Checking contracts in Event-B*

Reporting the introduction and the use of automated tools for verifying software-based systems in higher education

Dominique Méry

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

Abstract. Verification of program properties such as partial correctness (PC) or absence of errors at runtime (RTE) applies induction principles using algorithmic techniques for checking statements in a logical framework such as classical logic or temporal logic. Alan Turing was undoubtedly the first to annotate programs, namely Turing machines, and to apply an induction principle to transition systems. Our work is placed in this perspective of verifying safety properties of programs which could be executed sequentially or in a distributed manner, with the aim of presenting them as simply as possible to student classes in the context of a posteriori verification. We report on an in vivo experiment using the Event-B language and associated tools as an assembly and disassembly platform for correcting programs in a programming language. We revisit the properties of partial correction and the absence of runtime errors in the context of this experiment, which precedes the use of Event-B as a method of correct design by construction. We have adopted a contract-based approach to programming, which we are implementing with Event-B . A few examples are given to illustrate this pedagogical approach. This step is part of a process of learning both the underlying techniques and other tools such as Frama-C, Dafny and Why3 . . . , which are based on the same ideas.

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

* Supported by the ANR DISCONT Project (ANR-17-CE25-0005) and by the ANR EBRP Plus Project (ANR-19-CE25-0010) on May 30, 2024

1 Introduction

Programming by contract, as outlined in Meyer's work [?, ?, ?], is based on a *contract* between the software developer and software user. In Meyer's terms, the supplier and the consumer are linked by a contract, which expresses a link between a pair (precondition, postcondition) and a possibly annotated algorithm. The objective is to utilise the Event-B modelling language [?] as a framework for expressing verification conditions for contracts and to compare the resulting Rodin-based tool to other existing automated verification tools, such as Frama-C [?].This exercise also introduces the Event-B language and the use of the Rodin [?] and Atelier-B [?, ?, ?] environments.

Our current work is related to our lectures on *modelling, designing, verifying and validating software-based systems* taught in the MsC *Computer Science* at Faculty of Science of the University of Lorraine and in the Computer Engineering Master of the School *Telecom Nancy* of the University of Lorraine. The epistemological concepts were given using the classical blackboard and chalk and progressively we have moved to integrate automated verification techniques and tools as Dafny [?, ?, ?, ?], Why3 [?, ?, ?] and Frama-c [?, ?]. Our list is not exhaustive and our idea is to introduce progressively the concepts of verification using the Floyd-Hoare principle and to show how students can develop a tool for their programming language.

Our main reference is the work of Patrick and Radhia Cousot [?] who analyse the (16) different induction principles for proving program invariance properties. Figure 1 sketches the main steps of our method:

- FORMALISATION Expression of the contract as assertions defined in an Event-B context.
- TRANSLATION Translation of annotations as elements of the invariant and of the basic computation steps between two successive labels as events.

The SEES clause is implemented in the Event-B modelling language by the Rodin platform.

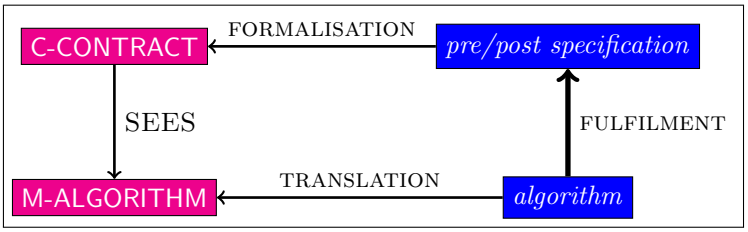


Fig. 1: The verification pattern

In [?], authors are describing a translation in a different way. They use the Dafny tool for checking the Event-B context and machines. In our case, it is

a matter of replacing Dafny and making full use of Rodin and its associated provers. This is an important exercise for students learning the Event-B notation, who will then be able to put the question of refinement into practice. It is also a way to illustrate the use of proof checkers when ealing with proofs of program ,properties. Finally, we found the way to write the proof checking using Event-B and Rodin without toil. The rest of the paper is organised as follow.

