s,c,x,y) \Rightarrow [c](\exists x'.(\exists u.G(u,s,c,x) \land ABAP(u,s,c,x,x')) \land J(s,c,x',y))$

equivalent to
(Definition of $[c]$)

(4) $AX(s,c) \vdash I(s,c,x) \land J(s,c,x,y) \Rightarrow (\forall y'.(\exists v.H(v,s,c,x) \land CBAP(v,s,c,y,y')) \Rightarrow ((\exists x'.(\exists u.G(u,s,c,x) \land ABAP(u,s,c,x,x')) \land J(s,c,x',y')))$

equivalent to
(Transformation by quantifier elimination $\forall$)

(5) $AX(s,c) \vdash I(s,c,x) \land J(s,c,x,y) \Rightarrow (\exists v.H(v,s,c,y) \land CBAP(v,s,c,y,y')) \Rightarrow ((\exists x'.(\exists u.G(u,s,c,x) \land ABAP(u,s,c,x,x')) \land J(s,c,x',y')))$

equivalent to
(Transformation by elimination of connector $\land$)

(6) $AX(s,c) \vdash I(s,c,x) \land J(s,c,x,y) \land (\exists v.H(v,s,c,y) \land CBAP(v,s,c,y,y')) \Rightarrow ((\exists x'.(\exists u.G(u,s,c,x) \land ABAP(u,s,c,x,x')) \Rightarrow J(s,c,x',y')))$

equivalent to
(Transformation by elimination of quantifier $\exists$)

(7) $AX(s,c) \vdash I(s,c,x) \wedge J(s,c,x,y) \wedge H(v,s,c,y) \wedge CBAP(v,s,c,y,y') \Rightarrow ((\exists x'.(\exists u.G(u,s,c,x) \wedge ABAP(u,s,c,x,x')) \wedge J(s,c,x',y')))$

equivalent to
(Transformation by property of quantifier $\exists$)

(8) $AX(s,c) \vdash I(s,c,x) \wedge J(s,c,x,y) \wedge H(v,s,c,y) \wedge CBAP(v,s,c,y,y') \Rightarrow ((\exists x'.((\exists u.G(u,s,c,x) \wedge ABAP(u,s,c,x,x')) \wedge J(s,c,x',y')))$

equivalent to
(Transformation by elimination of $\wedge$)

(9)
1) $AX(s,c) \vdash I(s,c,x) \wedge J(s,c,x,y) \wedge H(v,s,c,y) \wedge CBAP(v,s,c,y,y') \Rightarrow (((\exists u.G(u,s,c,x))$
2) $AX(s,c) \vdash I(s,c,x) \wedge J(s,c,x,y) \wedge H(v,s,c,x) \wedge CBAP(v,s,c,y,y') \Rightarrow ((\exists x'.\exists u.(ABAP(u,s,c,x,x')) \wedge J(s,c,x',y')))$

That formal interlude leads to a very simple characterisation of refinement in the form of two conditions: guard strengthening (GRD) and simulation. (SIM).

Property 9 (refinement between events (II))

Let x be the abstract variable (or list of variables) and $I(s,c,x)$ the abstract invariant, y the concrete variable (or list of variables) and $J(s,c,x,y)$ the concrete invariant. the concrete invariant.

Let c be a concrete event observing the variable y and a an event observing the variable x and preserving $I(s,c,x)$.

Event c refines event a with respect to x, $I(s,c,x)$, y and $J(s,c,x,y)$

if, and only if,
1) (GRD) $AX(s,c), I(s,c,x), J(s,c,x,y), H(v,s,c,x), CBAP(v,s,c,y,y') \vdash \exists u.G(u,s,c,x)$
2) (SIM) $\begin{cases} AX(s,c), I(s,c,x), J(s,c,x,y), H(v,s,c,x), CBAP(v,s,c,y,y') \\ \vdash \exists x'.\exists u.ABAP(u,s,c,x,x') \wedge J(s,c,x',y') \end{cases}$

The condition GRD indicates that in the refinement of a by c, the concrete guard ($H(v,s,c,y)$) is stronger than the abstract guard ($\exists u.G(u,s,c,x)$) . In this case, the witness u $u : WP(u,s,c,v,y)$ makes it possible to remove the existential quantification and the condition GRD) is then transformed into the form:

– (GRD-WIT) $AX(s,c), WP(u,s,c,v,y), I(s,c,x), J(s,c,x,y), H(v,s,c,x), CBAP(v,s,c,y,y') \vdash G(u,s,c,x)$

– (SIM-WIT) $\begin{cases} AX(s,c), \\ WP(u,s,c,v,y), WV(v,s,c,y',x'), \\ I(s,c,x), J(s,c,x,y), H(v,s,c,x), CBAP(v,s,c,y,y') \\ \vdash \\ ABAP(u,s,c,x,x') \wedge J(s,c,x',y') \end{cases}$

The witnesses u ($u : WP(u,s,c,v,y)$) and x' ($x' : WV(v,s,c,y',x')$) must exist and it is therefore important to add two new feasibility conditions, noted WFIS, which requires proving the existence of u and x':

– (WFIS(u)) $AX(s,c), I(s,c,x), J(s,c,x,y), H(v,s,c,x), CBAP(v,s,c,y,y') \vdash \exists u.WP(u,s,c,v,y')$

– (WFIS(x')) $AX(s,c), I(s,c,x), J(s,c,x,y), H(v,s,c,x), CBAP(v,s,c,y,y') \vdash \exists x'.WV(v,s,c,y',x')$

We have omitted the verification condition corresponding to the initialization and it is simply obtained by the same transformations as before. We summarize the list of proof obligations required for checking refinement of Event-B models.

Property 10 (verification condition for refinement initialisation)

Let $AInit(s,c,x')$ and $CInit(s,c,y')$ be the initialization predicates for the abstract machine and the concrete machine respectively. The refinement condition for initialization is an adaptation of the refinement relation:

(INIT) $AX(s,c), CInit(s,c,y') \vdash ((\exists x'.(AInit(s,c,x') \wedge J(s,c,x',y')))$

In order to simplify the conditions stated as sequents, we apply two very simple rules for calculating sequents.

Property 11 (simplification of sequents)

1) $AX(s,c), H \vdash P_1 \Rightarrow P_2 \Rightarrow \ldots \Rightarrow P_n \Rightarrow Q$ est equivalent to $AX(s,c), H, P_1, P_2, \ldots, P_n \vdash Q$

2) $AX(s,c), H \vdash P_1 \wedge P_2 \wedge \ldots \wedge P_n$ est equivalent to
   a) $AX(s,c), H \vdash P_1$
   b) $\ldots$
   c) $AX(s,c), H \vdash P_n$

By applying these two rules to our various verification conditions, we obtain the list of verification conditions expressed by sequents. We give a list of the verification conditions to be produced and verified to ensure the refinement of the abstract machine by the concrete machine. The two events are related by the refinement relationship.

Property 12 (Proof obligations for Event-B refinement)

- (INIT) $AX(s,c), CInit(s,c,y') \vdash \exists x'.(AInit(s,c,x') \wedge J(s,c,x',y')$
- (GRD) $AX(s,c), I(s,c,x), J(s,c,x,y), H(v,s,c,x), CBAP(v,s,c,y,y') \vdash \exists u.G(u,s,c,x)$
- (GRD-WIT) $\begin{cases} AX(s,c), I(s,c,x), J(s,c,x,y), H(v,s,c,x), CBAP(v,s,c,y,y'), WP(u,s,c,v,y) \\ \vdash \\ G(u,s,c,x) \end{cases}$
- (SIM) $\begin{cases} AX(s,c), I(s,c,x), J(s,c,x,y), H(v,s,c,x), CBAP(v,s,c,y,y') \\ \vdash \\ \exists x'.\exists u.ABAP(u,s,c,x,x') \wedge J(s,c,x',y') \end{cases}$
- (SIM-WIT) $\begin{cases} AX(s,c), I(s,c,x), J(s,c,x,y), H(v,s,c,x), CBAP(v,s,c,y,y'), \\ WP(u,s,c,v,y), WV(v,s,c,y,x') \\ \vdash \\ ABAP(u,s,c,x,x')) \wedge J(s,c,x',y') \end{cases}$
- (WFIS-P) $AX(s,c) \vdash I(s,c,x) \wedge J(s,c,x,y) \wedge H(v,s,c,x) \wedge CBAP(v,s,c,y,y') \vdash \exists u.WP(u,s,c,v,y)$
- (WFIS-V) $AX(s,c) \vdash I(s,c,x) \wedge J(s,c,x,y) \wedge H(v,s,c,x) \wedge CBAP(v,s,c,y,y') \vdash \exists x'.WV(v,s,c,y,x')$
- (TH) $AX(s,c) \vdash I(s,c,x) \wedge J(s,c,x,y) \vdash SAFE_1(s,c,x,y)$

Now, we describe the concept of refinement in the language itself and illustrate these conditions, to show their role and impact on correct development by construction. From a methodological and proof-theoretical point of view, the explicit witnesses are an effective help for the proof tool.

0.4.2. *Refinement machines in Event-B*

0.4.2.1. *The structure of refinement machine*

A basic abstract machine is described in sub-section 0.3.2 does not include a reference to a more abstract machine and the clause refines clause did not appear in the version presented. We establish a refinement relationship between a machine denoted CM and another machine denoted AM .

```
MACHINE CM   REFINES   AM
SEES  E
VARIABLES  $y$
INVARIANTS
  $jnv_1 : J_1(s,c,x,y)$
  ...
  $jnv_r : J_r(s,c,x,y)$
THEOREMS
  $th_1 : SAFE_1(s,c,x,y)$
  ...
  $th_n : SAFE_n(s,c,x,y)$
VARIANTS
  $var_1 : varexp_1(s,c,y)$
  ...
  $var_t : varexp_t(s,c,y)$
EVENTS
  Event initialisation
    begin
      $y : |(CInit(s,c,y'))$
    end
  ...
  Event c
    refines $a$
    any $v$ where
      $H(v,s,c,y)$
    witness
    $u : WP(u,s,c,v,y)$
    $x' : WV(v,s,c,y',x')$
    then
      $y : |CBAP(v,s,c,y,y')$
  end
  ...
END
```

– The machine CM is a model describing a set of events E(CM) modifying the y variable declared in the clause VARIABLES.

– A clause REFINES indicates that the CM machine refines a AM machine and E(AM) is the set of abstract events in AM .

– A particular event defines the initialisation of variable y according to the relationship $CInit(s,c,y')$.

– The property " Event c refines event a with respect to x, $I(s,c,x)$, y and $J(s,c,x,y)$" is denoted by the expression c refines a. Events a and c are attached to two machines AM and CM ; the invariant attached to each event is the invariant of its machine.

– A clause INVARIANTS describes the inductive invariant invariant $J(s,c,x,y)$ that this machine is assumed to respect provided that the associated verification conditions are shown to be valid in the theory induced by the context E mentioned by the clause SEES. $J(s,c,x,y)$ is the gluing invariant linking the variable y to the variable x.

– The clause THEOREMS introduces the list of safety properties derived in the theory. These properties relate to the variables y and x and must be proved valid. It is possible to add theorems about sets and constants; this can help the proofs to be carried out during the verification process.

– To conclude this description, we would like to add that events can carry very important information for the proof process, in particular for proposing witnesses during event refinement. Furthermore, each event has a status (ordinary, convergent, anticipated) which is important in the production of verification conditions. The clause VARIANTS is linked to events of convergent and anticipated status. The event c (concrete) explicitly refines an event a of the AM machine.

The diagram in figure 2 describes the organisation of Event-B machines and Event-B contexts according to relationship SEES, EXTENDS and REFINES. We have defined the refinement relation for two events a and c and we can propose an extension for machines. We denote refines the refinement relation defined in the previous section for events and it is possible that several concret events in CM refine an abstract event a and it is possible to have a concrete event e which is refining several abstract events and it will be a special case called *merging of abstract events*.

Definition 16 (Event-B machine refinement

The machine CM refines the machine AM , if any event c of CM refines an event a of AM :

$$\forall c.c \in E(CM) \Rightarrow \exists a.a \in E(AM) \wedge c \text{ refines } a.$$

Each machine has an event skip which does not modify the machine's variables. A concrete event c can refine an event skip whose effect is not to modify x in the abstract machine AM. We assume that the invariant of AM is

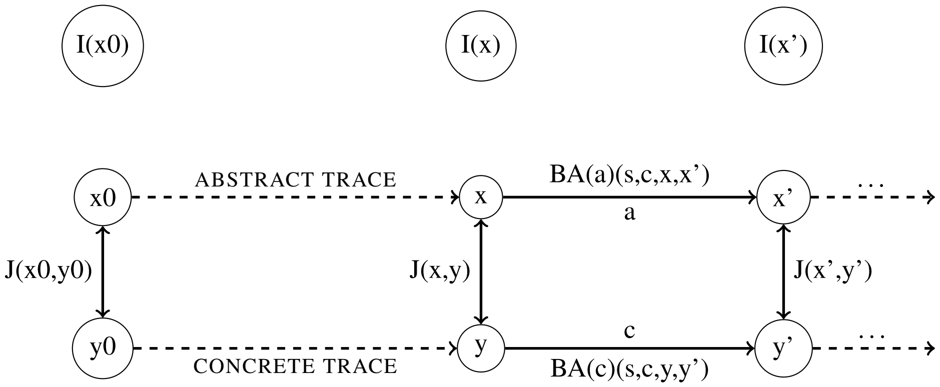


Figure 2. *Refinement between two machines*

$I(s,c,x)$ and that the initialisation of AM is $AInit(s,c,x')$. The philosophy of incremental modelling is based on the need to support proofs, and requires modelling to be carried out in conjunction with proofs. The proof witnesses are used to give properties of the parameter u and the variable x which have disappeared in the machine CM but for which the user must give an expression according to the state of CM . In the diagram below, the schematisation of the refinement relationship shows what is gained. Indeed, $I(s,c,x)$ is not *reproved* but is preserved insofar as the event c does not invalidate $I(s,s,x)$ at the next step.

We give a very simple example which shows what the refinement of two machines can express.

Example - 11 Example of clock

A machine M1 models hours or a machine M1 reports observations of hours and a machine M2 reports hours and minutes. These machines are described in figure 3. This is a very special case of refinement called *superposition* and the proof is fairly straightforward. The event skip is explicitly added in our text but it is left implicit in the Rodin archive.

We will now use the verification conditions set out in the 12 property, to give the identifications of the verification conditions as they appear in the Rodin tool.

0.4.3. *Proof obligations for refinement*

The verification conditions for refinement are identical to those we have already presented, with one important difference: they use abstract elements such as the abstract invariant and the abstract event. Simplification of the conditions is enhanced by reference to explicit witnesses. Finally, the gluing invariant $J(s,c,x,y)$ states a relationship between the abstract variables *x* and the concrete variables *y*, and these two expressions in fact designate a list of variables. It is possible to simplify the verification conditions when there is a variable z in the abstract machine AM which is declared in the concrete machine CM in the form *z*. The translation must take the context into account and we could use *az* for AM and *cz*. The link between the two variables would then be explained by the relationship $az = cz$. The choice is to keep the expression *z* which is both an occurrence of an abstract variable and an occurrence of a concrete variable. In the figure 3, *h* is an abstract variable of M1 and a concrete variable in M2.

Example - 12 Abstract and concrete variables

Figure 4 gives an example of an abstract variable *y* which disappears in the concrete machine in the form *z*. Note the role of the witness *y'* which links y' and *z'* in the initialisation of the machine M2.

```
CONTEXT C

 CONSTANTS H M

 AXIOMS
  @axm1 H = 0..23
  @axm2 M = 0..59
 end
```

```
MACHINE M1 SEES C

 VARIABLES h

 INVARIANTS
  @inv1 h ∈ H

 EVENTS
  EVENT INITIALISATION
     then
   @act1 h :∈ H
  end

  EVENT h1
     where
   @grd1 h < 23
     then
   @act1 h := h + 1
  end

  EVENT h2
     where
   @grd1 h = 23
     then
   @act1 h := 0
  end
 end
```

```
MACHINE M2 REFINES M1 SEES C

 VARIABLES h m

 INVARIANTS
  @inv1 m ∈ M
  theorem @inv2 h ∈ H

 EVENTS
  EVENT INITIALISATION
     then
   @act1 h :∈ H
   @act2 m :∈ M
  end

  EVENT h1m1
     where
   @grd1 h < 23
   @grd2 m < 59
     then
   @act2 m := m + 1
  end

  EVENT h1m2 REFINES h1
     where
   @grd1 h < 23
   @grd2 m = 59
     then
   @act1 h := h + 1
   @act2 m := 0
  end

  EVENT h2m1 REFINES h2
     where
   @grd1 h = 23
   @grd2 m = 59
     then
   @act1 h := 0
   @act2 m := 0
  end

  EVENT h2m2
     where
   @grd1 h = 23
   @grd2 m < 59
     then
   @act1 m := m + 1
  end
 end
```

Figure 3. *Raffinement de machines des heures aux minutes*

MACHINE $M1$

VARIABLES $x\ y$

INVARIANTS
$@inv1\ x+y=100$

EVENTS
EVENT $INITIALISATION$
then
$@act1\ x,y\ \ :|\ \ (x'+y'\ =\ 100)$
end

EVENT $evt1$
then
$@act1\ x,y\ \ \in\ \ |\,(x'+y'=100\ \ \wedge\ \ x'=x+1)$
end

EVENT $evt2$
then
$@act1\ x,y\ \ :|\ \ (x'+y'=100\ \ \wedge\ \ y'=y-1$
end
end

MACHINE $M1$

VARIABLES $x\ y$

INVARIANTS
$@inv1\ x+y=100$

EVENTS
EVENT $INITIALISATION$
then
$@act1\ x,y\ \ :|\ \ (x'+y'\ =\ 100)$
end

EVENT $evt1$
then
$@act1\ x,y\ \ \in\ \ |\,(x'+y'=100\ \ \wedge\ \ x'=x+1)$
end

EVENT $evt2$
then
$@act1\ x,y\ \ :|\ \ (x'+y'=100\ \ \wedge\ \ y'=y-1$
end
end

Figure 4. *Abstract and concrete variable*

0.4.3.1. *Proof obligations for INITIALISATION*

Initialization conditions accompany and complete the diagram **??** and two verification conditions are derived. We make assumptions about the definition of $CInit(s,c,y')$ which can have various decomposed forms. $J(s,c,x,y)$ is a conjunction of properties $inv_i : J_i(s,c,x,y)$ and a witness $x' : WV(s,c,y',x')$ is associated with this event. This simplifies the expression of the condition.

