Teaching formal techniques with Event-B and Rodin*

Dominique Méry

LORIA & *Université de Lorraine*
Vandœuvre-lès-Nancy, France
dominique.mery@loria.fr

Abstract. This document accompanies a presentation as part of the Nantes Université scientific days, and more specifically, the day in honor of Jean-Raymond Abrial and his scientific work. It presents the technical elements of the Event-B language and method taught at the Université de Lorraine, and in particular to students at the Ecole Telecom Nancy. The courses cover the foundations of formal methods and the modeling of software-intensive systems. The Event-B method and its Rodin environment are the tools of choice for verifying sequential algorithms and designing reactive systems incrementally and reliably. We are specifically referring to the models and algorithms course, the modeling, verification, and experimentation course, and the algorithms for parallel and distributed systems course. The course, "Modeling and Formal Design of Computer Systems," is a required course in the specialties of "Software Engineering" and "Embedded Systems and Software" for Telecom Nancy students, students in the Master of Computer Science program at the Université de Lorraine, and students in the Master of Computer Science program at Maynooth University. Teaching resources are available for free consultation.

Keywords: Verification, Safety, Program Properties, Event-B

1 From verification of software-based systems to design of correct-by-construction systems

In September 1969, I was in sixth grade at Lycée Fabert in Metz. I had just left a small village with a single class in a single primary school. That year, I encountered set theory and other new concepts for the first time. These concepts were new not only to me as a student but also to my math teacher, who was learning alongside us schoolchildren. We used Galion's potato diagrams and other Venn and Carroll diagrams. Mathematics was declared modern, and problems involving faucets or trains had become relics that our parents told us about. I

* Supported by the ANR DISCONT Project (ANR-17-CE25-0005) and by the ANR EBRP Plus Project (ANR-19-CE25-0010). The paper is also related to educational resources provided to students of the University of Lorraine (https://mery54.github.io/teaching) and they are updated according to the new sessions. Edited on May 16, 2025.

had already become quite familiar with the basic vocabulary of set-theoretical notation, and the mathematics taught in school had evolved over time. Set theory and the famous potatoes had become commonplace.

I had to wait until my DEA in Computer Science to resume my adventures in set theory. The teaching team for the main course (Claude Pair and Jean-Pierre Finance) introduced us to the Z language, which, contrary to what its name suggests, is not the latest language. The DEA was coming to an end, and I was already discovering the issues of systematic program construction using the deductive method promoted by the Castor team. I was also discovering issues of specifications and modeling, particularly in the algebraic framework.The DEA ended with a research internship, during which I became familiar with the dynamic logics of Pratt [50] and Harel [24]. Kozen's work [27].

Next year, my PhD focused on the correctness and semantical completeness of an axiomatic method for proving properties of parallel programs with fairness assumptions. I worked under the supervision of Patrick Cousot, based on the document published by Owicki and Lamport [47]. The work in this thesis and my interactions with Patrick resulted in the mechanization of the method [42] developed with financial support from CNET and the first version of Isabelle. Experiments with this CROCOS tool on small examples show that schemas can be implemented with Isabelle and its powerful functional framework. These schemas allow proofs of invariance and liveness properties to be derived using an intermediate language of the type (*condition*,*action*), but they also highlight the possibility of developing correct by construction (CbC) programs. The goal at that time was to produce proof lattices in the style of Owicki and Lamport. Other approaches were undoubtedly published and developed, such as Back's actions systems [7–11] or UNITY [15], an approach that combines both action systems and temporal logic. It is clear that the work on CROCOS was influenced by Manna and Pnueli's work on temporal verification of systems [32–35, 49]. CROCOS's design set the direction for correctness by construction.

The works of Hoare, Dijkstra, Lamport, Jones, Bjørner, P., and R. Cousot has had a major influence on my scientific approach. Meeting Jean-Raymond Abrial in 1988 led me to rediscover set theory and a very simple notation for reasoning about state systems. These systems would later be known as reactive systems and then the famous CPS.

This period led me to the first conference on the B method, held in Nantes from November 24-26, 1996. Henri Habrias organized the conference, where Jean-Raymond Abrial laid the foundations for what would become the method Event-B [1]. My contribution [45] aimed to show the semantical proximity of B and TLA+. I included one caveat regarding the proof environment for TLAk; at the time, it did not have a syntax analyzer. This work is ongoing and has been published in iFM [37]. It lays the foundations for my courses at ESIAL, now Telecom Nancy, part of the University of Lorraine, which succeeded the University Henri Poincaré Nancy 1. The adventure Event-B began, and my work on parallel program verification quickly proved invaluable in the development of research on the design of reactive systems correct by construction.

This note will discuss several educational experiments related to Event-B that originated in research work. Later, we will see that the reverse is also true. We describe three aspects of Event-B that we have integrated into our teaching:

- the verification of contracts in Event-B and Rodin.
- the development of sequential algorithms according to the correctness by construction paradigm.
- the development of distributed algorithms according to the correctness by construction paradigm.

We also aim to show the importance of Event-B in the evolution of teaching and learning a formal method and its environment. Figure 1 illustrates the dependency graph among courses on formal methods using Event-B and Rodin with the following acronyms MALG standing for Models and Algorithms, ASPD standing for Algorithmics for Parallel and Distributed Systems, MOSOS standing for Modelling Software-based Systems and MOVEX standing for Modelling, verification and experimentation. The course MOSOS is implemented at Telecom Nancy as MCFSI, at Faculty of Science and Technology as Modelling and at Maynooth University as a part of a course in the MsC programme. The dashed red line means that the source course validates the target course.

2 Checking contracts using Event-B and Rodin

2.1 Summary on Event-B

Event-B is a correct-by-construction, stated-based formal modelling language for system design [3]. First-order logic (FOL) and set theory underpin the Event-B modelling language. The design process consists of a series of refinements of an abstract model (specification) leading to a final concrete model. Refinement progressively contributes to add design decisions to the system. We are not considering the refinement relation in this paper. Three components define Event-B models: *Contexts*, *Machines*, and *Theories*. However, we will not use *Theories* [48] and will not describe this concept.

A *Context* (Figure 2) is the static part of a model. It is used to set up definitions, axioms, and theorems needed to describe required concepts. *Carrier sets* s defining algebraically new types (possibly constrained in axioms or other extending contexts), *constants* c, *axioms* $AX(s, c)$ and *theorems* $TH(s, c)$ are introduced.

A *machine* (Figure 2) describes the dynamic part of a model as a transition system. A set of possibly parameterised and/or guarded events (transitions) modifying a set of state variables (state) represents the core concepts of a machine. *Variables* x, *invariants* $I(s, c, x)$, *theorems* $S(s, c, x)$, *variants* $V(x)$, and *events* Event e (possibly guarded by G and/or parameterised by α) are defined in a machine. *Invariants* and *theorems* formalise system safety properties while *variants* define convergence properties (reachability).