Section 2 introduces Event-B notations required for expressing contracts; we give a short description of the small programming language used for illustrating concepts. Section 3 introduces basic notations and concepts for programming by contracts, the translation of a contract into an Event-B context and a Event-B machine. Section 4 is giving several examples of contract for classical algorithms with comments on the proof process. In section 5, we give the justification of the general translation process. Finally we conclude and give some perspectives. The full version of the paper with Rodin archives is available at the link[1].

2 Modelling and Programming Languages

2.1 Summary on Event-B

Event-B is a correct-by-construction, stated-based formal modelling language for system design [**?**]. 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* [**?**] 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 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

[1] `ldots`

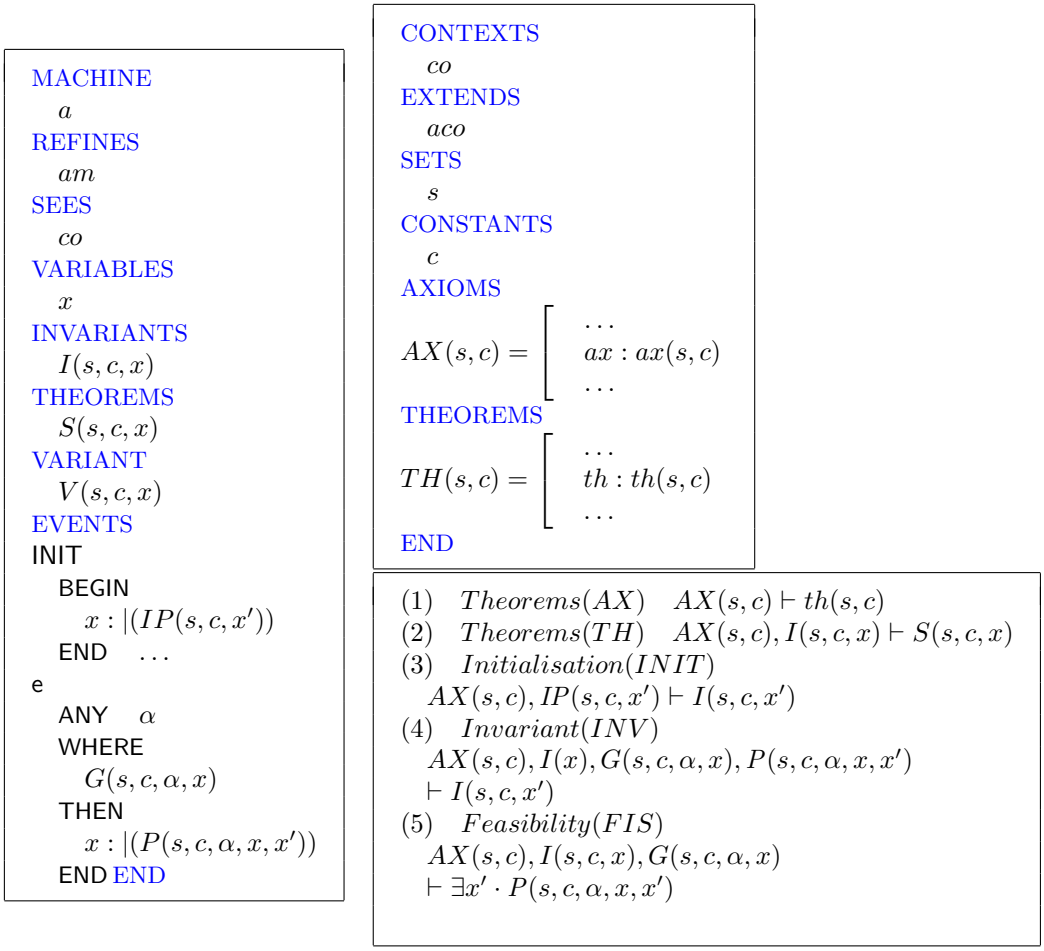


Fig. 2: Event-B structures: Context & Machine and Proof Obligations

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 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[2] 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's 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.