Proof Obligation INITIALISATION/jnv_i/INV

$$AX(s,c), CInit(s,c,y'), WV(s,c,y',x') \vdash AInit(s,c,x') \wedge J_i(s,c,x',y')$$

A second condition is the same as for the basic case and consists in showing that the initial state exists.

Proof Obligation INITIALISATION/act$_i$/WF

$$AX(s,c) \vdash WD(CInit_i(s,c,y_i)$$

Here again we can see the role of the witness, who really helps the work of the proof assistant and which is stated while modelling the system.

0.4.3.2. *Proof obligations for refinement e/l/INV et e/l/FIS*

To keep the rules as simple as possible, we assume that that there are two witnesses:

– a witness for u denoted WP(u,s,c,v,y)

– a witness for x' denoted WV(s,c,y',x')

Existential quantifications are replaced by witnesses and allow us to derive verification conditions for the refinement with a naming specific to Rodin.

The abstract guard $G(s,c,x)$ is labelled $grd: G(s,c,x)$ and note that the concrete guard $H(s,c,y)$ under the conditions of the invariants $I(s,c,x))$ and $J(s,c,x,y)$ implies (and triggers) the abstract guard.

Proof Obligation c/gr/GRD

$$AX(s,c), I(s,c,x), J(s,c,x,y), H(v,s,c,x), WP(u,s,c,v,y) \vdash G(u,s,c,x)$$

The second verification condition also uses the second witness. Our starting point is the following property:

$$AX(s,c), I(s,c,x), J(s,c,x,y), H(v,s,c,x), CBAP(v,s,c,y,y') \vdash \exists x'.(\exists u.ABAP(u,s,c,x,x')) \wedge J(s,c,x',y')$$

which expresses both the preservation of the J invariant and the simulation of the abstract action by the concrete action. If we we obtain the following expression:

$$AX(s,c), I(s,c,x), J(s,c,x,y), H(v,s,c,x), CBAP(v,s,c,y,y'), WP(u,s,c,v,y), WV(v,s,c,y',x') \vdash ABAP(u,s,c,x,x')) \wedge J(s,c,x',y')$$

Then we use the conjunction property by stating two conditions, one for preserving the invariant and the other for simulating it. We assume that the invariant J is labelled inv ($inv: J(s,c,x,y)$) and that the action corresponding to event c is labelled act ($act: x: |ABAP(u,s,c,x,x')$).

Proof Obligation c/inv/INV

$$\begin{array}{l}
AX(s,c),\\
I(s,c,x), J(s,c,x,y),\\
H(v,s,c,x), CBAP(v,s,c,y,y'),\\
WP(u,s,c,v,y), WV(v,s,c,y',x')\\
\qquad\qquad\qquad\qquad \vdash J(s,c,x',y')
\end{array}$$

Proof Obligation c/act/SIM

$$\begin{array}{l}
AX(s,c),\\
I(s,c,x), J(s,c,x,y),\\
H(v,s,c,x), CBAP(v,s,c,y,y'),\\
WP(u,s,c,v,y), WV(v,s,c,y',x')\\
\qquad\qquad\qquad\qquad \vdash ABAP(u,s,c,x,x')
\end{array}$$

The simplification provided by the witnesses must be guaranteed and it must be shown that the conditions defined by the axioms and the two invariants guarantee the existence of the two witnesses. To do this, it is sufficient to prove that witness u and witness x' exist.

Proof Obligation c/u/WFIS

$$AX(s,c), I(s,c,x), J(s,c,x,y) \wedge H(v,s,c,x) \vdash \exists u.WP(u,s,c,v,y)$$

Proof Obligation c/x'/WFIS

$$AX(s,c), I(s,c,x), J(s,c,x,y), H(v,s,c,x) \wedge CBAP(v,s,c,y,y') \vdash \exists x'.WV(v,s,c,y',x')$$

We have detailed the verification conditions that are produced by the Rodin platform and that must be proven by the proof tool. In fact, it is important to have in mind how the proof obligations are generated and what they do mean.

0.4.3.3. *Additional proof obligations*

The verification conditions are completed by the conditions associated with the events *convergent* or *anticipated*. Simply add the invariant $J(s,c,s,y)$ and we assume that $H(v,s,c,y)$ is the guard of the concrete event c.

Proof Obligation c/var/NAT

$$AX(s,c), I(s,c,x), J(s,c,x,y), H(v,s,c,y) \vdash varexp(s,c,y) \in \mathbb{N}$$

Proof Obligation c/var/VAR

$$AX(s,c), I(s,c,x), J(s,c,x,y), H(v,s,c,y), BAP(v,s,c,y,y') \vdash varexp(s,c,y') < varexp(s,c,y)$$

Finally, there is one last condition associated with the proof of theorems. th is the label of the theorem $SAFE(s,c,x,y)$.

Proof Obligation th/TH

$$AX(s,c), I(s,c,x), J(s,c,x,y) \vdash SAFE(s,c,x,y)$$

The verification conditions are expressed as sequents and are subjected in the Rodin platform to the various proof tools available, sometimes with decisive human interaction. These additional verification conditions are necessary to play a role in establishing the observation of a convergent (or helpful) event. These events are reminiscent of the help functions in Pnueli (Manna and Pnueli 1984) or the critical actions of Méry(Méry 1986). Their role is clearer when we consider Pnueli's induction rules or Lamport's rules for TLA, which is even more complete with fairness hypotheses. We can also mention the rule of progression by observation of a given event in the case of UNITY (Chandy and Misra 1988). For anticipated events, the idea is to develop progressively but to be able to add events across the board that won't call everything into question. In fact, when a machine is defined either by refinement or by creation, the model frame fixes the variables which cannot be modified subsequently, since the events which will be added in the next refinement must not invalidate the invariant of the refined machine, and in the case of new events, they must refine skip, i.e. not modify the variables of the refined machine. To a certain extent, it is important that all the events are introduced at the current level, but we don't know them all and the idea is to use an anticipated event which is still imprecise but which maintains the current invariant. This is the case when we are going to develop a sequential algorithm from a pre.post specification and it is quite simple to anticipate a calculation loop before delivering the result satisfying the postcondition. In this way, the anticipated event makes it possible to express that something is happening before the final computation event is observed.

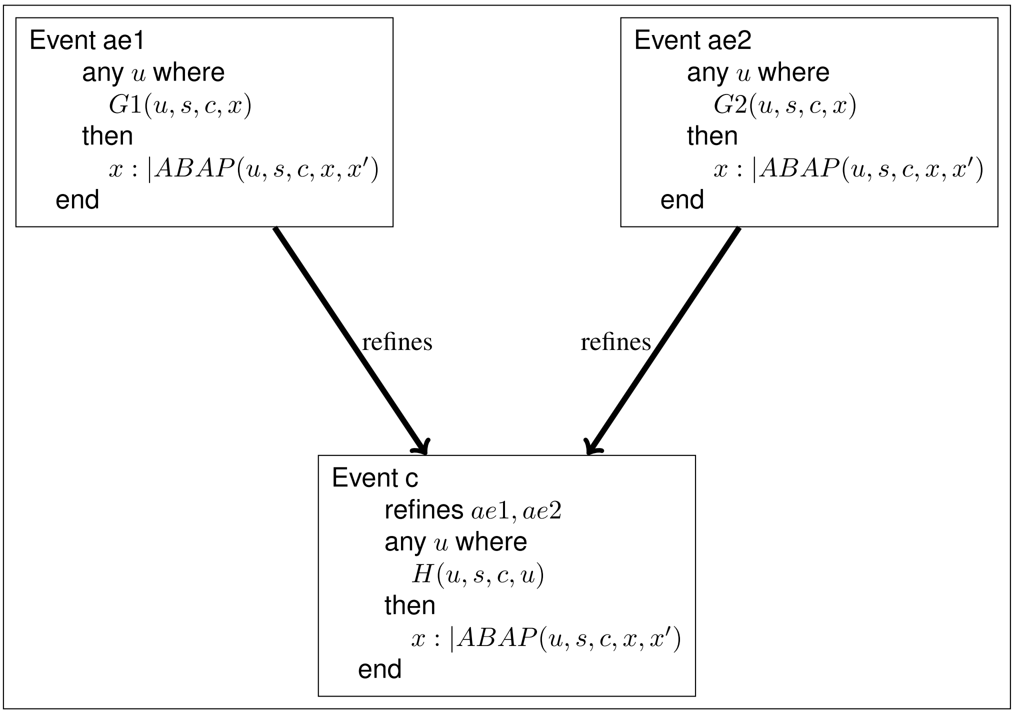
0.4.3.4. *Fusion of events*

In the refinement of a machine AM by CM , the definition 16 is reduced to this expression $\forall c.c \in E(CM) \Rightarrow \exists a.a \in E(AM).e$ refines a. Every concrete event c in CM corresponds ,to an abstract event a and is linked by the relation refines and in a way, we could understand that the number of abstract events is less than the number of concrete events. This is not the case and it is possible to to reduce the number of events per refinement by using event merging with very strong assumptions. Thus, by merging two abstract events a1 and a2, we obtain a concrete concrete event ca1+a2 which refines each of the two events, but the condition is that the action of two abstract be syntactically identical. The condition is strong but allows us to merge and simplify the concrete models. Merging two events is an important mechanism for completing the Event-B refinement. This possibility requires strong conditions on the actions, which must be identical for all three events. As shown in the diagram in figure **??**, the guards $G1$ and $G2$ are reinforced by the guard H and the verification condition is very simply the expression of this reinforcement.

Proof Obligation c/MRG

$$AX(s,c), I(s,c,x), H(v,s,c,y) \vdash G_1(s,c,x) \vee G_2(s,c,x)$$

The conditions for applying this merging are very restrictive, but we will give an example of its application. In a case study, we used this merge to reduce a set of events to a single event. As a result, it can be seen that during refinement either the number of events is increased or reduced, depending on the case.



```
Event ae1
    any $u$ where
      $G1(u,s,c,x)$
    then
      $x :| ABAP(u,s,c,x,x')$
  end
```

```
Event ae2
    any $u$ where
      $G2(u,s,c,x)$
    then
      $x :| ABAP(u,s,c,x,x')$
  end
```

```
Event c
    refines $ae1, ae2$
    any $u$ where
      $H(u,s,c,u)$
    then
      $x :| ABAP(u,s,c,x,x')$
  end
```

Figure 5. *Fusion de deux Ã©vÃ©nements*

```
CONTEXT  MRG0

 CONSTANTS  a b

 AXIOMS
  @axm1 a ∈ ℤ ∧ b ∈ ℤ
 end
```

We illustrate this rule by calculating the maximum of two numbers. The context MRG0 defines the integer values a and b, and we will ensure that sup contains the larger of the two values. The larger of the two values is the one we're looking for.

This machine compares two values and assigns the value of the maximum to sup. The variable ok controls this machine. When ok is TRUE, sup contains the maximum.

```
MACHINE $MRG1$ SEES $MRG0$

 VARIABLES $i\ j\ ok\ sup$

 INVARIANTS
  $@inv1\ i \in \mathbb{Z} \land j \in \mathbb{Z} \land ok \in BOOL \land sup \in \mathbb{Z}$
  $@inv2\ i = a \land j = b$
  $@inv3\ ok = TRUE \Rightarrow sup \geq i \land sup \geq j \land sup \in \{a, b\}$

 EVENTS
   EVENT $INITIALISATION$
      then
    $@act1\ i := a$
    $@act2\ j := b$
    $@act3\ ok := FALSE$
    $@act4\ sup :\in \mathbb{Z}$
   end

   EVENT $e1$
      where
    $@grd1\ i < j$
    $@grd2\ ok = FALSE$
      then
    $@act1\ sup := j$
    $@act2\ ok := TRUE$
   end

   EVENT $e2$
      where
    $@grd1\ i \geq j$
    $@grd2\ ok = FALSE$
      then
    $@act1\ sup := i$
    $@act2\ ok := TRUE$
   end
 $end$
```

Before applying the MRG rule, we must construct a refinement of this machine that satisfies the application conditions. This means refining e1 and e2 so that the action is identical for each event. The MRG2 machine refines the MRG1 machine, and the actions of the two events are identical.

We can apply the rule for merging the two events and construct an event merge(e1;e2) which corresponds to a conditional instruction. Figure 6 illustrates the application of this rule and gives a view of the events. The proof is fairly easy, since the H condition reduces to TRUE.

```
MACHINE $MRG2$
  REFINES $MRG1$
  SEES $MRG0$

 VARIABLES $i\ j\ ok\ sup$

 EVENTS
  EVENT $INITIALISATION$
      then
    $@act1\ i\ :=\ a$
    $@act2\ j\ :=\ b$
    $@act3\ ok\ :=\ FALSE$
    $@act4\ sup :\in\ \mathbb{Z}$
   end

  EVENT $e1$ REFINES $e1$
      where
    $@grd1\ i\ <\ j$
    $@grd2\ ok = FALSE$
      then
    $@act1\ sup, ok\ \ :|\ \ ($
         $(ok = FALSE$
          $\wedge\ (i\ <\ j\ \ \Rightarrow\ \ sup' = j)$
          $\wedge\ (i \geq j\ \ \Rightarrow\ \ sup' = i)$
          $\wedge\ \ ok' = TRUE))$
   end

  EVENT $e2$ REFINES $e2$
      where
    $@grd1\ i \geq j$
    $@grd2\ ok = FALSE$
      then
    $@act1\ sup, ok\ \ :|$
         $(ok = FALSE$
          $\wedge\ (i\ <\ j\ \ \Rightarrow\ \ sup' = j)$
          $\wedge\ (i \geq j\ \ \Rightarrow\ \ sup' = i)$
          $\wedge\ \ ok' = TRUE)$
   end
 $end$
```

```
MACHINE $MRG3$ REFINES $MRG2$
  SEES $MRG0$

 VARIABLES $i\ j\ ok\ sup$

 INVARIANTS
 $theorem\ @inv1$
  $ok = TRUE\ \ \Rightarrow\ \ sup\ \geq i\ \ \wedge\ \ sup\ \geq j\ \ \wedge\ \ sup\ \ \in\ \ \{a, b\}$
  $theorem\ @inv2\ i = a\ \ \wedge\ \ j = b$

 EVENTS
  EVENT $INITIALISATION$
      then
    $@act1\ i\ :=\ a$
    $@act2\ j\ :=\ b$
    $@act3\ ok\ :=\ FALSE$
    $@act4\ sup :\in\ \mathbb{Z}$
   end

  EVENT $merge(e1, e2)$ REFINES $e1\ e2$
      where
    $@grd2\ ok = FALSE$
      then
    $@act1\ sup, ok\ \ :|\ \ ($
        $ok = FALSE\ \wedge$
        $(i\ <\ j\ \ \Rightarrow\ \ sup' = j)\ \wedge$
        $(i \geq j\ \ \Rightarrow\ \ sup' = i)\ \wedge$
        $ok' = TRUE)$
   end
 $end$
```

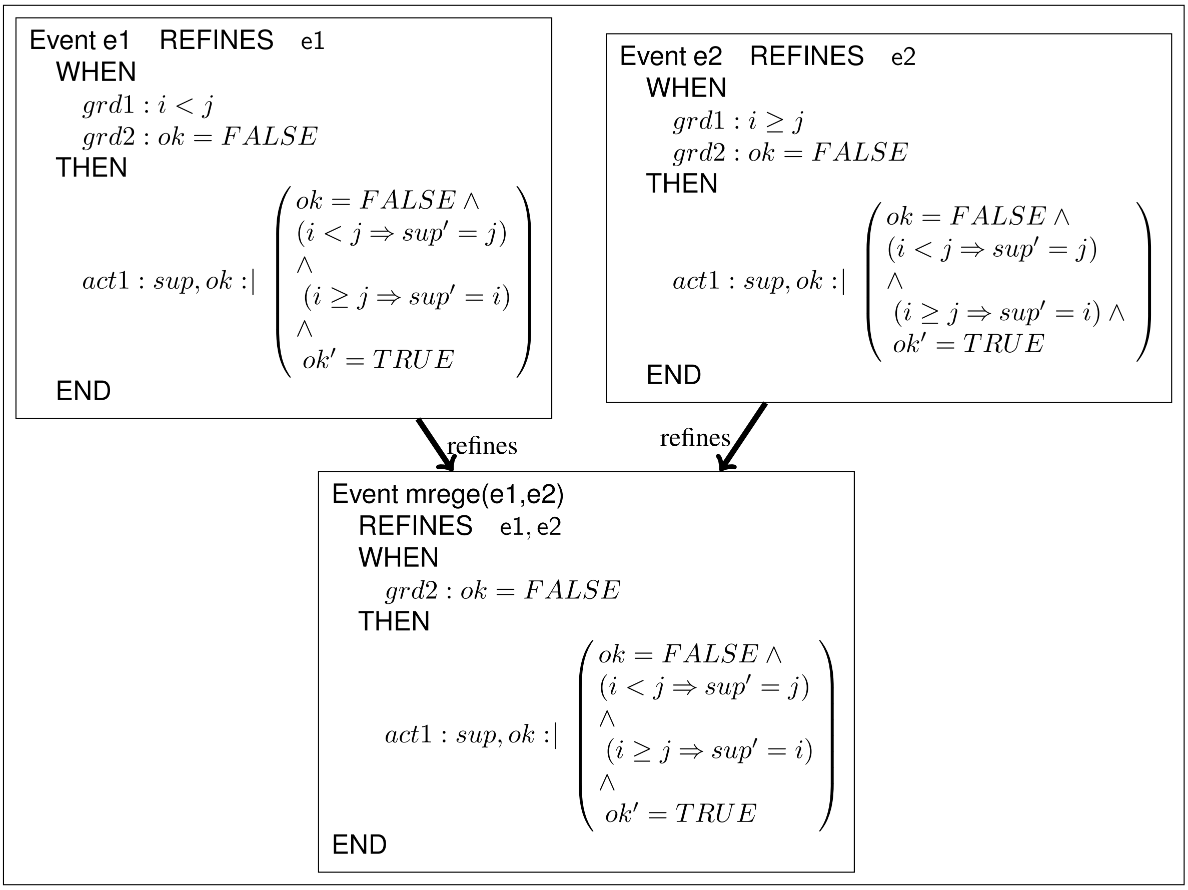
This merging rule will be used in the chapter 0.5.2 when developing sequential algorithms. Refinement is an operation that allows new events to be introduced as well as reinforcing existing events, and in the case of this rule, it allows the number of events to be reduced.