Before-After Predicates (BAP) express state variables changes using prime notation x' to record the new value of a variable x after a change. The *"becomes*

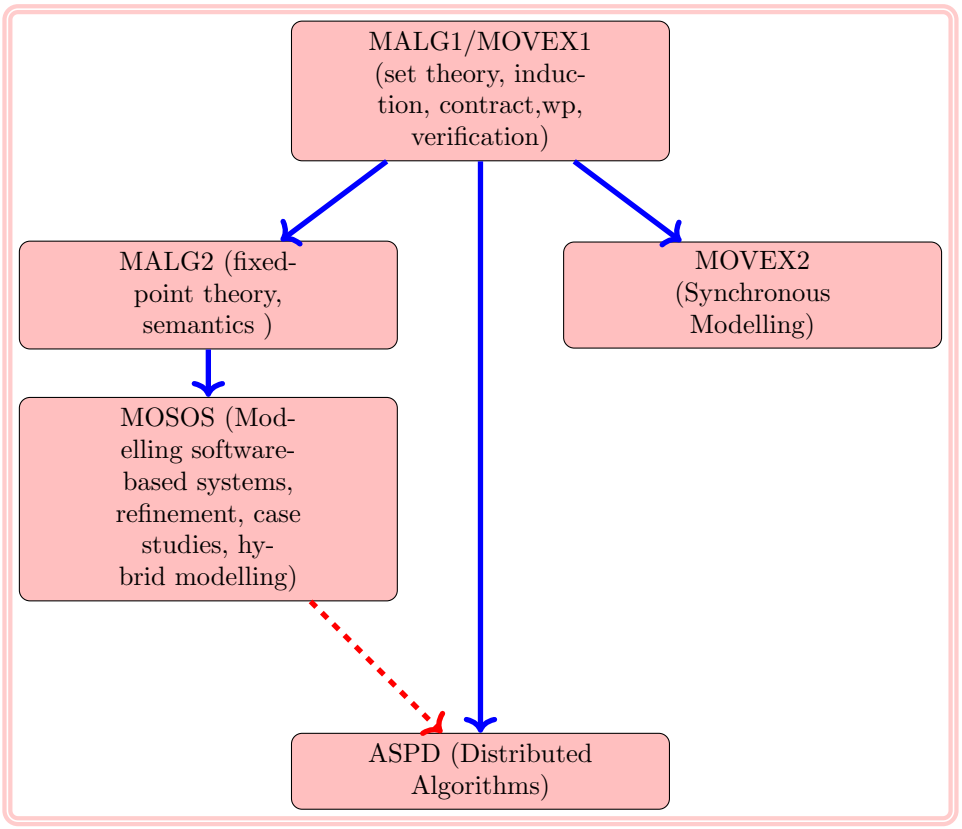


Fig. 1: Dependency Graph for Courses on Formal Methods at University of Lorraine

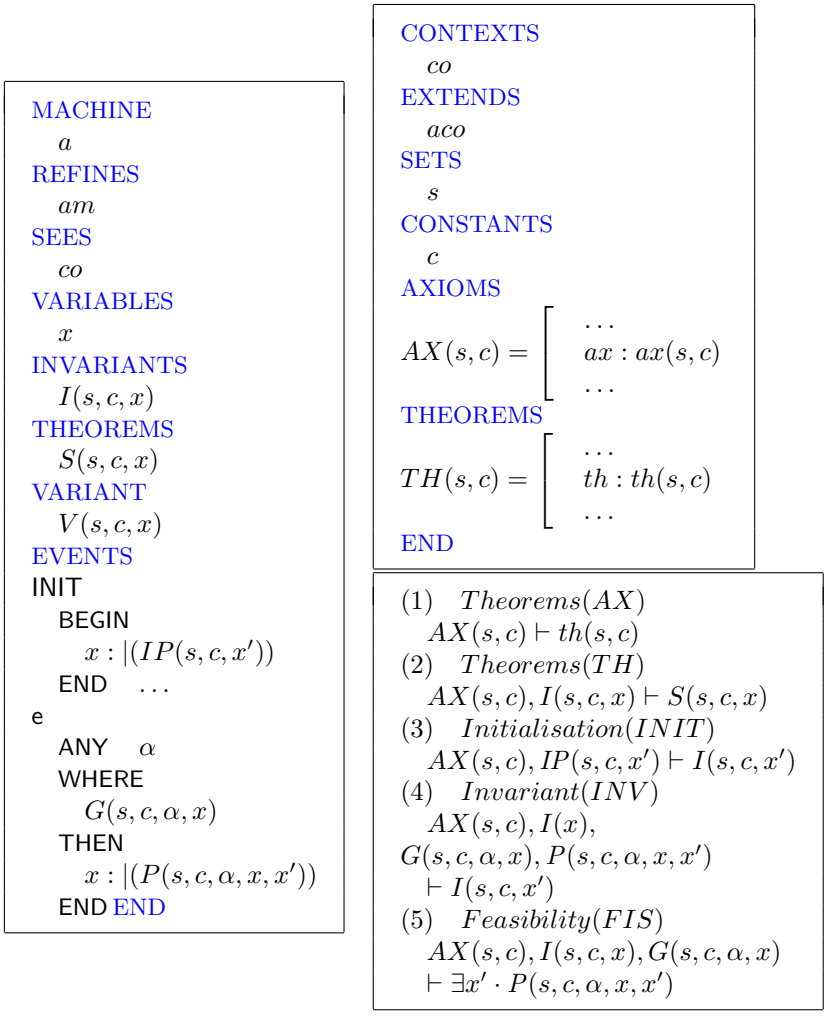


Fig. 2: Event-B structures: Context & Machine and Proof Obligations

such that" :| substitution is used to define the next (transition or event) value of a state variable. We write $x :| P(s, c, \alpha, x, x')$ to express that the next value of x (denoted by x') satisfies the predicate $P(s, c, \alpha, x, x')$ defined on before and after values of variable x. When a parameter α is involved in a variable the BAP is expressed as $x :| P(s, c, \alpha, x, x')$.

To establish the correctness of an Event-B machine, POs (automatically generated from the calculus of substitutions) need to be proved.

The main proof obligations (POs), relevant for this paper, are listed in the table of figure 2. They require to demonstrate the context and machine theorems (1,2), initialisation (3), invariant preservation (4) and event feasibility (5).

Rodin[1] is an open source, Eclipse-based Integrated Development Environment for modelling in Event-B . It offers resources for model editing, automatic PO generation, project management, refinement, proof, model checking, model animation, and code generation. Event-B theories extension is available in the form of a plug-in, developed for the Rodin platform. Many provers like predicate provers, SMT solvers, are plugins for Rodin.

2.2 Programming Constructs

Programming constructs are classical constructs as assignment ($\mathsf{v} := \mathsf{f}_{ell,\ell'}(\mathsf{v})$), skip statement skip, conditional statement (if cond(v) S_1 else S_2 fi) and iterative statement (while cond(v) do S od). We use these constructs for expressing programs or algorithms which are annotated possibly by labels.

ℓ_0 :
$k := 0$;
ℓ_1 :
$co := 0$;
ℓ_2 :
while $(k < n)$ do
  ℓ_3 :
    if $(k \,\%2 \;==\; 0)$
  ℓ_4 :
      $co := co + k + 1$;
    fi;
  ℓ_5 :
    $k := k + 1$;
od;
  ℓ_6 :
$ro := co$;
  ℓ_5 :

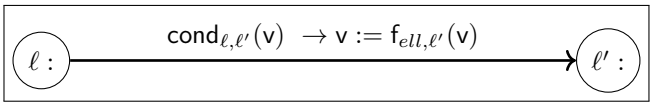
There different ways to annotate algorithms. One can assign a label $\ell \in L$ to each statement:

- v := $\mathsf{f}_{ell,\ell'}$(v) is labelled as follow
  ℓ :
    v := $\mathsf{f}_{\ell,\ell'}$(v);
- if cond(v) S_1 else S_2 fi is labelled as follow
  ℓ :
    if cond(v) S_1 else S_2 fi
- while cond(v) do S od is labelled as follow
  ℓ :
    while cond(v) do S od

The annotation process uses one label at most one time. For instance, the following annotation of a small algorithm is a small example in the left box.