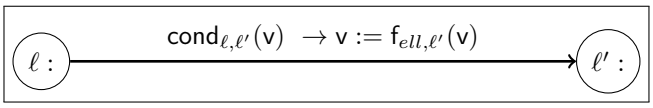
```
ℓ0 :
k := 0;
ℓ1 :
co := 0;
ℓ2 :
while (k < n) do
  ℓ3 :
    if (k %2 == 0)
  ℓ4 :
      co := co + k + 1;
    fi;
  ℓ5 :
    k := k + 1;
od;
  ℓ6 :
ro := co;
  ℓ5 :
```

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

- $\mathsf{v} := \mathsf{f}_{ell,\ell'}(\mathsf{v})$ is labelled as follow
  ℓ :
    $\mathsf{v} := \mathsf{f}_{\ell,\ell'}(\mathsf{v})$;
- 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:



[2] 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.

3 Programming by Contract

Programming by contract [**?**, **?**, **?**] 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 [**?**]) 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 [**?**] 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 [**?**, **?**] 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. We forget the $\tilde{}$ 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 31. *Contract in Event-B for square computation*

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

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

contract SQUARE
variables *n,r*
requires $n0 \in \mathbb{N} \wedge r0 \in \mathbb{Z}$
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 oroof process and it will be generally checked using *verification conditions* following the *Floyd-Hoare method [?, ?]*. The contract is codified and a code is added.

contrat 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 $v, v' \in$ D
  $\Big(\big(pre(v_0) \wedge P_\ell(v_0, v)) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \big) \Big)$,
  $\Rightarrow P_{\ell'}(v_0, v') \Big)$,

Three kinds of verification conditions should be checked and we will justify the method in a next section.

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

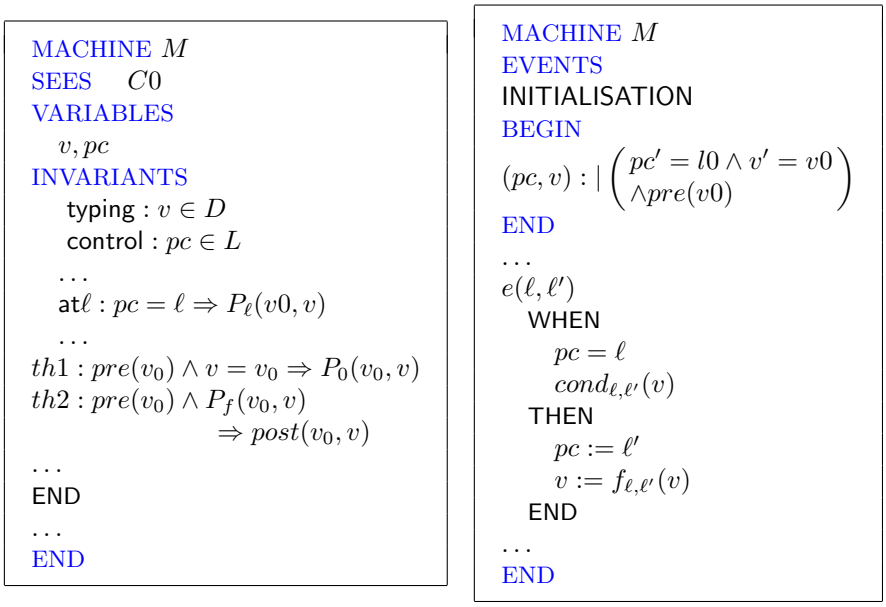


Fig. 3: Event-B machine for checking contract

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 4, we give examples which illustrate the methodology.

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 will justify verification conditions in Section 5 . 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.

contract SIMPLE
variables x
requires $x_0 \in \mathbb{N}$
ensures $x = 0$
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

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
$act1 : l := l1$

Obtaining the invariant simply involves copying the annotations as a conjunction of local annotations.

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

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 writing the Event-B machine.