0.5. Summary

We take up the elements of the Event-B language by developing a simplified abacus system, in order to illustrate the modelling language, refinement, ordinary, anticipated and convergent events, witnesses, invariants and safety properties. Other examples will be developed in the following chapters and will illustrate the modelling possibilities of Event-B .

0.5.1. *Playing with Event-B*

We are dealing with a simple example to illustrate the various elements we have already presented, but it is clear that the other chapters will be illustrations of the use of the Event-B language with the Rodin platform or the Atelier-B platform. In fact, we have an idea of a simple system for calculating the *addition* function, and this



Event e1 REFINES e1
WHEN
$grd1 : i < j$
$grd2 : ok = FALSE$
THEN

$$act1 : sup, ok :| \begin{pmatrix} ok = FALSE \land \\ (i < j \Rightarrow sup' = j) \\ \land \\ (i \geq j \Rightarrow sup' = i) \\ \land \\ ok' = TRUE \end{pmatrix}$$

END

Event e2 REFINES e2
WHEN
$grd1 : i \geq j$
$grd2 : ok = FALSE$
THEN

$$act1 : sup, ok :| \begin{pmatrix} ok = FALSE \land \\ (i < j \Rightarrow sup' = j) \\ \land \\ (i \geq j \Rightarrow sup' = i) \land \\ ok' = TRUE \end{pmatrix}$$

END

Event mrege(e1,e2)
REFINES e1, e2
WHEN
$grd2 : ok = FALSE$
THEN

$$act1 : sup, ok :| \begin{pmatrix} ok = FALSE \land \\ (i < j \Rightarrow sup' = j) \\ \land \\ (i \geq j \Rightarrow sup' = i) \\ \land \\ ok' = TRUE \end{pmatrix}$$

END

Figure 6. *Fusion of two events e1 and e2*

method is the one that the school teacher taught, when the author was 6 years old and discovering numbers and calculations. Figure 7 shows contexts and machines used to describe the rules for using the *abacus*. The abacus can be used either by moving balls from top to bottom in one move (rule 1), or by moving them one after the other (rule 2).The abacus uses potential energy, as the balls will descend very simply once they have been released. Initialization is done quite simply by inverting the top and bottom. Formally, the problem to solve can be defined as follow using a contract-based notation.

variables x, y, r, ok

$$\text{requires} \begin{cases} x_0 \in \mathbb{N} \\ y_0 \in \mathbb{N} \\ r_0 \in \mathbb{Z} \\ ok_0 \in \mathbb{BOOL} \end{cases}$$

$$\text{ensures} \begin{pmatrix} r_f = x_0 + y_0 \\ ok_f = TRUE \end{pmatrix}$$

The contract expresses a relationship between the initial values of x and y, denoted x0 and y0, and the final value of r denoted r_f. This relationship simply expresses that the final value of the variable r denoted r_f is the sum $x0 + y0$. Obviously, we use the mathematical operator $+$ and we aim to construct, by refinement, a set of events modelling the movements of the abacus and calculating the operator $+$.

Two contexts are mentioned in the figure 7 namely C0 and C1. C0 is the context for defining the two integer values a and b used for the computation. They are the *inputs* for short.

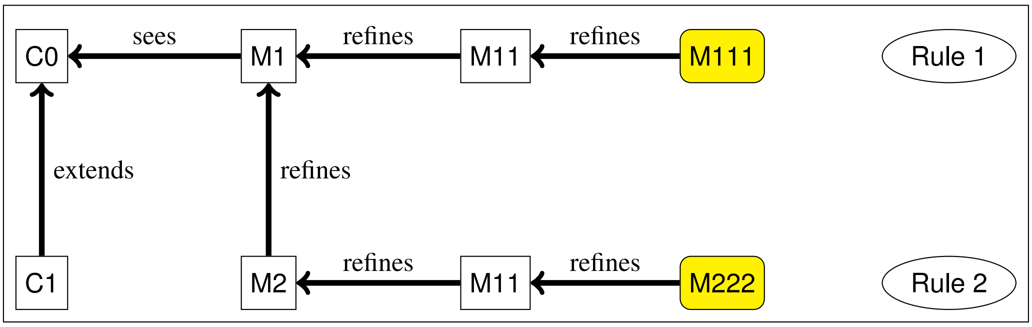


Figure 7. *Organisation of refinements for ABACUS*

a and b are two natural numbers and are defined in context C0. In context C1, we define balls of the abacus and the two sets of balls representing sets with cardinalities corresponding to a and b.

CONTEXT $C0$

CONSTANTS $a\ b$

AXIOMS
$@axm1:\ a \in \mathbb{N}$
$@axm2:\ b \in \mathbb{N}$
end

CONTEXT $C1$ EXTENDS $C0$

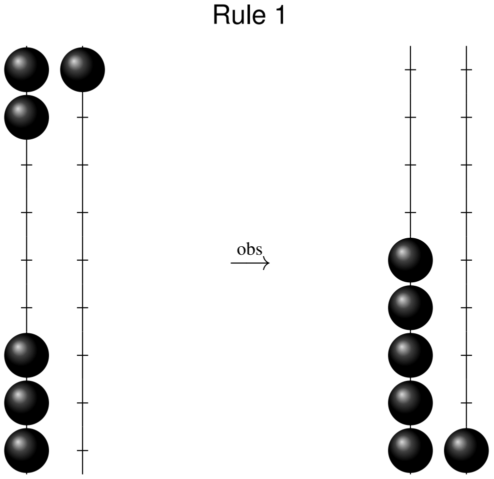
sets B

CONSTANTS *seta setb ball*

AXIOMS
$@axm1:\ B \neq \varnothing \wedge finite(B)$
$@axm5:\ seta \subseteq B \wedge card(seta) = a$
$@axm2:\ setb \subseteq B \wedge card(setb) = b$
$@axm3:\ seta \cap setb = \varnothing$
$theorem\ @axm4:\ finite(seta) \wedge finite(setb)$
$@axm6:\ ball \in B$
$@axm7:\ \forall\ u, v\ .$
$\quad u \subseteq B \wedge v \in B \wedge v \notin u$
$\quad \Rightarrow$
$\quad card(u \quad \{v\}) = card(u) - 1$
end

The aim of this modelling is to show the link between the mathematical function + and its calculation using two calculation rules, using variable refinement and abstraction. The representation of a number n on the abacus is a set of n balls. Figure 7 gives contexts and machines modelling the two rules, and we are detailing the different components.

0.5.1.1. *Rule 1*



The event obs triggers the mechanism for passing the two top balls onto the three bottom balls. The result is obtained by counting the number of lower balls. We could use the gaseous method, which would lead us to observe that the lower balls rise Whatever process is used, it is completed when the right ball is at the bottom (resp. at the top). Rule 1 for using the abacus is to move the balls from the top to the bottom and to update an indicator showing that the rule has been applied once and only this rule is applied.

The machine M1 with the context C0 models the contract associated with the addition function. The event calling_a_function models the relationship between the values before and after this event. Then the machine M11 refines M1 by introducing a modelling of the variables by sets of balls setr and setok initially containing a number of balls corresponding to the value of the variables setr and setok. The refinement is then continued by hiding the variables r and ok and the event obs models that of the figure on the left.

The machine M1 is simply expressing the contract as an event calling_a_function. After observing the event calling_a_function, the variable ok is set to TRUE and r is set to $a+b$. The event computing is *anticipating* a hidden computation process. The machine M1 is refined into a refinement machine M1. Two new variables setr and setok are introduced and the invariant expresses a relationship between ok,r, setr and setok.

```
MACHINE $M1$ SEES $C1$

 VARIABLES $r\ ok$

 INVARIANTS
  $@inv1\ r \in \mathbb{Z}$
  $@inv2\ ok \in BOOL$
  $@inv3\ ok = TRUE \Rightarrow r = a+b$

 EVENTS
  EVENT $INITIALISATION$
     then
    $@act3 :\ ok := FALSE$
    $@act4 :\ r :\in \mathbb{Z}$
  $end$

  EVENT $calling\_a\_function$
     where
    $@grd1 :\ ok = FALSE$
     then
    $@act1 :\ ok := TRUE$
    $@act2 :\ r := a+b$
  $end$

  $anticipated$ EVENT $computing$
     then
    $@act1 :\ r, ok :|$
        $(ok' \in BOOL \land r' \in \mathbb{Z}$
        $\land (ok' = TRUE \Rightarrow r' = a+b))$
  $end$
 $end$
```

```
MACHINE $M11$ REFINES $M1$
 SEES $C1$
 VARIABLES $r\ ok\ setr\ setok$
 INVARIANTS
  $@inv1\ setr \subseteq B \land setok \subseteq B$
  $@inv5\ setok = \{ball\}$
     $\Rightarrow setr = seta \cup setb$
  $@inv2\ setok = \{ball\} \Rightarrow ok = TRUE$
  $@inv3\ ok = TRUE \Rightarrow setok = \{ball\}$
  $@inv6\ card(setr) = r$
  $@inv7\ setok \subseteq \{ball\}$

 EVENTS
  EVENT $INITIALISATION$
     then
    $@act4\ r, setr, ok, setok :|$
       $(r' = 0 \land ok' = FALSE$
       $\land setok' = \varnothing \land setr' = \varnothing)$
   end

  EVENT $obs$
   REFINES $calling\_a\_function$
     where
    $@grd1\ ok = FALSE$
    $@grd2\ setok = \varnothing$
     then
    $@act1\ ok := TRUE$
    $@act2\ r := a+b$
    $@act3\ setr := seta \cup setb$
    $@act4\ setok := \{ball\}$
   end

  $anticipated$ EVENT $computing$
   REFINES $computing$
     then
    $@act1\ r, ok, setr, setok :|$
       $(ok' \in BOOL \land r' \in \mathbb{Z}$
       $\land setok' \subseteq \{ball\}$
       $\land (setok' = \{ball\}$
       $\Rightarrow ok' = TRUE$
       $\land setr' = seta \cup setb$
       $\land r' = a+b)$
       $\land (ok' = TRUE$
       $\Rightarrow r' = a+b$
       $\land setok' = \{ball\}$
       $\land setr' = seta \cup setb)$
       $\land card(setr') = r')$
   end
 $end$
```

The refinement machine M11 was used to introduce two new variables to model the ranks of the abacus. It is an intermediate machine and is refined into a refinement machine M111 where the variables ok and r are hidden. The event obs models the moving of the balls from the top to the bottom in a single move. This machine illustrates the use of witnesses in the INITIALISATION event. The event obs is simulating the rule 1.

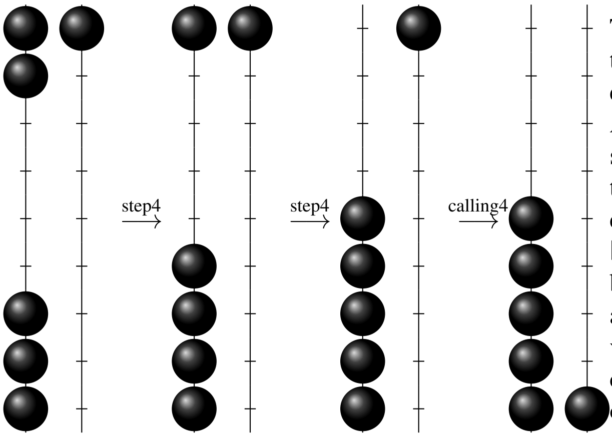
```
MACHINE $M111$ REFINES $M11$ SEES $C1$

 VARIABLES $setr\ setok$

 INVARIANTS
  $theorem\ @inv1\ setok = \{ball\} \Rightarrow setr = seta \cup setb$

 EVENTS
   EVENT $INITIALISATION$
    $with$
      $@r'\ r' = 0$
      $@ok'\ ok' = FALSE$
        then
      $@act4\ setr, setok\ :|\ (setok' = \varnothing \land$
          $setr' = \varnothing)$
    end

   EVENT $obs$ REFINES $obs$
        where
      $@grd2\ setok = \varnothing$
        then
      $@act3\ setr := seta \cup setb$
      $@act4\ setok := \{ball\}$
    end

  $anticipated$ EVENT $computing$ REFINES $computing$
    $with$
      $@r'\ r' = card(setr')$
      $@ok'\ (setok' = \{ball\} \Rightarrow ok' = TRUE)$
          $\land\ (setok' = \varnothing \Rightarrow ok' = FALSE)$
        then
      $@act1\ setr, setok\ :|\ (setok' \subseteq \{ball\} \land (setok' = \{ball\} \Rightarrow setr' = seta \cup setb))$
    end
 $end$
```

Rule 1 calculates the addition in one move and the refinement expresses that the + function is calculated according to this rule application.

The computing event is undoubtedly one that may seem like too much, and it could disappear at this level of refinement. In fact, it models the fact that there may be non-visible calculations that prepare the final result. In fact, it anticipates intermediate calculations which are not visible but which could exist. In all cases, the invariants remain verified.

Point d'arrêt

0.5.1.2. *Rule 2*



The M1 machine is refined by the M2 machine and introduces two new variables, x and y, which allow an iteration controlled by the value of y to be implemented. A new event step2 defines the computation or iteration step. The invariant $x + y = a + b$ expresses the maintenance of the number of balls. The variant y is used to express the termination of the process. Then the machine M22 introduces the balls and the event step3 observes both the movement of a ball and the updating of the variables x and y. Then a final refinement M222 hides the variables x,y and ok and we find the event step3. We recover the event step4, which models the movement of one ball at a time.

```
MACHINE $M2$ REFINES $M1$ SEES $C0$
 VARIABLES $x\ y\ r\ ok$
 INVARIANTS
  $@inv1\ x \in \mathbb{N}$
  $@inv2\ y \in \mathbb{N}$
  $@inv3\ x + y = a + b$
  $theorem\ @inv4\ ok = TRUE \Rightarrow r = a + b$
  VARIANT $y$
 EVENTS
  EVENT $INITIALISATION$
     then
    $@act1\ ok := FALSE$
    $@act2\ r :\in \mathbb{Z}$
    $@act3\ x := a$
    $@act4\ y := b$
   end
  EVENT $calling2$ REFINES $computing$
     where
    $@grd1\ ok = FALSE$
    $@grd2\ y = 0$
     then
    $@act1\ ok := TRUE$
    $@act2\ r := x$
   end
 $convergent$ EVENT $step2$ REFINES $computing$
     where
    $@grd1\ ok = FALSE$
    $@grd2\ y \neq 0$
     then
    $@act1\ x := x + 1$
    $@act2\ y := y - 1$
   end
 $end$
```

The refinement introduces two new variables x and y which are used to introduce the iterative process starting with y containing b and ending when y contains 0. The property of this process is that there is only one event which observes the decay of y and which is a convergent event.

The ok variable is used to express that the calculation process is complete when the value of y is 0. The end of the process is detected by the value of y and convergence is expressed by the variant y which decreases strictly when the event step2 is observed.

The proofs are simple and the property $x + y = a + b$ is inductive.

The refinement of M2 into M22 amounts to materialising the iterative process with balls and the M22 machine introduces four new variables setr, setok like M22 and setx and sety for the two sides of the abacus. The preservation of $x + y = a + b$ is ensured by the fact that no one can add or remove balls. The question arises of adding demonic events whose role is to allow balls to be lost or added: $setx \cup sety = seta \cup setb$.