Each pair of successive labels ℓ, ℓ' is interpreted by a condition denoted $\mathsf{cond}_{\ell,\ell'}(v)$ and an assignment $\mathsf{v} := \mathsf{f}_{\ell,\ell'}(\mathsf{v})$. A flowchart can be derived following the next diagram:

[1] `http://www.event-b.org/index.html`



In our paper, we assume that the programming language is $\mathcal{PL}$ and one can derive a flowchart from the annotated algorithmic notation.

2.3 Programming by Contract

Programming by contract [44] is based on a contract between the software developer and software user - in Meyer's terms the supplier and the consumer. Every process starts with a precondition that must be satisfied by the consumer and it ends with postconditions which the supplier guarantees to be true (if and only if the precondition was met). The contract is defined by two assertions a precondition and a postcondition; the algorithm is annotated. The postcondition establishes a relation between the initial values of variables and the final values of variables.

We use two languages a programming or algorithmic language $\mathcal{PL}$ for expressing algorithms and an assertion language denoted $\mathcal{AL}$ for expressing annotations. A contract is a pair $(pre(v_0), post(v_0, v_f)$ where $pre(v_0)$ states the specification of input values denoted v_0 and v_0 is the initial value of the variable v. $post(v_0, v_f)$ is the relation between the initial values v_0 of v. and the final values of v.

We adopt a convention to make our explanation as clear as possible and we will denote *non-logical* (or computer or flexible [28]) variables by strings using the font as v, Tax, Result, ... and logical variables by strings using the font as v, Tax, $Result$, ... The convention is adapted from Patrick Cousot's comments [17] on making a distinction between a value of a computer variable and the computer variable itself.

A program or an algorithm P over variables v *fullfills* a contract $(pre(v_0), post(v_0, v_f)$, when:

- P transforms a non-logical variable v from an initial value v_0 to a final value v_f: $v_0 \xrightarrow{\mathrm{P}} v_f$
- v_0 satisfies pre: $\mathsf{pre}(v_0)$ and v_f satisfies a relation post : $\mathsf{post}(v_0, v_f)$
- $\mathsf{pre}(v_0) \wedge v_0 \xrightarrow{\mathrm{P}} v_f \Rightarrow \mathsf{post}(v_0, v_f)$

We will denote a *contract* for P as follows.

contract P
variables v
requires $pre(v_0)$
ensures $post(v_0, v_f)$

The contract has a name which is the name of the program under construction. That program may be implicit or explicit. It may be a program which is not yet existing and we may follow the *refinement*-based approach or a direct construction.

As pointed out by C. Jones in his speech accepting the FM fellow, a postcondition is a relation between the current value of variables and their initial values. P.

and R. Cousot [18, 19] give detail on induction principles of the proposed methods as Hoare, Manna ... and partition invariance proof methods into assertional ones and relational ones; they explain how they are related using a cube representation and Galois connections for expressing these relationships. We consider the following general interpretation of $P(x)$ by expressing it as $x \in \tilde{P}$ from a correspondance between a predicate and the set of values validating this predicate. For ease of syntax we leave out the ˜ symbol.

CONTEXT $C0$
SETS
D
CONSTANTS
$v0, vf, post, pre$
AXIOMS
$def1 : pre \subseteq D$
$def2 : post \subseteq D \times D$
$pre(v0) : v0 \in pre$
$post(v0, vf) : v0 \mapsto vf \in post$
END

The translation in Rodin is simple and we have to define the domain of variables namely D. We have chosen a general form. The context is used for expressing theorems required for deriving the postcondition. The context SQUARE-C0 corresponds to the contract for computing the square of a positive integer. In this case, we have to define a sequence which is supporting the computation of the square of a natural number.

Example 21. *Contract in Event-B for square computation*

CONTEXT $SQUARE - C0$
CONSTANTS
$n0, r0, nf, rf$
AXIOMS
$pre(n0, r0) : n0 \in \mathbb{N} \land r0 \in \mathbb{Z}$
$post(n0, r0, nf, rf) : \begin{array}{l} nf = n0 \\ rf = n0 * n0 \end{array}$
END

The contract SQUARE is expressing the relation of computation of the square of n.

contract SQUARE
variables *n,r*
requires $n0 \in \mathbb{N} \land r0 \in \mathbb{Z}$
ensures $\begin{array}{l} nf = n0 \\ rf = n0 * n0 \end{array}$

A contract can be extended by the definition of an algorithmic section which is describing the computation process itself. The annotation of the algorithmic section is not required but it can help the proof process and it will be generally checked using *verification conditions* following the *Floyd-Hoare method [21, 25]*. The contract is stated and a code is added.

contract P
variables v
requires $pre(v_0)$
ensures $post(v_0, v_f)$

begin
$0 : P_0(v_0, v)$
S_0
...
$i : P_i(v_0, v)$
...
S_{f-1}
$f : P_f(v_0, v)$
end

Verification conditions are listed as follows:

- (initialisation)
  $pre(v_0) \wedge v = v_0 \Rightarrow P_0(v_0, v)$
- (finalisation)
  $pre(v_0) \wedge P_f(v_0, v) \Rightarrow post(v_0, v)$
- (induction)
  For each labels pair ℓ, ℓ'
  such that $\ell \longrightarrow \ell'$, one checks that,
  for any value $v, v' \in \mathsf{D}$
  $$\left(\begin{pmatrix} pre(v_0) \wedge P_\ell(v_0, v)) \\ \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \end{pmatrix} \right), \\ \Rightarrow P_{\ell'}(v_0, v')$$

Three kinds of verification conditions should be checked and we justify the method in the full version..

The method checks that the annotation denoting verification is correct. An Event-B machine (see Fig. 3) is built from the extended contract.

MACHINE M
SEES $C0$
VARIABLES
v, pc
INVARIANTS
typing : $v \in D$
control : $pc \in L$
...
atℓ : $pc = \ell \Rightarrow P_\ell(v0, v)$
...
$th1 : pre(v_0) \wedge v = v_0 \Rightarrow P_0(v_0, v)$
$th2 : pre(v_0) \wedge P_f(v_0, v)$
$\Rightarrow post(v_0, v)$
...
END
...
END

MACHINE M
EVENTS
INITIALISATION
BEGIN
$(pc, v) : | \begin{pmatrix} pc' = l0 \wedge v' = v0 \\ \wedge pre(v0) \end{pmatrix}$
END
...
$e(\ell, \ell')$
WHEN
$pc = \ell$
$cond_{\ell,\ell'}(v)$
THEN
$pc := \ell'$
$v := f_{\ell,\ell'}(v)$
END
...
END

Fig. 3: Event-B machine for checking contract

The machine M (Fig. 3) has variables as v for modelling v and we add a control variable pc whose values are in L. For each label ℓ, one adds an implication defining the current state, when, the control is at ℓ. The initialisation of the variables is defined by the precondition and the initial possible values $v0$. Events are defined for each pair of labels (ℓ, ℓ') and is modelling the flowchart

derived from the algorithm. In Section 3, we give examples which illustrate the methodology.

2.4 The Methodology in Action

We have described concepts required for using Event-B as support for expressing verification conditions that have been given. We justify verification conditions in the full version . This example does not illustrate the prover's performance, but rather the simplicity of the translation. We developed this translation based on a tutorial session during which I tested it without proving its correctness. We wanted to check the manual verification process, which works by applying simple rewriting and sequence simplification rules. The method allows you to teach the Event-B language and to state the need of refinement.

Obtaining the invariant simply involves copying annotations as a conjunction of local annotations: invariants inv1 and inv2 are type invariants, inv3, inv4 and inv5 come from the contract SIMPLE. x_0 is the input value of x and x_f is the final value of x at ℓ_2.

contract SIMPLE
variables x
requires $x_0 \in \mathbb{N}$
ensures $x_f = 0$
begin
$\ell_0 : \{0 \leq x \leq x_0 \wedge x_0 \in \mathbb{N}\}$
while $0 < x$ **do**
  $\ell_1 : \{0 < x \wedge x \leq x_0 \wedge x_0 \in \mathbb{N}\}$
  $x := x - 1$;
od
$\ell_2 : \{x = 0\}$end

INVARIANTS
  $inv1 : x \in \mathbb{N}$
  $inv2 : l \in L$
  $inv3 : l = l0 \Rightarrow$
  $0 \leq x \wedge x \leq x0 \wedge x0 \in \mathbb{N}$
  $inv4 : l = l1 \Rightarrow$
  $0 < x \wedge x \leq x0 \wedge x0 \in \mathbb{N}$
  $inv5 : l = l2 \Rightarrow x = 0$
  $requires : x0 \in \mathbb{N} \wedge x = x0$
  $\Rightarrow x = x0 \wedge x0 \in \mathbb{N}$
  $ensures : x = 0 \wedge x = x0$
  $\Rightarrow x = 0$

The writing process is straightforward for students. They write an invariant and then the events corresponding to the observation of the calculation described by the algorithm. Students concentrate mainly on the formal writing of the annotations and only discover the result of the proof when the file is saved.

Event *Init*
THEN
  $act1 : x := x0$
  $act2 : l := l0$

Event $el0l1$
WHEN
  $grd1 : l = l0$
  $grd2 : 0 < x$
THEN
  $act1 : l := l1$

Event $el0l2$
WHEN
  $grd1 : l = l0$
  $grd2 : \neg(0 < x)$
THEN
  $act1 : l := l2$

Event $el1l0$
WHEN
  $grd1 : l = l1$
THEN
  $act1 : l := l0$
  $act2 : x := x - 1$

This example is very simple and consists of one iteration which stops when the value of x is zero. It does not pose problem with Rodin and the proofs are derived at the same time as the at the same time as writing the elements of the invariant and the events. Proofs obligations are discharged by the proof tools while editing the Event-B machine in the Rodin platform.

2.5 Summary and comments

Verification of sequential algorithms has been taught since 1993, and for the first ten years, no tools were used. Leslie Lamport's integration of the TLC model checker into the TLA$^+$ language was a pivotal moment. It was then that we could finally utilize this environment to develop algorithms and properties for verification in TLA$^+$. The notation was challenging for the students, but we had become a site developing formal technology teaching with TLA$^+$. Then, a few years later, we had the idea of translating Floyd's verification conditions into the form of a machine Event-B . The idea of such a machine emerged organically during the tutorial session. We enriched our course with elements for mechanizing verification conditions and presented these ideas at the FMTEA conference [41]. This highlights the inductive nature of the machine invariant. The use of the language clearly exceeds the initial objectives, particularly in terms of refinement. We must also address the questions raised by the two languages TLA$^+$ and Event-B and their implementation of set theory. In TLA$^+$, the basic concept is the total function, and in Event-B , the basic concept is the set (relations, partial functions). However, the designers of these languages don't have the same objectives. Leslie Lamport proposes a modeling language based on his experience with distributed algorithms, while Jean-Raymond Abrial is interested in reactive systems. We have integrated these elements into the MALG1 and MOVEX1 courses (see figure 1).

3 The Correct by Construction Methodology in Action

In our second-year course, we illustrated the notion of contract by translating it quite easily into Event-B to derive the proof of its correctness. The next step is to show students that it is more practical to develop an algorithmic solution from the contract. We will propose a pattern to students, based on the use of refinement. This design pattern is illustrated here by a small example. We use this example in our course, and it is part of a collection of sequential algorithms developed according to these ideas We consider the calculation of 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

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

$$\text{REQUIRES} \begin{pmatrix} n_0 \in \mathbb{N} \land n_0 \neq 0 \\ v_0 \in 1..n_0 \to \mathbb{Z} \end{pmatrix}$$

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

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: $r = \sum_{k=1}^{k=n_0} v_0(k)$. 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 $\sum_{k=1}^{k=i} v_0(k)$. 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
$axm1 : n_0 \in \mathbb{N}$
$axm2 : n_0 \neq 0$
$axm3 : v_0 \in 1 \mathinner{..} n \rightarrow \mathbb{Z}$
$axm4 : u \in \mathbb{N} \rightarrow \mathbb{Z}$
$axm5 : u(0) = 0$
$axm6 : \forall i \cdot i \in \mathbb{N} \wedge i > 0 \wedge i \leq n_0 \Rightarrow u(i) = u(i-1) + v_0(i)$
$axm7 : \forall i \cdot i \in \mathbb{N} \wedge i > n_0 \Rightarrow u(i) = u(n_0)$
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.

3.1 Specification of the problem to solve

MACHINE $S1$
SEES $S0$
VARIABLES
r, v, n
INVARIANTS
$inv1 : r \in \mathbb{Z}$
$inv2 : v \in 1..n_0 \rightarrow \mathbb{Z}$
$inv3 : n \in \mathbb{Z}$
$inv4 : n = n_0 \wedge v = v_0$
INITIALISATION
BEGIN
$act1 : r :\in \mathbb{Z}$
$act2 : n := n_0$
$act3 : v := v_0$
END
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 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 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 final is therefore simply an assignment of the value $u(n)$ to r.
We can express it as a HOARE triple:
$\{n = n_0 \wedge v = v_0 \wedge n_0 > 0 \wedge v_0 \in 1..n_0 \rightarrow \mathbb{N}\}\mathsf{SUM}\{r = u(n_0)\}$.

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.

3.2 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 ($uu \in 0 \mathinner{..} n_0 \pfun \mathbb{Z}$) 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
$inv1 : i \in \mathbb{N}$
$inv2 : i \geq 0$
$inv3 : i \leq n$
$inv4 : uu \in 0 \mathinner{..} n \pfun \mathbb{Z}$
$inv5 : dom(uu) = 0 \mathinner{..} i$
$inv6 : n \notin dom(uu) \Rightarrow i < n$
$inv7 : dom(uu) \subseteq dom(u)$
$inv8 : \forall k \cdot \left(\begin{array}{l} k \in dom(uu) \\ \Rightarrow \\ uu(k) = u(k) \end{array} \right)$
$inv9 : n = n_0 \land v = v_0$
VARIANTS $vrn : n - i$

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

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

3.3 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* [15] 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
$inv1 : cu \in \mathbb{Z}$
$inv2 : cu = u(i)$
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

final
REFINES final
WHEN
$grd1 : n \in dom(uu)$
$grd2 : i = n$
THEN
$act1 : r := cu$
END
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.