A final refinement of M222 consists in hiding the variables r,x,y,ok, thus obtaining an iterative process which can only apply rule 2.

```
MACHINE $M22$ REFINES $M2$ SEES $C1$

 VARIABLES $r\ ok\ setx\ sety\ x\ y\ setr\ setok$

 INVARIANTS
  $@inv1\ setx \subseteq B$
  $@inv2\ sety \subseteq B$
  $@inv3\ setr \subseteq B \wedge setok \subseteq B \wedge setok \subseteq \{ball\}$
  $@inv4\ setx \cap sety = \varnothing \wedge setx \cup sety = seta \cup setb$
  $@inv5\ setok = \{ball\} \Rightarrow sety = \varnothing \wedge setr = seta \cup setb$
  $@inv6\ (setok = \varnothing \Rightarrow ok = FALSE) \wedge (ok = FALSE \Rightarrow setok = \varnothing)$
  $@inv7\ (setok = \{ball\} \Rightarrow ok = TRUE) \wedge (ok = TRUE \Rightarrow setok = \{ball\})$
  $@inv8\ x = card(setx) \wedge y = card(sety)$

 EVENTS
   EVENT $INITIALISATION$
       then
     $@act1\ ok := FALSE$
     $@act2\ r :\in \mathbb{Z}$
     $@act3\ x, y, setx, sety, setr, setok :|$
     $(setok' = \varnothing \wedge setr' \subseteq B \wedge setx' = seta \wedge sety' = setb \wedge x' = a \wedge y' = b)$
   end

   EVENT $calling3$ REFINES $calling2$
       where
     $@grd1\ ok = FALSE$
     $@grd2\ setok = \varnothing$
     $@grd3\ sety = \varnothing$
       then
     $@act1\ ok := TRUE$
     $@act2\ r := x$
     $@act3\ setr := setx$
     $@act4\ setok := \{ball\}$
   end

   EVENT $step3$ REFINES $step2$
       any $z$
       where
     $@grd1\ ok = FALSE$
     $@grd2\ sety \neq \varnothing$
     $@grd3\ z \in sety$
     $@grd4\ setok = \varnothing$
     $@grd5\ y \neq 0$
       then
     $@act1\ setx := setx \cup \{z\}$
     $@act2\ sety := sety \setminus \{z\}$
     $@act3\ x := x + 1$
     $@act4\ y := y - 1$
   end
 $end$
```

```
MACHINE $M222$ REFINES $M22$ SEES $C1$

 VARIABLES $setx\ sety\ setr\ setok$

 EVENTS
  EVENT $INITIALISATION$
   $with$
    $@x'\ x' = a$
    $@y'\ y' = b$
     then
    $@act7\ setx, sety, setr, setok\ :|$
        $(setok' = \varnothing \wedge setr' \subseteq B \wedge setx' = seta \wedge sety' = setb)$
   end

  EVENT $calling3$ REFINES $calling3$
     where
    $@grd3\ setok = \varnothing$
    $@grd4\ sety = \varnothing$
     then
    $@act3\ setr := setx$
    $@act4\ setok := \{ball\}$
   end

  EVENT $step3$ REFINES $step3$
     any $z$
     where
    $@grd2\ sety \neq \varnothing$
    $@grd3\ z \in sety$
    $@grd4\ setok = \varnothing$
     then
    $@act1\ setx := setx \cup \{z\}$
    $@act2\ sety := sety \setminus \{z\}$
   end
 $end$
```

Rule 2 allows the addition to be calculated in several moves and the refinement expresses that the + function is calculated according to this application of the rule. This is an iteration that is completed when there is a ball in the setok variable.

We have used Event-B modelling to describe the rules for using the abacus to calculate addition, and we have also verified these rules against the process of using the abacus. This example shows how Event-B can be used to describe reactive systems incrementally.

0.5.2. *Concluding Comments*

We have presented elements of the Event-B language, which is based on B but offers a simple way to express transitions. A machine in Event-B describes the state of an *observed* system by listing variables and asserting an invariant. It also allows for changes in variables to be expressed by a finite list of events. The machine is checked by verifying a list of verification conditions expressing the preservation of the invariant by the events. The data description uses set theory and the predicate calculus of the B language (Abrial 1996*a*) with with some differences (Leuschel 2021) . This method has been developed from the classical B (Abrial 1996*a*) method and proposes a general framework for developing reactive systems, using a progressive approach to model design by refinement. Event-B was developed from the ground up based on the foundational work of Jean-Raymond Abrial's presentation (Abrial 1996*b*) at the inaugural conference (Habrias 1996) and the contributions of Ralph Back and Kurki Suonio This language was developed from the ground up based on the foundational work of Jean-Raymond Abrial's presentation at the inaugural conference and the contributions of Ralph Back and Kurki Suonio (Back and Kurki-Suonio 1989). The choices made have enabled us to use the Atelier-B tool to develop the first models Event-B . Finally, it should be noted that the refinement of Event-B machines is an original element of this approach implemented in the Rodin platform. The Event-B language is designed for developing models of reactive systems using an incremental and progressive approach guided by a set of techniques such as proof animation and simulation supported by Rodin. However, there is still a learning phase to use this language and the following chapters will propose such techniques.

0.6. Bibliography

Abrial, J.-R. (1996*a*), *The B book - Assigning Programs to Meanings*, Cambridge University Press.

Abrial, J.-R. (1996*b*), Extending B without Changing it (for Developingg Distributed Systems), *in* H. Habrias, (ed.), First Conference on the B Method. (Habrias 1996).

Abrial, J.-R. (2010), *Modeling in Event-B: System and Software Engineering*, Cambridge University Press.

Abrial, J.-R., Butler, M. J., Hallerstede, S., Hoang, T. S., Mehta, F., Voisin, L. (2010), Rodin: an open toolset for modelling and reasoning in event-b, *STTT*, 12(6), 447–466.

Abrial, J.-R., Cansell, D., Méry, D. (2003), A Mechanically Proved and Incremental Development of IEEE 1394 Tree Identify Protocol., *Formal Aspects of Computing*, 14(3), 215–227.

Back, R.-J., Kurki-Suonio, R. (1989), Decentralization of process nets with centralized control, *Distributed Computing*, 3(2), 73–87.

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

Back, R.-J., von Wright, J. (1998), *Refinement Calculus A Systematic Introduction*, Graduate Texts in Computer Science, Springer-Verlag.

Bjørner, D. (2021), *Domain Science and Engineering - A Foundation for Software Development*, Monographs in Theoretical Computer Science. An EATCS Series, Springer.
URL: *https://doi.org/10.1007/978-3-030-73484-8*

Bjørner, D., Henson, M. C., (eds) (2007), *Logics of Specification Languages*, EATCS Textbook in Computer Science, Springer.

Cansell, D., Méry, D. (2007), *The Event-B Modelling Method: Concepts and Case Studies*, Springer, pp. 33–140. See (Bjørner and Henson 2007).

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

Clarke, E. M., Grunberg, O., Peled, D. A. (2000), *Model Checking*, The MIT Press.

Cle (2002), *Atelier B*. http://www.atelierb.eu.

Cousot, P. (1978), Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes, PhD thesis, Université Scientifique et Médicale de Grenoble Institut National Polytechnique de Grenoble.

Cousot, P. (2000), Interprétation abstraite, *Technique et science informatique*, 19(1-2-3), 155–164.

Cousot, P., Cousot, R. (1979), Systematic design of program analysis frameworks, *in* Proceedings Records of Sixth Proceedings of the Symposium on Principles of Programming Languages, The ACM Press, San Antonio, Texas, pp. 269–282.

Cousot, P., Cousot, R. (1992), Abstract interpretation frameworks, *Journal of Logic and Computation*, 2(4), 511–547.

Dijkstra, E. W. (1976), *A Discipline of Programming*, Prentice-Hall.

Floyd, R. W. (1967), Assigning meanings to programs, *in* J. T. Schwartz, (ed.), Proc. Symp. Appl. Math. 19, Mathematical Aspects of Computer Science, American Mathematical Society, pp. 19 – 32.

Habrias, H., (ed.) (1996), *First Conference on the B Method*, University of Nantes.

Hoare, C. A. R. (1969), An axiomatic basis for computer programming, *Communications of the Association for Computing Machinery*, 12, 576–580.

Holzmann, G. (1997), The spin model checker, *IEEE Trans. on software engineering*, 16(5), 1512–1542.

Lamport, L. (1980), Sometime is sometimes Not never: A tutorial on the temporal logic of programs., *in* Proceedings of the Seventh Annual Symposium on Principles of Programming Languages, ACM SIGACT-SIGPLAN, ACM, pp. 174–185.

Lamport, L. (1994), A temporal logic of actions, *Transactions On Programming Languages and Systems*, 16(3), 872–923.

Lamport, L. (2002*a*), *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley.
URL: *http://research.microsoft.com/users/lamport/tla/book.html*

Lamport, L. (2002*b*), *Specifying Systems: The TLA$^{+}+$ Language and Tools for Hardware and Software Engineers*, Addison-Wesley.

Leuschel, M. (2021), Spot the difference: A detailed comparison between B and event-b, *in* A. Raschke, E. Riccobene, K. Schewe, (eds), Logic, Computation and Rigorous Methods - Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday, vol. 12750 of *Lecture Notes in Computer Science*, Springer, pp. 147–172.
URL: *https://doi.org/10.1007/978-3-030-76020-5_9*

Manna, Z., Pnueli, A. (1984), Adequate proof principles for invariance and liveness properties of concurrent programs, *Science of Computer Programming*, 4(3), 257–289.
URL: *https://www.sciencedirect.com/science/article/pii/0167642384900030*

McMillan, K. L. (1993), *Symbolic Model Checking*, Kluwer Academic Publishers.

Méry, D. (1986), A proof system to derive evantually properties under justice hypothesis, *in* J. Gruska, B. Rovan, J. Wiedermann, (eds), Mathematical Foundations of Computer Science 1986, Bratislava, Czechoslovakia, August 25-29, 1996, Proceedings, vol. 233 of *Lecture Notes in Computer Science*, Springer, pp. 536–544.
URL: *https://doi.org/10.1007/BFb0016280*

Méry, D. (1999), Requirements for a temporal B - assigning temporal meaning to abstract machines... and to abstract systems, *in* K. Araki, A. Galloway, K. Taguchi, (eds), Integrated Formal Methods, Proceedings of the 1st International Conference on Integrated Formal Methods, IFM 99, York, UK, 28-29 June 1999, Springer, pp. 395–414.
URL: *https://doi.org/10.1007/978-1-4471-0851-1_21*

Méry, D., Mokkedem, A. (1992), Crocos: An integrated environment for interactive verification of SDL specifications, *in* G. von Bochmann, D. K. Probst, (eds), Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, June 29 - July 1, 1992, Proceedings, vol. 663 of *Lecture Notes in Computer Science*, Springer, pp. 343–356.
URL: *https://doi.org/10.1007/3-540-56496-9_27*

Métayer, C., Voisin, L. (2009), The Event-B Mathematical Language, Technical report, The Rodin Project.

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

Turing, A. (1949), On checking a large routine, *in* Conference on High-Speed Automatic Calculating Machines, University Mathematical Laboratory,Cambridge.

van Gasteren, A. J. M., Tel, G. (1990), Comments on "on the proof of a distributed algorithm": always true is not invariant, *Information Processing Letters*, 35, 277–279.

ith some di

Design of Correct by Construction Sequential Algorithms

Dominique MÉRY[1]

[1] *LORIA, Telecom Nancy & Université de Lorraine*

0.7. Introduction

<

The development of correct algorithms or sequential programs from specifications (Dijkstra 1976) is a scientific theme linked to that of the verification of programs or algorithms (Turing 1949 ; Floyd 1967 ; Hoare 1969). program or algorithm verification turing49a,floyd67a,hoare69a. The fundamental question can be summarised in the form of a symbolic equation $\mathsf{D}, \mathsf{A} \;\Rightarrow\; \mathsf{C}$ where D (resp. A, C) is the problem domain (resp. the algorithm, the contract). In this equation, we assume that the domain of the problem D is known and may be, for example, $\mathbb{Z}$ the domain of integers and we will be interested in problems requiring properties on integers. A problem is a general expression to designate the calculation of a value from data or the search for a value in a set of data. The A algorithm is an algorithmic expression for expressing assignment instructions, conditional instructions and bounded or unbounded iterations. Finally, C is a contract expression in the form of two elements a pre-condition $\mathsf{pre}(v_0)$ and a post-condition $\mathsf{post}(v_0, v_f)$ relating the initial value v_0 of a flexible variable *v* to its final value v_f. The solving the problem consists in expressing it in the form of a contract and ensuring that for any initial value v_0 satisfying $\mathsf{pre}(v_0)$, there exists a value v_f satisfying $\mathsf{post}(v_0, v_f)$. On the other hand, it is important that the final value of v_f corresponds to a calculation of an algorithm A in the classical sense of computability (Rogers 1967). The equation can therefore be rewritten in the following form:$\forall v_0, v_f \in \mathsf{D}.\mathsf{pre}(v_0) \wedge v_0 \xrightarrow{\text{A}} v_f \Rightarrow \mathsf{post}(v_0, v_f)$ and we obtain the expression for the partial correction of the A algorithm in relation to the contract $\mathsf{C}(v, \mathsf{pre}(v_0), \mathsf{post}(v_0, v_f))$ on the domain D. The relation $\xrightarrow{\text{A}}$ expresses the calculation of A and we can add a second expression which plays the role of the termination of A: $\forall v_0 \in \mathsf{D}.\mathsf{pre}(v_0) \Rightarrow \exists v_f \in \mathsf{D}.v_0 \xrightarrow{\text{A}} v_f$. The relation $\xrightarrow{\text{A}}$ has the right property of determinism in our case of classical sequential algorithms. The two translations produce a synthetic expression of the following form:

$$\forall v_0 \in \mathsf{D}.\mathsf{pre}(v_0) \Rightarrow \begin{pmatrix} \forall v_f \in \mathsf{D}.v_0 \xrightarrow{\text{A}} v_f \Rightarrow \mathsf{post}(v_0, v_f) \\ \exists v_f \in \mathsf{D}.v_0 \xrightarrow{\text{A}} v_f \end{pmatrix}$$

which we rewrite with the wp calculus as follows: $\forall v_0 \in \mathsf{D}.v = v_0 \wedge \mathsf{pre}(v_0) \Rightarrow wp(A)(\mathsf{post}(v0, v))$

which we rewrite with Hoare triples as follows: $\{v = v_0 \wedge \mathsf{pre}(v_0)\}\mathsf{A}\{\mathsf{post}(v_0, v)\}$.

This discussion led us to give meaning to the correctness of an algorithm by considering its partial correctness as well as its termination. Hoare logic most often expresses partial correctness and in all rigour it would be necessary to use two notations, one expressing partial correctness and the other total correctness, but the objective here is not to verify an algorithm A and therefore to verify a list of verification conditions as Floyd method indicates, but to find an algorithm A which satisfies this equation $\forall v_0 \in \mathsf{D}.v = v_0 \wedge \mathsf{pre}(v_0) \Rightarrow wp(A)(\mathsf{post}(v0, v))$. The problem is therefore to construct an algorithm A enabling the contract C to be fulfilled in the domain D. In the *a posteriori* approach to correcting algorithms, we propose a solution for A and then apply the list of verification conditions. This technique also consists of applying the verification conditions without having clearly stated the contract C.

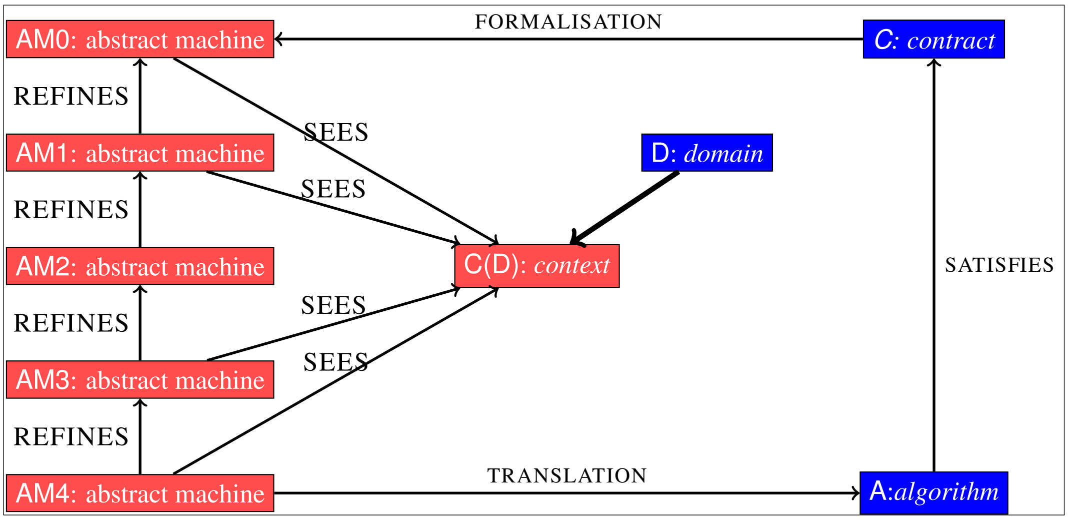


Figure 8. *Correctness by construction in Event-B*

Semantic analysis techniques can thus be developed with the abstract interpretation (Cousot 2021) and this is based on semantic techniques that simplify the life of the programmer who obtains analysis feedback. Correctness by construction is a technique which starts with an abstract algorithm A0 which fulfils the contract in a verified way and which is progressively enriched with increasingly complex control structures while observing the property of correctness with respect to the contract C. This progressive strategy of adding elements to make the result more precise is guided by the refinement relationship between the algorithms. Each transformation or refinement step guarantees that the resulting algorithm is correct. C. Morgan (Morgan 1990) develops the refinement calculus which makes it possible to progressively and correctly transform one algorithm into another algorithm by guaranteeing that the final algorithm is correct with respect to the first algorithm which is a pre/post specification considered as an algorithmic action of the form $v : |(pre, post)$. $v : |(pre, post)$ designates an algorithmic instruction I whose effect is to modify v by respecting the contract defined by pre and post. Thus, the strategy consists of constructing a sequence of algorithms A0,. . . , Ai, . . . , An with the properties:

– A0 is the expression of the contract: D, A0 $\Rightarrow$ A0

– for all i in $0..n-1$, the algorithm Ai refines Ai $-$ 1: D, Ai $\Rightarrow$ C and D, Ai $-$ 1 $\Rightarrow$ C.