3.4 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

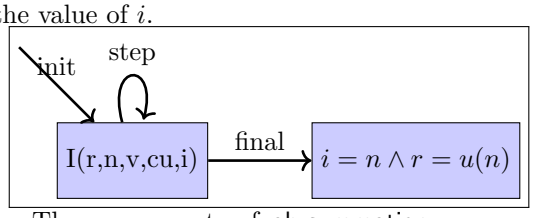
called theorems in the S4 machine. These properties are proved from the properties of previous refined machines. We have thus obtained a machine comprising an initialisation and two events:

- 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
$inv1 : cu = u(i)$
$inv2 : i \leq n$
INITIALISATION
BEGIN
$act1 : r :\in \mathbb{Z}$
$act2 : n := n_0$
$act3 : v := v_0$
$act4 : uu := \{0 \mapsto 0\}$
$act5 : i := 0$
END

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

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 [52] plugin which produces the above algorithm from the ab-summation archive. Jean-Raymond Abrial [3, 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 [52].

VARIABLES n, v, r

DEFINITIONS

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

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

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);
```

3.5 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 [52] and the use of Jean-Raymond Abrial [3, 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*.

3.6 Summary and comments

The MOSOS course (see figure 1) is designed to provide a comprehensive introduction to the elements of the language Event-B and its environment Rodin. The proof of verification conditions linked to machines and refinements are studied through the development of sequential algorithms. We are proud to present Jean-Raymond Abrial's groundbreaking transformations and the cutting-edge

technique we have developed that has led to the creation of recursive algorithms [16, 38, 40]. The idea is simple but elegant, and mechanizes C. Morgan's refinement calculus. In MALG1 and MOVEX1, students have been given a course on contract programming with verification using Frama-c as well as Rodin, and have an example of correctness by construction. This course provides students with extensive guidance through methodological guides, ensuring a seamless experience with Rodin refinement and environment. Students must develop a sequential algorithm correct by construction. This will allow them to delve deeper into the concepts of refinement while combining the Rodin and Frama-c [12] environments. Students evaluate both verification methods by conducting an in-depth analysis of the problem according to refinement experiment with abstractions at various levels of refinement.

4 The Service-as-Event Paradigm

Now, we intend to handle concurrent and distributed algorithms corresponding to different programming paradigms as message-passing or shared-memory or coordination-based programming. C. Jones [26] develops the rely/guarantee concept for handling (possible and probably wanted) interferences among sequential programs. Rely/guarantee intends to make *implicit* [4] interferences as well as cooperation proofs in a proof system. In other methods as Owicki and Gries [46], the management of non-interference proofs among annotated processes leads to a important amount of extra proof obligations: checking interference freeness is explicitly expressed in the inferences rules. When considering an event as modelling a call of function or a call of a procedure, we implicitly express a computation and a sequence of state. In a joint work [43] we propose a temporal extension of Event-B to express liveness properties. The extension is a small bridge between Event-B and TLA/TLA$^+$ [28] with a refinement perspective. As C. Jones in rely/guarantee, we express implicit properties of the environment on the protocol under description by extending the call-as-event paradigm by a service-as-event paradigm. In [5, 6], the service-as-event paradigm is explored on two different classes of distributed programs/algorithms/applications: the snapshot problem and the self-healing P2P by Marquezan et al [36]. The self-healing problem is belonging to the larger class of self-$\star$ systems [20].

In previous patterns, we identify one event which *simulates* the execution of an algorithm either as an iterative version or as a recursive version. We are now introducing and illustrating the distributed pattern which is a representative of the service-as-event paradigm.

4.1 The PCAM pattern

Coordination [13] is a paradigm that allows programmers to develop distributed systems; web services are using this paradigm for organising interactions among services and processes. In parallel programming, coordination plays also a central role and I. Foster [22] has proposed the PCAM methodology for designing

concurrent programs from a problem statement: PCAM emphasizes a decomposition into four steps corresponding to analysis of the problem and leading to a machine-independant solution. Clearly, the goal of I. Foster is to make concurrent programming based on abstractions, which are progressively adding details leading to specific concurrent programming notation as, for instance MPI (http://www.open-mpi.org/). The PCAM <methodology identifies four distinct stages corresponding to a Partition of identified tasks from the problem statement and which are concurrently executed. A problem is possibly an existing complex C or Fortran code for a computing process requiring processors and concurrent executions. Communication is introduced by an appropriate coordination among tasks and then two final steps, Agglomeration and Mapping complete the methodology steps. The PCAM methodology includes features related to the functional requirements in the two first stages and to the implementation in the two last stages. I. Foster has developed the PCAM methodology together with tools for supporting the implementation of programs on different architectures. The success of the design is mainly due to the coordination paradigm which allows us to freely organise the stages of the development.

The PCAM methodology (Fig. 4) includes features related to the functional requirements in the two first stages and to the implementation in the two last stages. The general approach is completely described in [39].

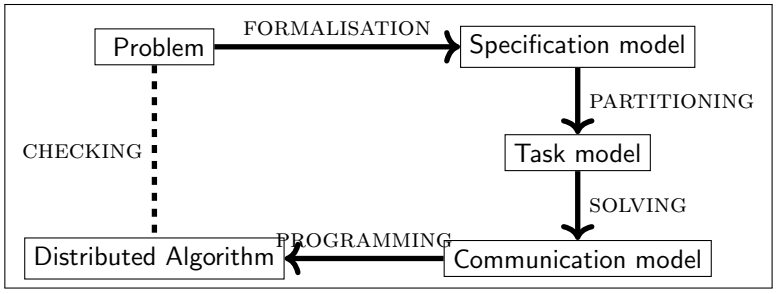


Fig. 4: The PCAM Methodology

We consider the two first stages (Partitioning, Communication) for producing state-based models satisfying functional requirements and which will be a starting point for generating a concurrent program following the AM last suffix. We have described a general methodology for developing *correct-by-construction* concurrent algorithms and we have developed a solution specified by a unique event.

4.2 The distributed pattern

An event can be observed in a complex environment. The environment may be active and should be expressed by a set of events which are simulating the

environment. Since the systems under consideration are reactive, it means that we should be able to model a service that a system should ensure. For instance, a communication protocol is a service which allows to transfer a file of a process A into a file of a process B.

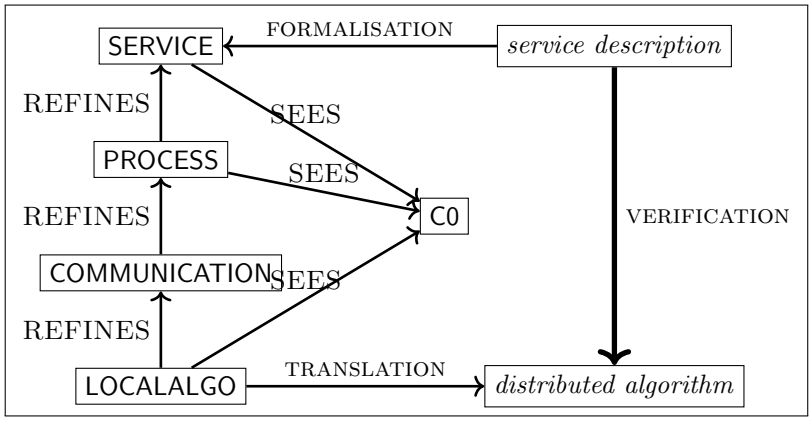


Fig. 5: The distributed pattern