– An is the algorithm satisfying the contract: D, An $\Rightarrow$ C.

More recently, Derrick G. Kourie and Bruce W. Watson (Kourie and Watson 2012) follow this strategy and implement the correction-by-construction paradigm on classical examples of classical programming problems. This approach is equipped with Key to enable the rules applied to be validated using a tool. the rules applied. We can identify in these two calculations of the fact that the contract becomes an algorithmic statement corresponding to a generalised algorithmic structure. In fact, as we can see, a contract can express the halting of Turing machines (Rogers 1967) in a language of assertions which is still fairly abstract, but this does not mean that we have solved the halting problem, but that we have extended the algorithmic language with a statement magic enabling it to be solved. This amounts to extending the space of solutions and then choosing what corresponds to the theory of computability. C. Morgan reminded us that all second degree equations have solutions in fields of complexes, but that the method of solving in the set of reals retains only the real solutions. The specification statement v : [pre, post] is a valid statement in this algorithmic language. Such a specification instruction can be expressed in the Event-B language.

This approach to developing correct by construction algorithms is quite simply implemented in the Event-B language and in fact equipped by the Rodin environment. Figure 8 describes the general idea. This idea consists of translating the contract as a *event* which performs the calculation described by the contract. The contract expresses the *what* but carries out the calculation as an observation of the event. This event is placed in a first abstract machine AM0, which uses the mathematical elements extracted from the problem domain D and expressed in the context Event-B C(D). The development of an algorithm consists in the gradual enrichment of the extracted machines AM0, . . . AMn, by expressing the computations necessary to systematically translate the last machine into an algorithmic form so as to guarantee the correctness of the A algorithm thanks to the correctness of the transformations provided by the refinement. It should be noted that the correctness concerns partial correctness and termination, and that the abstract machines are models containing variables that will not be implemented in the algorithm produced.

We will present two techniques that implement this development pattern:

– the inductive pattern based on transformations of abstract machines by Jean-Raymond Abrial (Abrial 2010, Chapter 15).

– the recursive pattern based on the relation call,textitevent that we have developed (Méry 2009 ; Cheng *et al.* 2016).

We will present these two methods by highlighting case studies of classical sequential algorithms.

0.8. Design of a Iterative Sequential Algorithm

0.8.1. *Problem 1 : Calculating the sum of a vevtor v of integer values*

First, we define the contract associated with this problem of calculating the sum of the elements of the vector v_0. The algorithm we are looking for is SUM.

variables n, v, r

definitions

$$pre(n_0, v_0, r_0) \stackrel{def}{=} \left\{ \begin{array}{l} n_0 \in \mathbb{N} \wedge n_0 \neq 0 \\ v_0 \in 1..n_0 \rightarrow \mathbb{Z} \\ r_0 \in \mathbb{Z} \wedge i_0 \in \mathbb{Z} \end{array} \right.$$

requires $\left(\begin{array}{l} n_0 \in \mathbb{N} \wedge n_0 \neq 0 \\ v_0 \in 1..n_0 \rightarrow \mathbb{Z} \end{array} \right)$

ensures $\left(\begin{array}{l} r_f = \sum\limits_{k=1}^{k=n_0} v_0(k) \\ n_f = n_0 \\ v_f = v_0 \end{array} \right)$

The domain of the problem to be solved is that of the integers $\mathbb{Z}$ and the contract states that the value of the result is the sum of the integers in the sequence v. This mathematical expression is not directly expressible in the mathematical language of Event-B and we define a sequence u characterising the values of the partial sums. The context associated with our C($\mathbb{Z}$) Event-B model is defined by enumerating the *requires* hypotheses and defining u.

First, we need to express the sum r of the sequence v_0 in the language of Event-B ; this formulation is immediate in mathematical terms: $r = \sum\limits_{k=1}^{k=n_0} v_0(k)$. As the notation for summing a finite sequence of values is not provided in the basic elements of the language, we must *define* this notion in a context *c0* which will contain the data of the problem and the notations defined specifically for this case. Thus, the *data* n_0 and v_0 are defined as being respectively a non-zero natural integer (axioms axm1,axm2) and a function v_0 of domain $1..n_0$ and codomain $\mathbb{Z}$ (axiom axm3). The aim is to define the theory in which we will describe our data.

Secondly, we introduce a sequence u of integer values corresponding to the partial sums $\sum\limits_{k=1}^{k=i} v_0(k)$. To do this, the idea is to define the partial sums using an inductive definition inductive definition, which technically requires us to be sure of the *well definition* of this sequence u. The sequence u is therefore defined as follows:

– u is a total function of $\mathbb{N}$ in $\mathbb{Z}$ (axiom axm4).

– Initially, the summation starts with 0 and $u(0) = 0$ (axiom axm5).

– For values of i less than n_0, the value of $u(i)$ is defined from that of $u(i-1)$ and $v_0(i)$ (axiom axm6).

– For all values greater than n_0, the value of $u(i)$ is equal to that of $u(n_0)$ (axiom axm7).

The axioms are given in the context of *c0* and constitute a theory which will be useful for proving the properties of the models we will develop later.

```
CONTEXT $c0$
CONSTANTS
  $n_0, v_0, u$
AXIOMS
  $axm1 : n_0 \in \mathbb{N}$
  $axm2 : n_0 \neq 0$
  $axm3 : v_0 \in 1 \,..\, n \to \mathbb{Z}$
  $axm4 : u \in \mathbb{N} \to \mathbb{Z}$
  $axm5 : u(0) = 0$
  $axm6 : \forall i \cdot i \in \mathbb{N} \wedge i > 0 \wedge i \leq n_0 \Rightarrow u(i) = u(i-1) + v_0(i)$
  $axm7 : \forall i \cdot i \in \mathbb{N} \wedge i > n_0 \Rightarrow u(i) = u(n_0)$
END
```

Each axiom is validated by a set of proof obligations to ensure the consistency of the definitions. We have therefore defined the mathematical framework of the problem and we will now define the problem of summing the sequence v_0.

0.8.1.1. *Specification of the problem to solve*

```
MACHINE  $S1$
SEES  $S0$
VARIABLES
  $r, v, n$
INVARIANTS
  $inv1 : r \in \mathbb{Z}$
  $inv2 : v \in 1..n_0 \to \mathbb{Z}$
  $inv3 : n \in \mathbb{Z}$
  $inv4 : n = n_0 \wedge v = v_0$
Event INITIALISATION
  BEGIN
    $act1 : r :\in \mathbb{Z}$
    $act2 : n := n_0$
    $act3 : v := v_0$
  END
Event final
  BEGIN
  $act1 : r := u(n)$
  END
END
```

The problem is therefore to calculate the value of the sum of the elements of the sequence v. We define a *S1* machine which is an abstract machine expressing through the Event final event the expression of the *postcondition* $r = u(n)$. In fact, the new value of the variable r will be $u(n)$, when the event Event final has been observed. The initial value of r is arbitrary at initialisation. Finally, the variable r must satisfy the very simple invariant $inv1 : r \in \mathbb{Z}$; this information constitutes a typing of the variable r. The event Event final is therefore simply an assignment of the value $u(n)$ to r.
We can express it as a HOARE triple:
$\{n = n_0 \wedge v = v_0 \wedge n_0 > 0 \wedge v_0 \in 1 \,..\, n_0 \to \mathbb{N}\}\mathsf{SUM}\{r = u(n_0)\}$.

Note that the data is *visible* from the context *s0*. The problem is therefore to find an algorithm that calculates the value $u(n)$ and stores it in r.
We have therefore described the domain of the problem to be solved and we have formulated what we want to calculate. The next step is to inventing a *method of calculation* and this requires a *idea of solution* and the use of refinement.

0.8.1.2. *Refining to compute inductively*

We have defined the specification of the problem for calculating the sum of the elements of a sequence v_0 and we now need to find a way to *calculate* the value of the sequence u at term n_0. The assignment $r := u(n_0)$ is an expression mixing a variable r and a mathematical value $u(n_0)$. A trivial and inefficient solution is well known: store the values of the sequence u in an array uu and translate the assignment into the form $r := uu(n)$ where uu verifies the following property $\forall k.k \in dom(t) \Rightarrow uu(k) = u(k)$ and this property constitutes an element of the invariant inv8. The idea is therefore to use the variable uu ($uu \in 0 \,..\, n_0 \nrightarrow \mathbb{Z}$) to control the calculation and its progress. Progression is ensured by the event step2, which decreases the quantity $n - i$ and therefore ensures that the progression process converges.

```
MACHINE   $S2$
REFINES   $S1$
SEES   $S0$
VARIABLES
   $r, uu, i$
INVARIANTS
   $inv1 : i \in \mathbb{N}$
   $inv2 : i \geq 0$
   $inv3 : i \leq n$
   $inv4 : uu \in 0\,..\,n \rightarrow\!\!\!\!+ \mathbb{Z}$
   $inv5 : dom(uu) = 0\,..\,i$
   $inv6 : n \notin dom(uu) \Rightarrow i < n$
   $inv7 : dom(uu) \subseteq dom(u)$
   $inv8 : \forall k \cdot \left(\begin{array}{l} k \in dom(uu) \\ \Rightarrow \\ uu(k) = u(k) \end{array}\right)$
   $inv9 : n = n_0 \wedge v = v_0$
VARIANTS $vrn : n - i$
```

```
Event INITIALISATION
   BEGIN
      $act1 : r :\in \mathbb{N}$
      $act2 : n := n_0$
      $act3 : v := v_0$
      $act4 : uu := \{0 \mapsto 0\}$
      $act5 : i := 0$
   END
Event final
   REFINES final
   WHEN
      $grd1 : n \in dom(uu)$
   THEN
      $act1 : r := uu(n)$
   END
END
```

```
Event step2
   WHEN
      $grd11 : n \notin dom(uu)$
   THEN
      $act11 : uu(i+1) := uu(i) + v$
      $act12 : i := i + 1$
   END
END
```

The $S2$ model therefore describes a process which progressively fills the uu table and therefore retains all the intermediate results. The proof obligations are fairly easy to prove insofar as we have *prepared* the work of the proof assistant. We will give the details of the statistics in a table at the end of the development. It is quite clear that the variable uu is in fact a witness or a trace of the intermediate values and that this variable can therefore be hidden in this model which will have to be refined. Before hiding this variable, we will set aside the value that we need to keep $uu(i)$.

0.8.1.3. *Focus on the value to be preserved*

The following refinement S3 will lead to the introduction of a new variable cu which will retain the last current value $uu(i)$. We therefore operate a *superposition* (Chandy and Misra 1988) on the S2 machine. The idea is therefore that this model refines or simulates the model S2 and this also means that the properties of the refined machines remain verified by the new machine S3 insofar as the proof obligations are all verified.

```
MACHINE   $S3$
   REFINES   $S2$
SEES   $S0$
VARIABLES
   $r, n, v, i, uu, cu$
INVARIANTS
   $inv1 : cu \in \mathbb{Z}$
   $inv2 : cu = u(i)$
Event INITIALISATION
   BEGIN
      $act1 : r :\in \mathbb{N}$
      $act2 : n := n_0$
      $act3 : v := v_0$
      $act4 : uu := \{0 \mapsto 0\}$
      $act5 : i := 0$
   $act6 : cu := 0$
   END
```

```
Event final
   REFINES final
   WHEN
      $grd1 : n \in dom(uu)$
      $grd2 : i = n$
   THEN
      $act1 : r := cu$
   END
Event step3   REFINES step2
   WHEN
      $grd1 : n \notin dom(uu)$
      $grd2 : i < n$
   THEN
      $act1 : uu(i+1) := uu(i) + v(i+1)$
      $act2 : i := i + 1$
      $act3 : cu := cu + v(i+1)$
   END
END
```

This machine is very expressive and provides a lot of information about the information required to ensure that the machine is suitable for the problem expressed in the S1 machine, which is refined by this S3 machine. It is even clearer that this S3 machine is expensive in terms of variables and the refinement allows us to leave only the variables that are useful for the calculation. In what follows, we will make the model more algorithmic and retain only those variables in the concrete model that are sufficient for the calculation.

0.8.1.4. *Obtaining an algorithmic machine*

In this final step, we refine the S3 machine into a S4 machine and hide the uu variable from the abstract S3 machine. Thus, the S4 machine includes the variables r, n, v, cu and i and we will also note that it satisfies safety properties called theorems in the S4 machine. These properties are proved from the properties of previous refined machines. We have thus obtained a machine comprising an initialisation and two events:

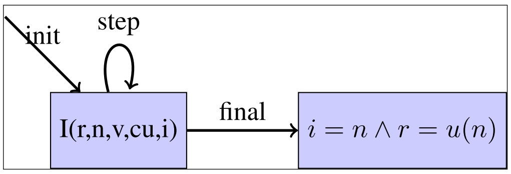
– The event final is observed when the value of i is n and, in this case, the variable cu contains the value $u(n)$. The invariant guarantees that the value of cu is $u(n)$.

– The event step4 is observed, when the value of i is less than n. This also means that, as long as this value is less than n, the event can be observed and the traces generated from these events therefore correspond to an iteration algorithmic structure.

```
MACHINE   $somme4$
  REFINES   $somme3$
SEES   $somme0$
VARIABLES
  $r, n, v, cuni$
THEOREMS
  $inv1 : cu = u(i)$
  $inv2 : i \leq n$
Event INITIALISATION
  BEGIN
    $act1 : r :\in \mathbb{Z}$
    $act2 : n := n_0$
    $act3 : v := v_0$
    $act4 : uu := \{0 \mapsto 0\}$
    $act5 : i := 0$
  END
```

```
Event final   REFINES final
  WHEN
    $grd1 : i = n$
  THEN
    $act1 : somme := psomme$
  END
Event step4   REFINES step3
  WHEN
    $grd1 : i < n$
  THEN
    $act1 : i := i + 1$
    $act2 : cu := cu + v(i + 1)$
  END
$i$END
```

The Rodin project archive ab-summation corresponds to this development by refinement, taking care to use the calculation method defined by the u sequence. The following diagram describes a view of the events observed as a function of the value of i.



```
$r :\in \mathbb{Z} \parallel i := 0 \parallel cu := 0 \parallel v := v0 \parallel n := n0$
while $i < n$ do
$i := i + 1 \parallel cu := cu + v(i + 1)$
od;
```

The components of ab-summation are constructed using the u sequence as a guide, taking care to obtain conditions that can be expressed in an algorithmic language. In our case, the condition $n \notin dom(uu)$ (resp. $n \in dom(uu)$) is refined by the condition $i < n$ (resp.$i = n$). Note that the diagram on the left corresponds to the algorithm on the right. These transformations can be defined more clearly and are implemented in a EB2ALGO (Singh 2024) plugin which produces the above algorithm from the ab-summation archive. Jean-Raymond Abrial (Abrial 2010, Chapter 15). suggests progressive transformation rules to be applied on model events like S4 and we will give a more complete treatment of these transformation rules implemented in EB2ALGO (Singh 2024).

variables n, v, r

definitions

$$pre(n_0, v_0, r_0) \overset{def}{=} \begin{cases} n_0 \in \mathbb{N} \land n_0 \neq 0 \\ v_0 \in 1..n_0 \rightarrow \mathbb{Z} \\ r_0 \in \mathbb{Z} \land i_0 \in \mathbb{Z} \end{cases}$$

requires $\begin{pmatrix} n_0 \in \mathbb{N} \land n_0 \neq 0 \\ v_0 \in 1..n_0 \rightarrow \mathbb{Z} \end{pmatrix}$

ensures $\begin{pmatrix} r_f = \sum_{k=1}^{k=n_0} v_0(k) \\ n_f = n_0 \\ v_f = v_0 \end{pmatrix}$

```
$int$ SUM($intn, v; int\ r$)
variables
   $int\ r, i = 0, cu = 0, v = v0, n = n0;$
while $i < n$ do
   $cu := cu + v(i+1);$
   $i := i + 1;$
od;
$r := cu;$
$return(r);$
```

0.8.1.5. *Comments on the methodology*

A specific methodology was employed in the selection of the variables. The uu variable is utilised for the storage of the calculated values of the u sequence, with the convention being to link the u sequence and the uu variable obtained by doubling the name of u. Obviously, we don't want to store all the intermediate values, just the ones used in the induction step. So the variable cu acts as a cursor to the value of uu that is useful in the induction step. uu is a model variable that is no longer necessary to retain for the algorithm. However, it has made the proof work easier, so it should be retained. Hiding uu provides a truly algorithmic view. It is also possible to obtain the termination of this algorithm with minimal effort, thanks to the variant which indicates that the event step leads to the dectreasing and convergence of this algorithm. The name final is only imposed by the plugin EB2ALGO (Singh 2024) and the use of Jean-Raymond Abrial (Abrial 2010, Chapter 15). Note that abstract machines implicitly contain the event skip and that each new event refines the previous level event skip. Another strategy would have been to introduce into the machine S1 an event keep which simulates the loop by anticipation. The archive abk-summation gives a version using this artifice and illustrates the use of an event *anticipated*.

0.8.2. *Transformations machines Event-B into sequential algorithms*

We take the conclusions of this simple problem and add the extra elements you need to develop iterative sequential algorithms. The plugin EB2ALGO implements the transformations of Jean-Raymond Abrial (Abrial 2010, Chapitre 15). We apply two transformations to the abstract machines obtained at the end of the refinement process, which we called ALGO, and it simplify the calculation process by hiding model variables. We recall the algorithmic language used by Jean-Raymond Abrial (Abrial 2010, Chapter 15)).