Fig. 5 sketches the distributed pattern. The machine SERVICE is modelling services of the protocol; the machine PROCESS is refining each service considered as an event and makes the (computing) process explicit. The machine COMMUNICATION is defining the communications among the different agents of the possible network. Finally the machine LOCALALGO is localizing events of the protocol. The distributed pattern is used for expressing *phases* of the target distributed algorithm (for instance, requesting mutual exclusion) and to have a separate refinement of each phase. We sketch the service-as-event paradigm as follows. We consider one service. The target algorithm $\mathcal{A}$ is first described by a machine M0 with variables x satisfying the invariant $I(x)$.

The first step is to list the services e $S \ \widehat{=} \ \{s_0, s_1, \ldots, s_m\}$ provided by the algorithm $\mathcal{A}$ and to state for each service s_i a liveness property $P_i \rightsquigarrow Q_i$. We characterise by $\Phi_0 \ \widehat{=} \ \{P_0 \rightsquigarrow Q_0, P_1 \rightsquigarrow Q_1, \ldots, P_m \rightsquigarrow Q_m\}$. We add a list of safety properties defined by $\Sigma_0 = \{Safety_0, Safety_1, \ldots, Safety_n\}$. An event is defined for each liveness property and standing for <mutualthe eventuality of e by a fairness assumption which is supposed on e. Liveness properties can be visualised by assertions diagrams helping to understand the relationship among phases.

The second step is its refinement M1 with variables y glued properties in by $J(x, y)$ using the Event-B refinement and using the REF refinement which is defined using the temporal proof rules for expanding liveness properties. $P \rightsquigarrow Q$ in Φ_0 is proved from a list of Φ_1 using temporal rules. For instance, $P \rightsquigarrow Q$

in Φ_0 is then refined by $P \rightsquigarrow R, R \rightsquigarrow Q$, if $P \rightsquigarrow R, R \rightsquigarrow Q \vdash P \rightsquigarrow Q$. If we consider C as the context and M as the machine, C, M satisfies $P \rightsquigarrow Q$ and C, M satisfies $\Box Safety$. We use a temporal semantics relating contexts, machines and properties [43]. The link called LIVE expresses the satisfaction relationship. The diagram in table MITEX1 is summarising the relationship among models.

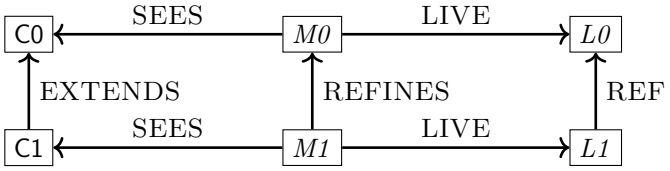


Diagram 1: The REF Refinement

The liveness property $P \xrightarrow{e} Q$ means that

- $\forall x, x' \cdot P(x) \wedge I(x) \wedge BA(e)(x, x') \Rightarrow Q(x')$
- $\forall x \cdot P(x) \wedge I(x) \Rightarrow (\exists x' \cdot BA(e)(x, x'))$
- $\forall f \neq e \cdot \forall x, x' \cdot P(x) \wedge I(x) \wedge BA(f)(x, x') \Rightarrow (P(x') \vee Q(x'))$

$P \xrightarrow{e} Q$ expresses implicitly that tyhe event e is under weak fairness. Each liveness property $P_i \rightsquigarrow Q_i$ in Φ_0 is modelled by an event:

$$\mathsf{e}_i \ \widehat{=}\ \ \mathsf{WHEN}\ \ P_i(x)\ \ \mathsf{THEN}\ \ x :|Q_i(x')\ \ \mathsf{END}$$

We can add some fairness assumption over the event:

- $P_i \xrightarrow{\mathsf{e_i}} Q_i$ with weak fairness on e ($WF_x(\mathsf{e_i})$),
- $P_i \overset{\mathsf{e_i}}{\Rightarrow} Q_i$, with strong fairness on e ($SF_x(\mathsf{e_i})$).

If we consider the leader election protocol [2], we have the following elements:

- **Sets**: ND (set of nodes).
- **Constants :** g is acyclic and connected ($acyclic(g) \wedge connected(g)$).
- **Variables :** $x = (sp, rt)$ (sp is a spanning tree of g).
- **Precondition) :**
  $P(x)\ \widehat{=}\ sp = \varnothing \wedge rt \in ND$
- **Postcondition :** $Q(x)\ \widehat{=}\ spanning(sp, rt, g)$

We can express the main liveness property: $(sp = \varnothing \wedge rt \in ND) \rightsquigarrow spanning(sp, rt, g)$ and we define the machine $\mathsf{Leader_0}$ satisfying the liveness property:

$\mathsf{election_0}\ \widehat{=}$
  BEGIN
    $sp, rt :|spanning(sp', rt', g)$
  END

$$\mathsf{C}_0 \xleftarrow{\text{SEES}} \mathsf{Leader}_0 \xrightarrow{\text{LIVE}} (WF_x(\mathsf{election}_0), \{P \rightsquigarrow Q\}$$

We have introduced the service specification which should be refined separately from events of the machine *M0*. The next refinement should first introduce details of a computing process and then introduce communications in a very abstract way. The last refinement intends to localise the events. The model LOCALALGO is in fact an expression of a distributed algorithm. A current work explores the DistAlgo programming language as a possible solution for translating the local model into a distributed algorithm. Y.A. Liu et al [31] have proposed a language for distributed algorithms, DistAlgo, which is providing features for expressing distributed algorithms at an abstract level of abstractions. The DistAlgo approach includes an environment based on Python and managing links between the DistAlgo algorithmic expression and the target architecture. The language allows programmers to reason at an abstract level and frees her/him from architecture-based details. According to experiments of authors with students, DistAlgo improves the development of distributed applications. From our point of view, it is an application of the coordination paradigm based on a given level of abstraction separating the concerns.

4.3 Analysing the mutual exclusion problem

The mutual exclusion problem (MUTEX) [14, 29, 30, 51] is an important and interesting problem to address using the refinement-based methodology. The problem is to find a way to ensure that a finite set of distributed processes are sharing a resource R in an exclusive way under fairness assumption.

The general idea of MUTEX is to manage a *fifo* queue of processes *waiting for* the agreement of the other processes as indicated in the diagram 2.

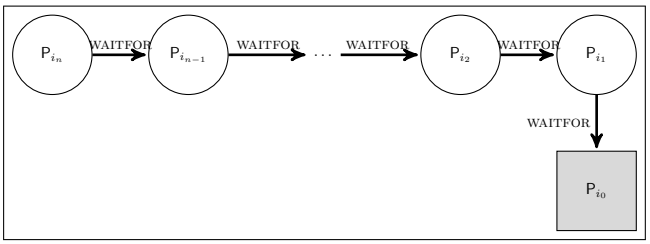


Diagram 2: General idea of the MUTEX problem

The diagram is expressing that the set of *waiting* processes is $\{P_{i_1}, \ldots, P_{i_n}\}$ and the *waitfor* relation over processes is defining a queue which is satisfying the fifo policy. The diagram is indicating that the *square* process P_{i_0} is using the critical section (cs). The *waitfor* relation means that when P *waitfpr* Q is true, P is waiting for the agreement of Q and the agreement of processes related to Q

by the *waitfor* relation. In our diagram, the process P_{i_j} waits for the processes $P_{i_{j-1}}, \ldots P_{i_1}, P_{i_0}$. The protocol is then simply described as follows:

- When the process P_0 exits the critical section, it informs each process that it agrees to give an agreement for entering the critical section.
- When the process P_1 gets the agreement from P_0, it can enter the critical section.

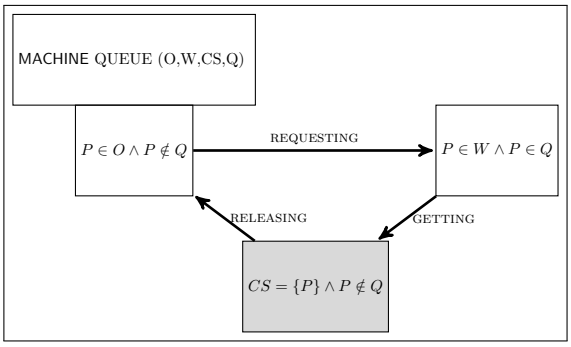


Diagram 3: General steps for machines QUEUE