Definition 17 (The Pidgin Programming Language)

Statements of the language are:

– $variable\_list := expression\_list$

– statement; statement

– if *condition* then *statement* else *statement* end

– if *condition* then *statement* elseif ... else *statement* end

– while *condition* do *statement* end

A program can be *broken down* into a set of events, which are then triggered according to the values of the variables. This decomposition leads to the use of a composition of events. The idea is straightforward, but it must adhere to strict conventions to use the EB2ALGO (Singh 2024). plugin. We give these conditions which are implemented in the plugin and which must be respected when designing the development. during development design.

Let us consider two events, which we will merge into an algorithmic expression:

```
EVENT $e$
WHEN
  $P$
  $Q$
THEN
  $S$
END
```

```
EVENT $f$
WHEN
  $P$
  $\neg Q$
THEN
  $T$
END
```

$\leadsto$

```
EVENT $merging(e, f)$
WHEN
  $P$
THEN
  WHILE  $Q$  DO
    $S$
  OD;
  $T$;
END
```

We must specify the conditions of application. The event e appears as new or non-anticipated, therefore convergent at a lower level of refinement than that of f. We can be sure that there is a variant which terminates the loop. We must also assume that P is invariant to the event e. The event merge(e,f) appears at the same level as the event f. It is possible that P is not present.

```
WHEN
  $P$
  $Q$
THEN
  $S$
END
```

```
WHEN
  $P$
  $\neg Q$
THEN
  $T$
END
```

$\leadsto$

```
event $merging(e, f)$
WHEN
  $P$
THEN
  IF  $Q$  THEN
    $S$
  ELSE;
    $T$;
  FI;
END
```

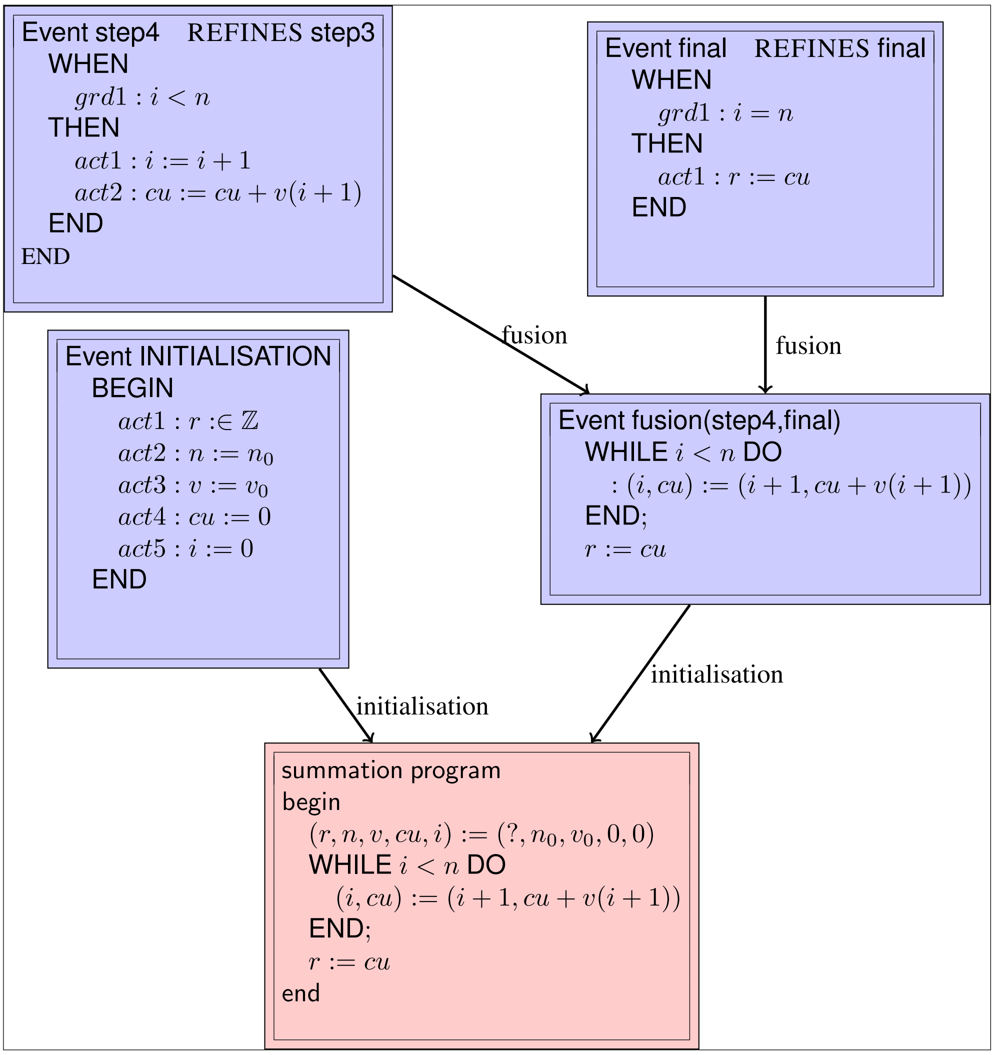
This transformation should be applied when the two events have been introduced at the same time. The event merge(e,f) must appear at the same level as the component. The guard P may not be present.

These two transformations can be used to design a plugin that produces a program in the language mentioned. The initial event is always called final and corresponds to the specification. Then the refinement process guides the design phase and it is also important to express that the new events that are introduced must decrease by a variant that ensures the convergence of the process described by the events.

Let us take the example we've already dealt with and apply the transformations.

```
MACHINE  $somme4$
  REFINES  $somme3$
SEES  $somme0$
VARIABLES
  $r, n, v, cuni$
THEOREMS
  $inv1 : cu = u(i)$
  $inv2 : i \leq n$
Event INITIALISATION
  BEGIN
    $act1 : r :\in \mathbb{Z}$
    $act2 : n := n_0$
    $act3 : v := v_0$
    $act4 : cu := 0$
    $act5 : i := 0$
  END
```

```
Event final  REFINES final
  WHEN
    $grd1 : i = n$
  THEN
    $act1 : r := cu$
  END
Event step4  REFINES step3
  WHEN
    $grd1 : i < n$
  THEN
    $act1 : i := i + 1$
    $act2 : cu := cu + v(i + 1)$
  END
$i$END
```



```
Event step4   REFINES step3
  WHEN
    $grd1 : i < n$
  THEN
    $act1 : i := i + 1$
    $act2 : cu := cu + v(i + 1)$
  END
END
```

```
Event final   REFINES final
  WHEN
    $grd1 : i = n$
  THEN
    $act1 : r := cu$
  END
```

```
Event INITIALISATION
  BEGIN
    $act1 : r :\in \mathbb{Z}$
    $act2 : n := n_0$
    $act3 : v := v_0$
    $act4 : cu := 0$
    $act5 : i := 0$
  END
```

```
Event fusion(step4,final)
  WHILE $i < n$ DO
    $: (i, cu) := (i + 1, cu + v(i + 1))$
  END;
  $r := cu$
```

```
summation program
begin
  $(r, n, v, cu, i) := (?, n_0, v_0, 0, 0)$
  WHILE $i < n$ DO
    $(i, cu) := (i + 1, cu + v(i + 1))$
  END;
  $r := cu$
end
```

We have given an example of how to apply the merge transformation for iteration and we refer the user to the plugin that implements these transformations.

0.9. Examples of development

0.9.1. *Problem 2: Computing the function power 3 $\lambda x.x^3$ using only additions*

The objective is to calculate the function power 3 ($\lambda x.x^3$) of a positive or zero integer using only addition operations. The method is to define a sequence z corresponding to the cubes of positive integers. The analysis that leads to the sequences defining z is not provided here, but the sequence is correct.

variables x, r

definitions

$$pre(x_0, , r_0) \overset{def}{=} \begin{cases} x_0 \in \mathbb{N} \\ r_0 \in \mathbb{Z} \end{cases}$$

$$\text{requires} \begin{pmatrix} n_0 \in \mathbb{N} \\ r_0 \in \mathbb{Z} \end{pmatrix}$$

$$\text{ensures} \begin{pmatrix} r_f = x_0 * x_0 * x_0 \\ x_f = x_0 \end{pmatrix}$$

Calculating the function power 3 ($\lambda x.x^3$) using only addition is based on the following sequences:

- $z_0 = 0$ et $\forall n \in \mathbb{N} : z_{n+1} = z_n + v_n + w_n$
- $v_0 = 0$ et $\forall n \in \mathbb{N} : v_{n+1} = v_n + t_n$
- $t_0 = 3$ et $\forall n \in \mathbb{N} : t_{n+1} = t_n + 6$
- $w_0 = 1$ et $\forall n \in \mathbb{N} : w_{n+1} = w_n + 3$
- $u_0 = 0$ et $\forall n \in \mathbb{N} : u_{n+1} = u_n + 1$

The first step is to show that the sequence z defines the sequence of cubes of different integers and therefore gives an inductive way of calculating the cube of a positive integer $x0$ using only addition alone. The domain of the problem to be solved is that of the integers $\mathbb{Z}$ and the contract expresses that the value of the result is the cube of the positive integer x_0.

The context P3-0 expresses the sequences defining the values to be calculated to produce a value corresponding to the cube of x_0. The important result is to show the following theorem with the Rodin platform.

Property 13 (Soundness of the sequence z)

$$\forall k.k \in \mathbb{N} \Rightarrow z(k) = k * k * k$$

This property guarantees that calculating the terms of the z sequence allows the value to be calculated using auxiliary sequences and only addition operations. These elements are given in the context P3-0. The contract can then be written in the form of a machine. P3-1 contains a single event, FINAL, whose action is `z := z(x0)`.

The method consists in refining the P3-1 machine into a P3-2 machine and introducing an iteration variable i covering the interval $0..x_0$ and variables for each sequence uu, vv, ww, tt, zz whose role is to store the values of the sequences to calculate $z(x_0)$. A new event is introduced to update the variables uu, vv, ww, tt, zz and i.. The invariant expresses the link between the mathematical values of the sequences u, v, w, t, z and the values stored and calculated in the variables uu, vv, ww, tt, zz. The invariant is fairly easy to determine, but we probably need to provide more relationships between the different sequences, so we come back to those sequences which have expressions that only mention i.

Property 14 (Properties of sequences u,v,w,t)

- $\forall k \in \mathbb{N} : v_k = 3 * k * k$
- $\forall k \in \mathbb{N} : w_k = 3 * k + 1$
- $\forall k \in \mathbb{N} : u_k = k$
- $\forall k \in \mathbb{N} : t_k = 3 * k + 3$

The properties are proved in the context of P3-0 and will be used in the refinement of the P3-2 machine. We introduce variables that point to the elements of the sequences that are sufficient for the computation. The refinement of P3-2 into P3-3 amounts to adding a variable for each sequence cu, cv, cw, ct, cz and these variables verify the following invariant property:

$$\begin{array}{l} inv1 : 0 \leq i \wedge i \leq x0 \wedge cv = vv(i) \\ inv2 : cw = ww(i) \\ inv3 : cz = zz(i) \\ inv4 : ct = tt(i) \\ inv5 : cu = uu(i) \end{array}$$

In addition, the events final and step are refined by making the guards *verifiable*. An expression of the form $x0 \in dom(zz)$ or $x0 \notin dom(zz)$ is difficult to translate efficiently into an algorithmic language. Thus, the new guard $i < x0$ implies $x0 \notin dom(zz)$ and the new guard $i = x0$ implies $x0 \in dom(zz)$. The resulting machine is therefore more deterministic and more approximate to an algorithmic expression. The proofs are automatic.

We finish by hiding the variables uu, vv, ww, tt, zz in a refinement P3-4 of the machine P3-3. It remains to use the property of sequences that we have proved in the context. The proof effort made in the context of P3-4 pays off when it comes to expressing the invariant properties constituting the loop invariant of the iterative algorithm produced from this machine. from this machine. The variable cu is useless, since it contains i. We have translated the P3-4 machine in the form of an ACSL algorithm **??** verified by the Frama-c application automatically. We obtained a cross-check of the algorithm obtained by translation from the P3-i machines.

Listing 1: ACSL power3.c

```
/*@   requires   0 <= x;
      ensures \result == x*x*x;
*/
int power3(int x)
{int   r, cz,cv,cu,cw,ct,  i;
  cz=0;cv=0;cw=1;ct=3;i=0;
        /*@
          @ loop invariant ct == 6*i +3;
          @ loop invariant cv== 3*i*i;
          @ loop invariant cw == 3*i+1;
          @ loop invariant cz == i*i*i;
          @ loop invariant i  <= x;
          @ loop assigns ct,cz,i,cv,cw,r; */
  while (i<x)
          {
            cz=cz+cv+cw;
            cv=cv+ct;
            ct=ct+6;
            cw=cw+3;
            i=i+1;}
  r=cz;return(r);
}
```

The bb-power3 archive contains all the machines used to develop this algorithm.

0.9.2. *Problem 3: Searching a value in an array*

The problem is to find the occurrence of a value x in an array t of dimension n. There are no constraints on the array or the search technique.

variables t, n, x, r

definitions

$$pre(x_0,,r_0) \overset{def}{=} \begin{cases} x_0 \in \mathbb{V} \\ t_0 : 1..n_0 \to \mathbb{V} \\ r_0 \in \mathbb{Z} \times BOOL \end{cases}$$

requires $\begin{pmatrix} x_0 \in \mathbb{V} \\ t_0 : 1..n_0 \to \mathbb{V} \\ r_0 \in \mathbb{Z} \times BOOL \end{pmatrix}$

ensures $\begin{pmatrix} t_f = t_0 \wedge n_f = n_0 \\ x \notin ran(t) \Rightarrow r_f = (0, FALSE) \\ x \in ran(t) \Rightarrow prj2(r_f) = FALSE \Rightarrow t_f(prj1(r_j)) = x \end{pmatrix}$

This problem must be reformulated in the form of a sequence that expresses for a value $i \in 0..n$ whether the array t contains the value x between 1 and i. As soon as the value is found in i, $u(j)$ is equal to $u(j)$ and the value found is i. If the value x is not in the table, then the sequence u is identically equal to a sequence of pairs $(0, FALSE)$.
We will define the sequence u in the context of S-0 and use the same methodology.

The context S-0 contains the definitions of the data t_0, n_0, x_0 and the axioms defining the sequence u whose value $u(n_0)$ is the expected solution.

$$\begin{array}{l}
axm4 : u \in 0\,..\,n \to \mathbb{Z} \times BOOL \\
axm5 : u(0) = 0 \\
axm6 : \forall i \cdot i \in 0\,..\,n-1 \land t(i+1) = x \land prj2(u(i)) = FALSE \Rightarrow u(i+1) = i+1 \mapsto TRUE \\
axm7 : \forall i \cdot i \in 0\,..\,n-1 \land t(i+1) \neq x \Rightarrow u(i+1) = u(i) \\
axm8 : \forall i \cdot i \in 0\,..\,n-1 \land t(i+1) = x \land prj2(u(i)) = TRUE \Rightarrow u(i+1) = u(i) \\
th1 : \forall i \cdot i \in 1\,..\,n \land prj2(u(i)) = FALSE \Rightarrow (\forall k \cdot k \in 1\,..\,i \Rightarrow t(k) \neq x) \\
th2 : \forall i \cdot i \in 1\,..\,n \land prj2(u(i)) = TRUE \Rightarrow (\exists k \cdot k \in 1\,..\,i \Rightarrow t(k) = x) \\
th3 : \forall i \cdot i \in 1\,..\,n \land prj2(u(i)) = TRUE \Rightarrow (t(prj1(u(i))) = x)
\end{array}$$

The machine S-1 expresses that the desired value is $u(n_0)$ and in fact expresses the contract for this problem. Unlike the Unlike the previous cases, the S-1 machine is refined into a S-2 machine which introduces the variables i and uu which define the inductive calculation scheme and which introduces two events step1 and step2 which simulate a conditional instruction according to the axioms defining u. The process continues with the introduction of the value of the sequence uu useful for the calculation, i.e. cu in the S-3 machine. The S-4 machine hides the uu variable and produces a fairly classic algorithm.

```
$r := (0, FALSE) \parallel k := 0 \parallel cuu := (0, FALSE)$
while $k < n$ do
  if $t(k+1) = x$ then
    if $prj2(cuu) = FALSE$ then
      $k := k+1 \parallel cuu := (k+1, TRUE)$
    else
      $k := k+1 \parallel cuu := (k+1, TRUE)$
    $elsek := k+1 || cuu := cuuelsek := k+1 || cuu := cuuod;$
```

The db-searching archive contains all the machines used to develop this algorithm.

0.9.3. *Problem 4: Computing a primitive recursive function*

Recursive primitive functions correspond to bounded iterations and a recursive primitive function is constructed from a scheme using two recursive primitive functions to define a function whose value we want to construct the algorithm that calculates its value. Examples of such functions are addition, multiplication, division, primality, ...

variables x, n, r

services
$H \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
$G \in \mathbb{N} \to \mathbb{N}$
$F \in \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
$F = PRIMREC(G, H)$

requires $\left(\begin{array}{l} x_0 \in \mathbb{N} \\ n_0 \in \mathbb{N} \end{array} \right.$
ensures $\left(\begin{array}{l} x_f = x_0 \land n_f = n_0 \\ r_f = F(x_0, n_0) \end{array} \right.$

Le problÃ¨me est de calculer la fonction F dÃ©finie Ã l'aide de deux autres fonctions G et H qui sont calculÃ©es par des algorithmes. F est dÃ©finie par l'Ã©quation suivante: pour toute valeur naturelle x,y, $\left\{ \begin{array}{l} F(x,0) = G(x) \\ F(x; y+1) = H(x, y, F(x,y)) \end{array} \right.$ The function F is defined using G and H, which are themselves defined by the same calculability schemes. This function F is itself a sequence which is specialised in v. We obtain the following algorithm.

```
$r :\in \mathbb{Z} \parallel k := 0 \parallel cv := G(x)$
while $k < x$ do
$k := k+1 \parallel cv := H(x, k, cv)$
od;
```

The eb-primitiverecursive archive contains all the machines used to develop this algorithm.