Figure 6 is applying the distributed pattern (see Figure 5) and the machine SERVICE is describing the protocol by events requesting, getting and releasing. The machine SERVICE is only describing the three main steps of the process for requesting the mutual exclusion. The second machine QUEUE is explicitly using the *fifo* queue and the queue is expressing the required priority for the service MUTEX. However, the model is remaining *centralized* and we have now to introduce the possibility of using messages for requesting and releasing the ressource. The new machine COMQUEUE is introducing new variables as *req* ($p \mapsto q \in req$ means that p is sending a request to q for obtaining the ressource) and *rel* ($p \mapsto q \in rel$ means that p is sending a release to q for releasing the ressource). New events are introduced and are modelling the management of messages among the processes. Finally, the last refinement is implementing the queue by a pair of numbers namely the number for the process and the *osn* variable which is used for defining a total ordering among processes requesting the critical section but in a distributed way. The queue is simulated by these pairs of numbers. Proofs are not easy and models have been checked first using the ProB model-checker of Rodin. We have cited solutions proposed in the literature [14, 29, 30, 51] . The last machine is very close to the two solutions and we have in fact a distributed solution for the MUTEX problem. When teaching the list of algorithms on the MUTEX problem, we use the current solution for showing the *story* of these algorithms.

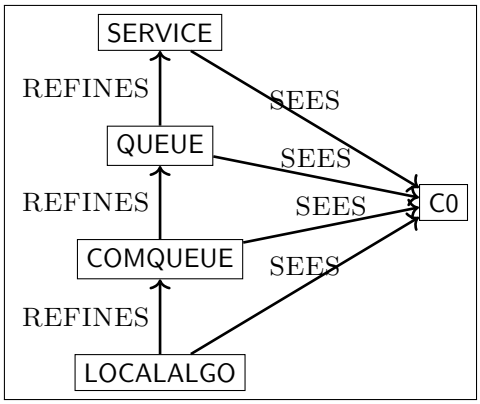


Fig. 6: The mutual exclusion problem

4.4 Summary and comments

In this section, we present a comprehensive report on a seminal body of work that has profoundly influenced pedagogy and research. The design of parallel or distributed programs is a field of research that has led to numerous works and a wide variety of formalisms. Our thesis work focused squarely on parallel programs with fairness hypotheses. This research led to the development of the proof of Ricart and Agrawala's mutual exclusion algorithm DBLP:journals/cacm/RicartA81. We started over when we applied the refinement technique to the design of the distributed leader election algorithm [2]. This triggered a rethinking of the ASPD course, which uses levels of refinement to describe how a distributed algorithm works. This activity was very fruitful and fertile. We proposed a joint ANR RIMEL project with ClearSy and our colleagues at LABRI. In this project, we developed several distributed algorithms. We presented these algorithms to students using Event-B abstractions and using a TLA^+ module. This section presents a clear example of distributed mutual exclusion, which is based on an abstract queue and explains the role of stamps. The red link in the figure clearly explains the influence of the development of mutual exclusion algorithms on the ASPD course. The course takes up the explanation of the models but ignores the technical details. The ASPD course is taken before the MOSOS course, so MOSOS validates ASPD.

5 Final Comments and Conclusion

The evolution of teaching in our courses on software engineering and distributed algorithms is marked by the use of a number of verification tools with master level students. In fourth year, students learn the basic concepts and techniques they will need to know, including how to use logic to model program properties, the semantics of programming languages and induction principles. They are trained

in fundamental tools that they will almost certainly need to use, such as model checking, runtime verification or test management. Additionally, we sought to collaborate with students on authentic programming challenges, which led to the development of a language centered around contracts.

Upon their arrival in the fourth year, we realized that there was a lack of connection between the problem posed by a particular execution, such as calculating the average of two numbers in C, and the proposal to calculate this average for the two numbers equal to the maximum that can be coded. The value returned (-1) is still largely misunderstood. Using Frama-C demonstrated that the RTE plugin facilitated the management of potential errors. The most common issues are managing tools and, in particular, distributing them across different types of operating systems. One solution is to create a virtual machine with the necessary software installed, but this can cause problems on machines that are not powerful enough or too new.

Then we integrate formal methods into a university curriculum; we tteach the Event-B modelling language and use incremental development based on refinement. We handle the notion of contract in 4th year so that we can continue to master the Event-B language and, in particular, to introduce the refinement of formal models. The MALG1 and MALG2 courses are reviewed with the formal expression of refinement and its use. In particular, the incremental development of sequential and distributed algorithms is covered with Event-B and Rodin. The leader election algorithm [2]. was the starting point for this work. It made it possible to explain this algorithm simply to students. Our students cohorts include a significant proportion of students who have learned mathematical proof while preparing for university entrance examinations. These students are well-equipped to play with the tools and interact effectively. Finally, we would like to emphasise the Knaster-Tarski [17] theorem, which also allows us to play with inductions and inductive definitions.

6 Epilogue

The story began in September 1969 with my arrival at the Lycée Fabert (Metz, Moselle, France), and later I met computer science pioneers such as Jean-Raymond Abrial. who proposed a simple, powerful, and elegant formalism for modeling software-based systems. He also proposed a tool for these techniques, notably the famous B Tool and KRT. The story continues, and it is essential to reinforce the tools and, above all, to continue developing case studies, especially for hybrid systems.

Jean-Raymond, I thank you for setting us on a fertile and demanding path.

References

1. J.-R. Abrial. Extending B without Changing it (for Developingg Distributed Systems). In Henri Habrias, editor, *First Conference on the B Method*, 1996. In [23].

2. J.-R. Abrial, D. Cansell, and D. Méry. A Mechanically Proved and Incremental Development of IEEE 1394 Tree Identify Protocol. *Formal Aspects of Computing*, 14(3):215–227, 2003.
3. Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
4. Yamine Aït Ameur and Dominique Méry. Making explicit domain knowledge in formal system development. *Sci. Comput. Program.*, 121:100–127, 2016.
5. Manamiary Bruno Andriamiarina, Dominique Méry, and Neeraj Kumar Singh. Analysis of self-⋆ and P2P systems using refinement. In Yamine Aït Ameur and Klaus-Dieter Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of *Lecture Notes in Computer Science*, pages 117–123. Springer, 2014.
6. Manamiary Bruno Andriamiarina, Dominique Méry, and Neeraj Kumar Singh. Revisiting snapshot algorithms by refinement-based techniques. *Comput. Sci. Inf. Syst.*, 11(1):251–270, 2014.
7. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1998.
8. R. J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1979.
9. R. J. R. Back. Correctness preserving programs refinements: proof theory and applications. Mathematical Centre Tracts 131, Mathematical Centre, Amsterdam, 1980.
10. R. J. R. Back and K. Sere. Stepwise refinement of action systems. In J. L. A van de Snepscheut, editor, *Mathematics for Program Construction*, pages 113–138. Springer-Verlag, june 1989. LNCS 375.
11. Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3(2):73–87, 1989.
12. Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs: the frama-c software analysis platform. *Commun. ACM*, 64(8):56–68, 2021.
13. Nicholas Carriero and David Gelernter. A computational model of everything. *Commun. ACM*, 44(11):77–81, 2001.
14. Osvaldo Carvalho and Gérard Roucairol. On the distribution of an assertion. In Robert L. Probert, Michael J. Fischer, and Nicola Santoro, editors, *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Ottawa, CanadaAugust 18-20, 1982*, pages 121–131. ACM, 1982.
15. K. Mani Chandy and Jay Misra. *Parallel Program Design A Foundation*. Addison-Wesley Publishing Company, 1988. ISBN 0-201-05866-9.
16. Zheng Cheng, Dominique Méry, and Rosemary Monahan. On two friends for getting correct programs - automatically translating event B specifications to recursive algorithms in rodin. In Tiziana Margaria and Bernhard Steffen, editors, *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*, volume 9952 of *Lecture Notes in Computer Science*, pages 821–838, 2016.
17. Patrick Cousot. *Principles of Abstract Interpretation*. The MIT Press, 2021.
18. Patrick Cousot. Calculational design of [in]correctness transformational program logics by abstract interpretation. *Proc. ACM Program. Lang.*, 8(POPL):175–208, 2024.

19. Patrick Cousot and Radhia Cousot. Induction principles for proving invariance properties of programs. In D. Néel, editor, *Tools & Notions for Program Construction: an Advanced Course*, pages 75–119. Cambridge University Press, Cambridge, UK, August 1982.
20. Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.
21. Robert W. Floyd. Assigning meanings to programs. In Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin, editors, *Program Verification: Fundamental Issues in Computer Science*, pages 65–81. Springer Netherlands, Dordrecht, 1993.
22. Ian T. Foster. *Designing and building parallel programs - concepts and tools for parallel software engineering*. Addison-Wesley, 1995.
23. Henri Habrias, editor. *First Conference on the B Method*. University of Nantes, November 24-26 1996. ISBN 2-906082-25-2.
24. David Harel. *First-Order Dynamic Logic*, volume 68 of *Lecture Notes in Computer Science*. Springer, 1979.
25. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
26. Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.
27. Dexter Kozen. On the duality of dynamic algebras and kripke models. In Erwin Engeler, editor, *Logics of Programs, Workshop, ETH Zürich, May-July 1979*, volume 125 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 1979.
28. L. Lamport. A temporal logic of actions. *Transactions On Programming Languages and Systems*, 16(3):872–923, May 1994.
29. Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.
30. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
31. Yanhong A. Liu, Scott D. Stoller, and Bo Lin. From clarity to efficiency for distributed algorithms. *ACM Trans. Program. Lang. Syst.*, 39(3):12:1–12:41, 2017.
32. Zohar Manna and Amir Pnueli. How to cook a temporal proof system for your pet language. In John R. Wright, Larry Landweber, Alan J. Demers, and Tim Teitelbaum, editors, *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*, pages 141–154. ACM Press, 1983.
33. Zohar Manna and Amir Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming*, 4(3):257–289, 1984.
34. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.
35. Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.
36. Clarissa Cassales Marquezan and Lisandro Zambenedetti Granville. *Self-* and P2P for Network Management - Design Principles and Case Studies*. Springer Briefs in Computer Science. Springer, 2012.
37. Dominique Méry. Requirements for a temporal B - assigning temporal meaning to abstract machines... and to abstract systems. In Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors, *Integrated Formal Methods, Proceedings of the 1st International Conference on Integrated Formal Methods, IFM 99, York, UK, 28-29 June 1999*, pages 395–414. Springer, 1999.
38. Dominique Méry. Refinement-based guidelines for algorithmic systems. *Int. J. Softw. Informatics*, 3(2-3):197–239, 2009.

39. Dominique Méry. Playing with state-based models for designing better algorithms. *Future Gener. Comput. Syst.*, 68:445–455, 2017.
40. Dominique Méry. Modelling by patterns for correct-by-construction process. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Modeling - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I*, volume 11244 of *Lecture Notes in Computer Science*, pages 399–423. Springer, 2018.
41. Dominique Méry. Checking contracts in event-b - reporting the introduction and the use of automated tools for verifying software-based systems in higher education. In Emil Sekerinski and Leila Ribeiro, editors, *Formal Methods Teaching - 6th Formal Methods Teaching Workshop, FMTea 2024, Milan, Italy, September 10, 2024, Proceedings*, volume 14939 of *Lecture Notes in Computer Science*, pages 91–105. Springer, 2024.
42. Dominique Méry and Abdelillah Mokkedem. Crocos: An integrated environment for interactive verification of SDL specifications. In Gregor von Bochmann and David K. Probst, editors, *Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, June 29 - July 1, 1992, Proceedings*, volume 663 of *Lecture Notes in Computer Science*, pages 343–356. Springer, 1992.
43. Dominique Méry and Michael Poppleton. Towards an integrated formal method for verification of liveness properties in distributed systems: with application to population protocols. *Softw. Syst. Model.*, 16(4):1083–1115, 2017.
44. Bertrand Meyer. Design by contract. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 1–50, 1991.
45. D. Méry. Machines Abstraites Temporelles. Analyse comprative de B et TLA+). In Henri Habrias, editor, *First Conference on the B Method*, 1996. In [23].
46. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6:319–340, 1976.
47. Susan S. Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.
48. The Rodin PLatform. Theory plug-in. https://wiki.event-b.org/index.php/Theory_Plug-in, 2010.
49. Amir Pnueli. The temporal semantics of concurrent programs. In Gilles Kahn, editor, *Semantics of Concurrent Computation, Proceedings of the International Symposium, Evian, France, July 2-4, 1979*, volume 70 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 1979.
50. Vaughan R. Pratt. Semantical considerations on floyd-hoare logic. In *17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976*, pages 109–121. IEEE Computer Society, 1976.
51. Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, 1981.
52. Neeraj Kumar Singh. EB2ALGO. Plugin Rodin, 2024.