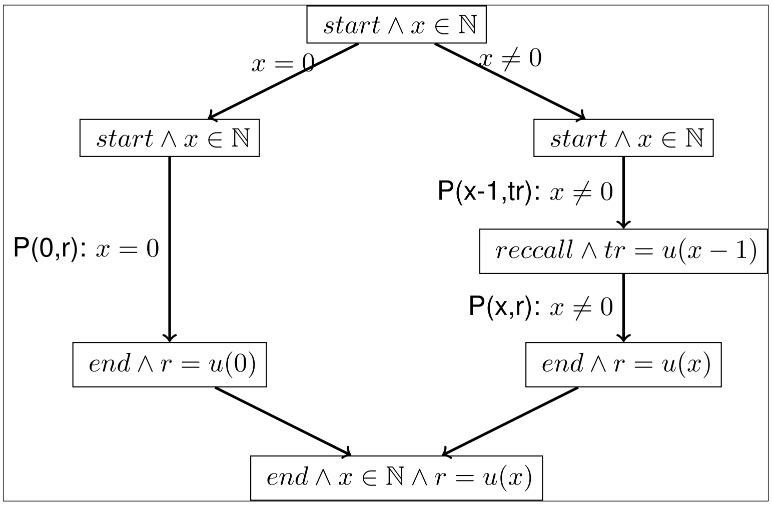

$start \wedge x \in \mathbb{N}$
$x = 0$
$x \neq 0$
$start \wedge x \in \mathbb{N}$
$start \wedge x \in \mathbb{N}$
P(x-1,tr): $x \neq 0$
P(0,r): $x = 0$
$reccall \wedge tr = u(x-1)$
P(x,r): $x \neq 0$
$end \wedge r = u(0)$
$end \wedge r = u(x)$
$end \wedge x \in \mathbb{N} \wedge r = u(x)$

Figure 9. *Organisation of the computation in a recursive solution using assertion diagram*

Jean-Raymond Abrial (Abrial 2010, Chapter 15) gives a list of programs built using this methodology and provides the Rodin[2]archives. It is important to note that Jean-Raymond Abrial's examples begin with a machine with an event final modelling the calculation corresponding to the problem solved, but also an event progress whose status *anticipated* means that events will appear to model one or more loops. This event models the implicit and hidden presence of intermediate calculations which must respect the invariant of the abstraction level. This event *conserves* or *preserves* the calculation and its invariant; we sometimes call it keep as opposed to an event which is always present called skip. Finally, it is quite clear that the translation still requires an intervention to produce an executable program as we have shown with the 1 algorithm and this allows us to check that the translation is correct since Frama-C is used to check the contract obtained. The loop invariant is derived from the Event-B development. Jean-Raymond Abrial uses Hoare triplets to express the specification of the problem to be solved and we prefer to use contracts that are available in programming languages such as ACSL/Frama-C or Spec#/Boogie.

0.10. Design of a Recursive Sequential Algorithm

In this section, we will apply the simple idea of analysing the diagram in figure 8 to develop a recursive program or algorithm. We will also promote one-shot refinement. In the previous section, we refined as long as necessary, especially as long as we obtained a set of events corresponding to computable elements. We have produced the EB2RC plugin (Cheng *et al*. 2016) , which implements this idea.

0.10.1. *The "Call as Event" Idea*

The refinement-based design of iterative sequential algorithms uses a sequence of values in a domain $\mathcal{D}$ and the computation process is based on the recording of the values of the sequence. In the case of the call-as-event paradigm, the pattern is based on the link between the occurrence of an event and a call of a function or procedure or method satisfying the pre-condition and post-condition respectively at the call point and the return point. The context C0 defines the sequence of values and the definition of the sequence is used as a guide for the shape of events. The definitions of sequence are reformulated by a diagram which is simulating the different cases when the procedure under development is called namely $P(x, r)$.

The diagram is derived from the Event-B model called ALGOREC and is a finite state diagram. It includes a liveness proof very close to the proof lattices of Owicki and Lamport (Owicki and Lamport 1982). We use special names for events in the diagram: P(0,r): $x = 0$ stands for the event observed when the procedure P(x,r) is called

[2] The link `https://web-archive.southampton.ac.uk/deploy-eprints.ecs.soton.ac.uk/122/index.html` gives a list of sequential program developments with a tutorial detailing how to refine and what to transform.

with x=0; P(x-1,tr): $x \neq 0$ models the observation of the recursive call of P; P(x,r): $x \neq 0$ stands for the event observed when the procedure P(x,r) is called with $x \neq 0$. P(0,r): $x = 0$ and P(x,r): $x \neq 0$ are refining the event computing which is observed when the procedure P is called.

```
MACHINE $ALGOREC$
REFINES $PREPOST$
SEES $C0$
VARIABLES
  $r, pc, tr$
INVARIANTS
    $art : pc \in L$
    $inv1 : tr \in D$
    $inv2 : pc = callrec \Rightarrow tr = v(x-1)$
    $inv3 : pc = end \Rightarrow r = v(x)$
```

The refinement is an organisation of the inductive definition using a control variable pc. The control variable pc is organising the different steps of the computations simulated by the events. The invariant is derived directly from the definitions of the intermediate values. Proof obligations are simple to prove. It remains to prove that the values of the sequence v correspond to the required value in the post-condition.

```
Event P(x,r):x=0
  REFINES computing
  WHEN
    $grd1 : x = 0$
    $grd2 : pc = start$
  THEN
    $act1 : r := d0$
    $act2 : pc := end$
  END
```

```
Event P(x-1,tr):x/=0
WHEN
  $grd1 : pc = start$
  $grd2 : x \neq 0$
THEN
  $act1 : tr := v(x-1)$
  $act2 : pc := callrec$
END
```

```
Event P(x,r):x/=0
  REFINES computing
  WHEN
    $grd1 : pc = callrec$
  THEN
    $act1 : r := f(tr)$
    $act2 : pc := end$
  END
```

The machine is simulating the organisation of the computations following two cases according to the figure 9. The first case is the path on the left part of the diagram and is when x is 0 and the second case if when x is not 0.

The first path is a three steps path and is labelled by the condition $x = 0$ an d the event P(0,r):x=0. The event P(x,r):x=0 is assigning the value $d0$ to r according to the definition of $u(0)$. It refines the event computing in the abstraction. The third step is an implication leading to the postcondition.

The second path is a four steps path and is labelled by the condition $x \neq 0$, then the event P(x-1,r):$x \neq 0$ is modelling the recursive call of the same procedure. Finally the event P(,r):$x \neq 0$ is refining the event computing. The call as event paradigm is applied when one considers that one event is defining the specification of the recursive call and the user is giving the name of the call to indicate that the event should be translated into a call. The EB2RC plugin (Cheng *et al.* 2016) generates automatically a C-like program.

The model ALGOREC is simple to checked. Proof obligations are simple, because the recursive call is hiding the previous values stored in the variable vv of the iterative paradigm. The prover is much more efficient.

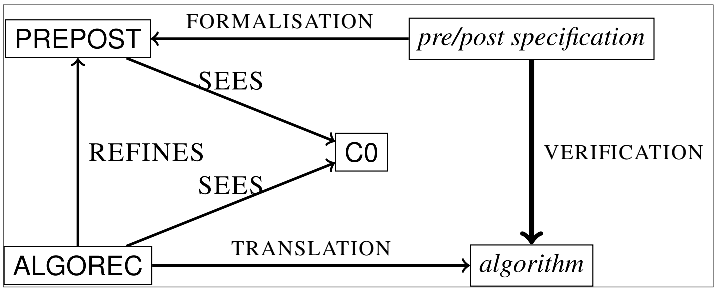


Figure 10. *The recursive pattern*

The recursive pattern is linked to a diagram which is helping to structure the solution. We have labelled arrows by guards or by events. The diagram helps to structure the analysis based on the inductive definitions. Following this pattern, we have developed the ERB2RC plugin based on the identification of three possible events. When a pre/post specification is stated, the program to build can be expressed by a simple event expressing the relationship between input and output and it provides a way to express pre/post specification as events. The first model is a very abstract model containing the pre/post events.

Since the refinement-based process requires an idea for introducing more concrete events. A very simple and powerful way to refine is to introduce a more concrete model which is based on an inductive definition of outputs with respect to the input.

A first consequence is that the concrete model is containing events which are computing the same function but corresponding to a recursive call expressed as events (Event rec%PROC(h(x),y)%P(y)). The event Event rec%PROC(h(x),y)%P(y) is simply simulating the recursive call of the same function and this expression makes the proofs easier. The invariant is defined in a simpler way by analysing the inductive structure and a control variable is introduced for structuring the inductive computation. We have identified three possible events to use in the concrete model:

```
Event
  e
where
  $\ell = \ell_1$
  $g_{\ell_1,\ell_2}(x)$
then
  $\ell := \ell_2$
  $x := f_{\ell_1,\ell_2}(x)$
end
```

```
Event
  rec%PROC(h(x),y)%P(y)
any $y$
where
  $\ell = \ell_1$
  $g_{\ell_1,\ell_2}(x, y)$
then
  $\ell := \ell_2$
  $x := f_{\ell_1,\ell_2}(x, y)$
end
```

```
Event
  call%APROC(h(x),y)%P(y)
any $y$
where
  $\ell = \ell_1$
  $g_{\ell_1,\ell_2}(x, y)$
then
  $\ell := \ell_2$
  $x := f_{\ell_1,\ell_2}(x, y)$
end
```

0.10.2. *Applying the call as event technique*

We will illustrate this method of designing recursive programs using a few relevant examples.

0.10.2.1. *Problem 1: Computing the power 2 ($\lambda x.x^2$)*

Applying the recursive pattern is made easier by the first steps of the iterative pattern. In fact, the context C0 and the machine PREPOST are the starting points of the iterative pattern as well as the recursive pattern. We use the computation of the function x^2 and we obtained the following refinement of PREPOST. Fig. 9 is the diagram analysing the way to solve the computation of the value of $u(x)$ following the call-as-event paradigm.

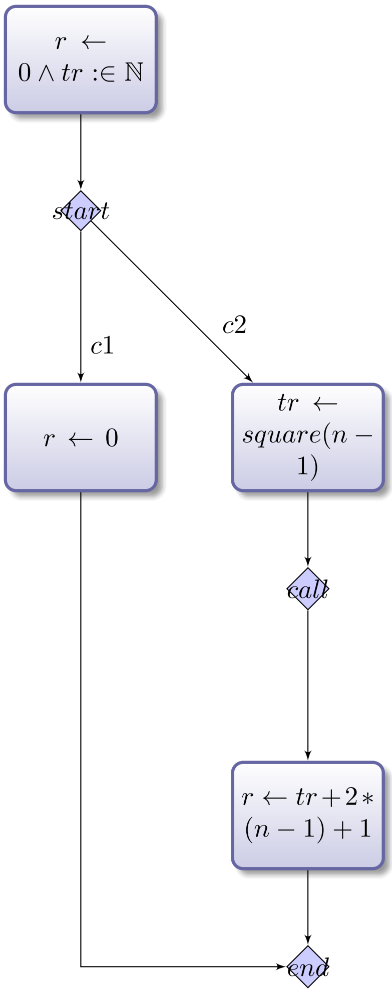
```
MACHINE $square$ // $square(n;r)$
  REFINES $specquare$  SEES $control0$

 VARIABLES $r\ l\ tr$

 INVARIANTS
  $@inv1\ r\ \in\ \mathbb{N}$
  $@inv2\ l = end\ \Rightarrow\ r\ =\ n*n$
  $@inv3\ l = call\ \Rightarrow\ n\ \neq\ 0$
  $@inv4\ l = call\ \Rightarrow\ tr\ =\ (n-1)*(n-1)$
  $@inv5\ l\ \in\ C$
  $@inv6\ tr\ \in\ \mathbb{N}$
  $@inv7\ l = end\ \Rightarrow\ r\ =\ n*n$
  $@inv8\ l = end\ \wedge\ n\ \neq\ 0$
        $\Rightarrow\ \ tr = (n-1)*(n-1)\ \ \wedge\ \ r = tr + 2*(n-1)+1$
  $theorem\ @inv9\ l = call\ \ \Rightarrow\ \ n*n\ =\ tr\ +\ 2*(n-1)+1$
```

```
EVENTS
  EVENT $INITIALISATION$
      then
    $@act1\ r\ \ :=\ \ 0$
    $@act2\ l\ \ :=\ \ start$
    $@act3\ tr\ \ :\in\ \ \mathbb{N}$
   end

  EVENT $square(n;r)@n = 0$
   REFINES $square(n;r)$
      where
    $@grd1\ l = start$
    $@grd2\ n = 0$
      then
    $@act1\ l\ \ :=\ \ end$
    $@act2\ r\ \ :=\ \ 0$
     end
```

```
 EVENT $square(n;r)@n/ = 0$  REFINES $square(n;r)$
      where
    $@grd1\ l = call$
      then
    $@act1\ r\ \ :=\ \ tr\ +\ 2*(n-1)\ +1$
    $@act2\ l\ \ :=\ \ end$
   end

  EVENT $rec@square(n-1;tr)@n\ \neq\ 0$
      where
    $@grd1\ l = start$
    $@grd2\ n\ \neq\ 0$
      then
    $@act1\ l\ \ :=\ \ call$
    $@act2\ tr\ \ :=\ \ (n-1)*(n-1)$
   end
 $end$
```

The variable l is modelling the control in the diagram. We introduce control points corresponding to assertions in the labels of the diagram as $C = \{start, end, callrec\}$. Three events are defined and the invariant is written very easily and proofs are derived automatically. The event rec%square(n-1;tr) is the key event modelling the recursive call. In the current example, we have modified the machine by using directly the fact that $v(n) = n*n$ and normally we had to use the sequence following the recursive pattern and then we had to derive the theorem $v(n) = n*n$.

Proofs are simpler and invariants are easier to extract from the inductive definitions. From the contexts and the machines we constructed, respecting the rules and choosing a name for the elements that allow us to produce an algorithm using the EB2RC plugin, we obtain a recursive algorithm that meets the problem specification and we obtain a diagram that shows the different steps in this algorithm.


$r \leftarrow 0 \wedge tr :\in \mathbb{N}$
$start$
$c1$
$c2$
$r \leftarrow 0$
$tr \leftarrow square(n-1)$
$call$
$r \leftarrow tr + 2 * (n-1) + 1$
end

The diagram is produced with tikz and has annotations defined by this list.

c1 $n = 0$

c2 $n \neq 0$

The algorithm produced is given below and is very simple to produce from the model.

```
procedure $square(n; r)$
begin
$r \leftarrow 0$
$tr :\in \mathbb{N}$
if $n = 0$
      $r \leftarrow 0$
elsif $n \neq 0$
      $tr \leftarrow square(n-1)$
      $r \leftarrow tr + 2 * (n-1) + 1$
endif
end
```

The invariant states simple and obvious properties related to control points. Theorem 9 is particularly worth examining. It is easy to derive because it corresponds to the recursive call. All the calls and all the details are swept under the carpet, leaving only the last call. This shows the importance and interest of recursion.

```
MACHINE  $square$ // $square(n; r)$
  REFINES  $specquare$   SEES  $control0$

 VARIABLES  $r\ l\ tr$

 INVARIANTS
  $@inv1\ r \in \mathbb{N}$
  $@inv2\ l = end \Rightarrow r = n * n$
  $@inv3\ l = call \Rightarrow n \neq 0$
  $@inv4\ l = call \Rightarrow tr = (n-1) * (n-1)$
  $@inv5\ l \in C$
  $@inv6\ tr \in \mathbb{N}$
  $@inv7\ l = end \Rightarrow r = n * n$
  $@inv8\ l = end \wedge n \neq 0$
       $\Rightarrow tr = (n-1) * (n-1) \wedge r = tr + 2 * (n-1) + 1$
  $theorem\ @inv9\ l = call \Rightarrow n * n = tr + 2 * (n-1) + 1$
```

0.10.2.2. *Problem 2: Binary search in an array*

We solve the problem of *searching for a value in a table*. The input parameters of the *binsearch* procedure are: a sorted array t; the bounds of the array within which the algorithm should search (lo and hi); and the value for which the algorithm should search (val). Output parameters are $result$ and a boolean flag ok that indicates if $t(result) = val$. The procedures pre and post conditions are presented as follow:

contract $binsearch(t, val, lo, hi, ok, result)$

$$\text{requires} \begin{pmatrix} t \in 0..t.Length \longrightarrow \mathbb{N} \\ \forall k.k \in lo..hi-1 \Rightarrow t(k) \leq t(k+1) \\ val \in \mathbb{N} \\ l, h \in 0..t.Length \\ lo \leq hi \end{pmatrix}$$

$$\text{ensures} \begin{pmatrix} ok = TRUE \Rightarrow \ t(result) = val \\ ok = FALSE \Rightarrow (\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{pmatrix}$$

```
EVENT $find$
    any $j$
    where
  $@grd1\ j \in lo..hi$
  $@grd2\ t(j) = key$
    then
  $@act1\ ok := TRUE$
  $@act2\ i := j$
end

EVENT $fail$
    where
  $@grd1\ \forall\ k\ k \in lo..hi \Rightarrow t(k) \neq key$
    then
  $@act1\ ok := FALSE$
end
```

The array t is sorted with respect to the ordering over integers and a simple inductive analysis is applied leading to a binary search strategy.

The specification is first expressed by two events corresponding to the two possible cases: either a key exists in the array t containing the value val, or there is no such key. These two events correspond to the two possible resulting *calls* to the procedure $binsearch(t, val, lo, hi; ok, result)$:

– Event find is $binsearch(t, val, lo, hi; ok, result)$ where $ok = TRUE$

– Event fail is $binsearch(t, val, lo, hi; ok, result)$ where $ok = FALSE$

These two events form the machine called *binsearch1* which is refined to obtain *binsearch2* (corresponding to PROCESS of Figure 14). In addition to these events, the events of this refined machine contains a new control label, l, which *simulates* how the binary search is achieved.

The refinement of binsearch1 is interesting for showing an invariant derived from the properties of the decomposition following the analysis of the searching process. The invariant is explicitly considering the role of the index *middle* and it is a clear statement of the property. We say that the recursivity is putting the dust under the carpet and it makes simpler to write and to read the solution.

```
MACHINE $binsearch2$ REFINES $binsearch1$ SEES $binsearch0$

 VARIABLES $i\ ok\ l\ mi$

 INVARIANTS
  $@inv1\ i \in 1..n$
  $@inv2\ l \in LOC$
  $@inv3\ dom(t) = 1..n$
  $@inv4\ mi \in 1..n$
  $@inv5\ l = middle \Rightarrow lo < hi \land mi \in lo..hi$
  $@inv6\ l = middle \land key < t(mi) \Rightarrow (\forall\ k\ k \in mi..hi \Rightarrow t(k) \neq key)$
  $@inv7\ l = middle \land key > t(mi) \Rightarrow (\forall\ k\ k \in lo..mi \Rightarrow t(k) \neq key)$
  $@inv8\ l = end \land ok = TRUE \Rightarrow i \in lo..hi \land t(i) = key$
  $@inv9\ l = end \land ok = FALSE \Rightarrow (\forall\ k\ k \in lo..hi \Rightarrow t(k) \neq key)$
  $theorem\ @inv10\ lo..hi \subseteq 1..n$
  $theorem\ @inv11\ (\exists\ j\ l = middle \land j \in mi+1..hi \land key > t(mi) \land t(j) = key \land mi+1 \leq hi)$
             $\Rightarrow (mi+1 \leq hi)$
```

The first events are the INITIALISATION event, the two events m1 and m2 corresponding the the case $mo = hi$ and the split event which is corresponding to the case $mo \neq hi$. The events are written the case analysis.

```
EVENT $INITIALISATION$
    then
  $@act1\ i :\in\ 1..n$
  $@act2\ ok\ := FALSE$
  $@act3\ l\ := start$
  $@act4\ mi\ :\in 1..n$
 end

EVENT $m1$ REFINES $find$
    where
  $@grd1\ l = start$
  $@grd2\ lo = hi$
  $@grd3\ t(lo) = key$
 $with$
  $@j\ j = lo$
    then
  $@act1\ l\ := end$
  $@act2\ ok\ := TRUE$
  $@act3\ i\ := lo$
 end
```

```
EVENT $m2$ REFINES $fail$
    where
  $@grd1\ l = start$
  $@grd2\ lo = hi$
  $@grd3\ t(lo) \neq key$
    then
  $@act1\ l\ := end$
  $@act2\ ok\ := FALSE$
 end

EVENT $split$
    where
  $@grd1\ l = start$
  $@grd2\ lo < hi$
    then
  $@act1\ l\ := middle$
  $@act2\ mi\ := (lo + hi)2$
 end
```

The split event directs the analysis and divides the exploration space into three parts of the indexes. Figure 0.10.2.2, the m3 event considers the case where the mi index is the one where the searched value is found. Then the other two events correspond to the segment between mi+1 and hi and are in fact recursive calls to the procedure under construction. These two events refine find and fail respectively. We need to add two more events corresponding to the segment $lo, mi - 1$ segment to complete the model.

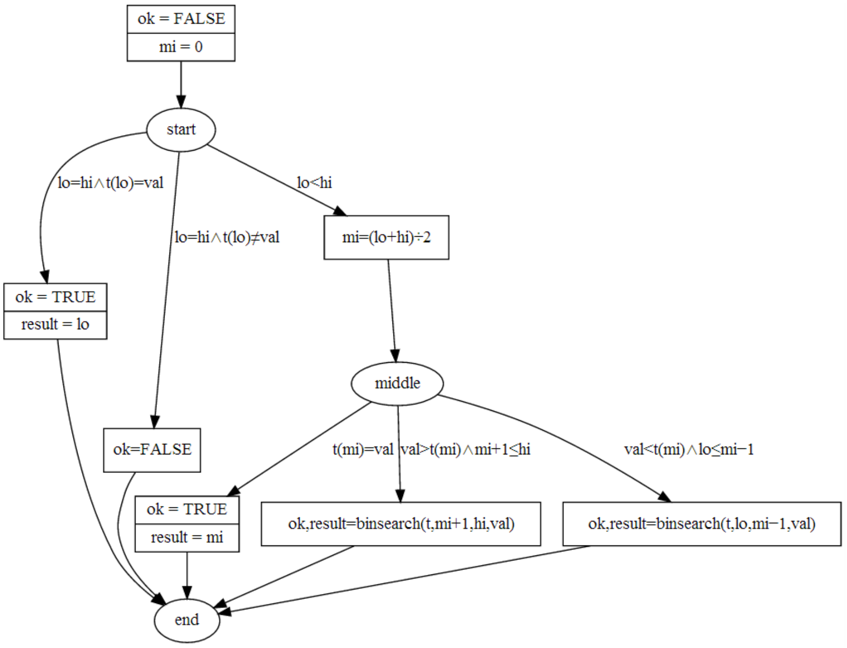


Figure 11. *Visualized Representation of the Binary Search Algorithm*

A textual representation of the binary search algorithm is constructed by the EB2RC. The produced algorithm (as shown in Algorithm **??**) has been compared to the algorithm produced by hand by the authors. The two algorithms are identical up to a slight reformatting.

Model	Total	Auto	Manual	Reviewed	Undischarged	% auto
binsearch1	5	5	0	0	0	100 %
binsearch2	71	63	8	0	0	78 %

Table 4. *Proof effort of our refinement approach for the binary search case study*

The proof effort of our refinement approach for the Binary Search case study is illustrated in Table 4. The first abstract model is proved automatically and the second concrete model is automatic in 78 % of its proof obligations.

The integrated development framework takes this into consideration. As shown in Fig. 14, we suggest to translate every recursive algorithm ALGORITHM into a partially annotated and iterative OPTIMISED ALGORITHM to be verified within the Spec# Programming System. In (Méry and Monahan 2013), we have proposed and proved a sound translation procedure from ALGORITHM to OPTIMISED ALGORITHM to perform this task. For example, the iterative version of the binary search algorithm in Spec# is shown in Fig. 12.

By sending this program to Spec#, Spec# reports the program as verified. No user interaction is required in this verification as all assertions required (preconditions, postconditions and loop invariants) have been generated as part of the refinement and transformation of the initial abstract specification into the final iterative algorithm. The automatic verification of the final Spec# program is available online at `http://www.rise4fun.com/SpecSharp/kyKW`.

```
Recursive Algorithm binsearch(t,lo,hi,val;ok,result) generated by EB2RC
begin
$ok := FALSE; mi := 0;$
if $lo = hi \wedge t(lo) = val$ then
   $ok := TRUE;$
   $result := lo;$
elseif $lo = hi \wedge t(lo) \neq val$ then
   $ok := FALSE;$
elseif $lo < hi$ then
   $mi := (lo + hi) \div 2;$
   if $t(mi) = val$ then
      $ok := TRUE;$
      $result := mi;$
   elsif $val > t(mi) \wedge mi + 1 \leq hi$ then
      $ok, result := binsearch(t, mi + 1, hi, val);$
   elsif $val < t(mi) \wedge lo \leq mi - 1$ then
      $ok, result := binsearch(t, lo, mi - 1, val);$
end
```

0.10.3. *Comments on the call as event idea*

In the example of subsubsection0.10.2.1, we do not use the event like call%APROC(h(x),y)%P(y) but the event is clearly a call for another procedure or function. For instance, when a sorting algorithm is developed, you may need an auxiliary operation for scanning a list of values to get the index of the minimum. It means that we have a way to define a library of models and to use correct-by-construction procedures or functions. In (Cheng *et al.* 2016) , we detail the tool and the way to define a library of *correct-by-construction programs*. The EB2RC plugin is used on this project and we obtain two files: one containing the algorithm and another containing the diagram built from the Event-B model.

0.11. Final comments

We have presented a use of the Event-B language in the derivation of sequential programs or algorithms. The first technique in fact uses the sequences of values leading to a given term constituting the sought-after solution.e Thus the calculation of the square or the cube is carried out by defining a sequence one of whose terms is the

```
class BS {
 int BinarySearch(int[] t, int val, int lo, int hi, bool ok)
  requires 0 <= lo && lo < t.Length && 0 <= hi && hi < t.Length;
  requires lo <= hi && 0 < t.Length;
  requires forall{int i in(0:t.Length),int j in (i:t.Length);t[i]<=t[j]};
  ensures -1 <= result && result < t.Length;
  ensures (0 <= result && result < t.Length)==> t[result] == val;
  ensures result == -1 ==> forall {int i in (lo..hi); t[i] != val};
 { int mi = (lo + hi) / 2;
   while (!(lo == hi && t[lo] == val) || ( lo == hi && t[lo] != val)
              || (lo < hi && (mi == (lo + hi) /2) && t[mi] == val))
    invariant 0 <= lo && lo < t.Length && 0 <= hi && hi < t.Length;
    invariant 0 <= mi && mi < t.Length;
    invariant (val < t[mi]) ==> forall {int i in (mi..hi); t[i] != val};
    invariant (val > t[mi]) ==> forall {int i in (lo..mi); t[i] != val};
   { mi = (lo + hi) / 2;
     if ((mi+1 <= hi) && (val > t[mi])) lo = mi +1;
     else if ((lo <= mi-1) && (val < t[mi])) hi = mi - 1;
   }
   if ((lo == hi) && (t[lo] == val)) {ok = true; return lo;}
   else{
    if ((lo == hi) && (t[lo] != val)) {ok = false; return -1;}
    else if ((lo < hi) && (t[mi] == val)) {ok = true; return mi;}
    else {ok = false; return -1;}
   }
 }
}
```

Figure 12. *Binary Search C# program corresponding to the generated iterative procedure.*

value of the square or the cube. terms is the value of the square or cube. Insofar as refinement is a very general relationship on a set of very general models too, we must try to set ourselves a refinement discipline. The first technique invites us to play with the model variables and to reduce the variables to those which are exclusively used for the calculation. In this way, we can better understand the notions of model variable or ghost variable used in certain proof tools which look for these variables. In our case, we introduce them and then hide them in the abstract models. in the abstract models and they make it easier to prove and state invariants. The second technique is simpler because it relies on inductive definitions that are interpreted in the recursive paradigm. It's about saving refinement and developing at a step of refinement. The resulting program is recursive and the recursiveity should be deleted and it is relatively easy to produce using our event conventions. In this case, we noted that the proofs were relatively simpler to derive. Both techniques rely on experimental plugins but could be combined. Figure 14 illustrates an environment for co-ordinating the various techniques for relatively complex sequential systems, but it is necessary to develop a new formal IDE integrating these techniques and close to the diagram in figure. 14 . Naturally, the development of concurrent, parallel or distributed algrorithms or programs is a challenge to be taken up, and the chapter entitled *Design of Correct by Construction Distributed Algorithms* will provide some ideas.

0.12. Bibliography

Abrial, J.-R. (2010), *Modeling in Event-B: System and Software Engineering*, Cambridge University Press.

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

Cheng, Z., Méry, D., Monahan, R. (2016), On two friends for getting correct programs - automatically translating event B specifications to recursive algorithms in rodin, *in* T. Margaria, B. Steffen, (eds), Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I, vol. 9952 of *Lecture Notes in Computer Science*, pp. 821–838.
URL: *https://doi.org/10.1007/978-3-319-47166-2_57*

Cousot, P. (2021), *Abstract Interpretation*, The MIT Press. ISBN: 9780262044905.

Dijkstra, E. W. (1976), *A Discipline of Programming*, Prentice-Hall.

Floyd, R. W. (1967), Assigning meanings to programs, *in* J. T. Schwartz, (ed.), Proc. Symp. Appl. Math. 19, Mathematical Aspects of Computer Science, American Mathematical Society, pp. 19 – 32.

Hoare, C. A. R. (1969), An axiomatic basis for computer programming, *Communications of the Association for Computing Machinery*, 12, 576–580.

```
EVENT $m3$ REFINES $find$
    where
  $@grd1\ l = middle$
  $@grd3\ t(mi) = key$
 $with$
  $@j\ j = mi$
    then
  $@act1\ l\ := \ end$
  $@act2\ ok\ := \ TRUE$
  $@act3\ i\ := \ mi$
 end

 EVENT $rec@search(t, val, mi + 1, hi, ok, result)@ok = TRUE$ REFINES $find$
    any $j$
    where
  $@grd1\ l = middle$
  $@grd2\ key\ >\ t(mi)$
  $@grd3\ j\ \in\ mi + 1..hi$
  $@grd4\ t(j) = key$
  $@grd5\ mi + 1\ \leq\ hi$
    then
  $@act1\ i\ := \ j$
  $@act2\ ok\ := \ TRUE$
 end

 EVENT $rec@search(t, val, mi + 1, hi, ok, result)@ok = FALSE$ REFINES $fail$
    where
  $@grd1\ l = middle$
  $@grd2\ key\ >\ t(mi)$
  $@grd4\ \forall\ j\ j\ \in\ mi + 1..hi\ \Rightarrow\ t(j)\ \neq\ key$
  $@grd5\ mi + 1 \leq hi$
    then
  $@act3\ l\ := \ end$
  $@act2\ ok\ := \ FALSE$
 end
```

fig:

Figure 13. *Events for recursive calls*

Kourie, D. G., Watson, B. W. (2012), *The Correctness-by-Construction Approach to Programming*, Springer. **URL:** *https://doi.org/10.1007/978-3-642-27919-5*

Méry, D. (2009), Refinement-based guidelines for algorithmic systems, *Int. J. Softw. Informatics*, 3(2-3), 197–239. **URL:** *http://www.ijsi.org/ch/reader/view_abstract.aspx?file_no=197&flag=1*

Méry, D., Monahan, R. (2013), Transforming event B models into verified c# implementations, *in* A. Lisitsa, A. P. Nemytykh, (eds), First International Workshop on Verification and Program Transformation, VPT 2013, Saint Petersburg, Russia, July 12-13, 2013, vol. 16 of *EPiC Series in Computing*, EasyChair, pp. 57–73. **URL:** *https://doi.org/10.29007/9wm9*

Morgan, C. (1990), *Programming from Specifications*, Prentice Hall International Series in Computer Science, Prentice Hall.

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

Rogers, H. J. (1967), *Theory of Recursive Functions and Effective Computability*, The MIT Press.

Singh, N. K. (2024), EB2ALGO. Plugin Rodin.

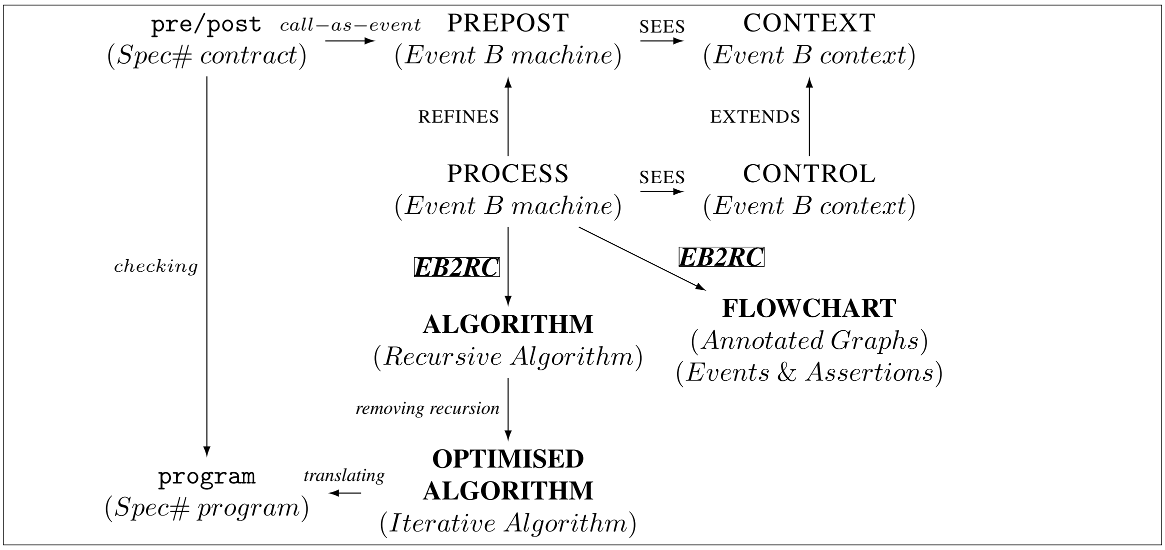


Figure 14. *An overview of our integrated development framework to combine program refinement with program verification ((Cheng* et al. *2016))*

Turing, A. (1949), On checking a large routine, *in* Conference on High-Speed Automatic Calculating Machines, University Mathematical Laboratory,Cambridge.