5 Soundness and Completeness of the Methodology

The methodology is based on induction principles for proving invariance properties of programs and we base our justification on works of P. and R. Cousot [**?**,**?**]. We assume that we have a contract defined by a *requires* statement *pre(*x_0*)*, a *ensures* statement *post(*x_0, x_f*)* and a list of non-logical variables x. The algorithm is called P and a semantics for P is supposed to be defined by the transition relation $s \xrightarrow[P]{\star} s'$. A state s of P is generally defined as a mapping from variables to set of possible values D and the expression $s(\mathsf{x})$ is the values stored in the non-logical variable x and $s(\mathsf{x}) = x$ and, when s' is the next state of s, $s'(\mathsf{x}) = x'$. Hence, we choose to use logical variables to express our properties. When dealing with program semantics, we can use an operational semantics using two possible styles namely sos or nat and we define two transition relations over the set of configurations defined as either state s or control state (S, s) where S is a state-

ment or a program fragment. More precisely, we define the following relations $(S,s) \underset{nat}{\longrightarrow} s'$ or $(S,s) \underset{sos}{\longrightarrow} s'$ or $(S,s) \underset{sos}{\longrightarrow} (S',s')$. The validity of the contract is expressed by the following expression:

$$\forall x_0, x \in \mathsf{D}.pre(x_0) \wedge (P, x_0) \underset{nat}{\longrightarrow} x \Rightarrow post(x_0, x) \tag{1}$$

Verification conditions are based on the application of induction principles that we are sketching in the next lines. The partial correctness of a program P with respect to a precondition *pre* and a postcondition *post* is a state property called a safety property and the definition is expressed as follow. We assume that P is the annotated program, x is the list pf variables of P and pc is the control variable of P. The pair (pc,x) is denoted z and we consider that $init(z)$ defines the initial states of P and more precisely $init(z) \hat{=} z = (pc, x) \wedge pc = l0 \wedge pre(x0) \wedge x = x0$ and $z0 = (l0, x0))$ is denoting an initial state . An assertion $S(z0, z)$ expresses a relation between $z0$ and z and $z0$ intends to mean the initial value. L is the set of labels used for annotating the algorithm P.

Definition 1. *(Safety property)*

A property $S(z0, z)$ is a safety for an annotated program P, if $\forall z0, z \in \mathsf{L} \times \mathsf{D}.init(z0) \wedge (z0 \underset{P}{\overset{\star}{\rightarrow}} z) \Rightarrow S(z0, z)$.

Property 1. *(Induction Principle (I))*

A property $S(z0, z)$ is a safety for an annotated program P if, and only if, there exists a property $I(z0, z)$ satisfying:

1. $\forall z0, z \in \mathsf{L} \times \mathsf{D}.init(z0) \wedge z = z0 \Rightarrow I(z0, z)$
2. $\forall z0, z, z' \in \mathsf{L} \times \mathsf{D}.init(z0) \wedge I(z0, z) \wedge (z \underset{P}{\rightarrow} z') \Rightarrow I(z0, z')$
3. $\forall z0, z \in \mathsf{L} \times \mathsf{D}.init(z0) \wedge I(z0, z) \Rightarrow S(z0, z)$

The relation $z \underset{P}{\rightarrow} z'$ expresses a basic transition of P and corresponds to an event denoted $e(\ell, \ell')$. $BA(e(\ell, \ell'))(x, x')$ is the before-after predicate simulating the transition $z \underset{P}{\rightarrow} z'$ and it means that $BA(e(\ell, \ell'))(x, x') \hat{=} z \underset{P}{\rightarrow} z' \wedge z = (\ell, x) \wedge z' = (\ell', x')$. The property $I(z0, z)$ is called an (inductive) invariant and the problem is to find the invariant. In the annotation-based approach, the discovery of the invariant is related to the annotation produced by the user.

We are considering the following property $J(z0, z)$ defined from the annotated program P:

$$J(\ell 0, x0, \ell, x) \hat{=} \begin{cases} pre(x0) \\ pc \in L \wedge x \in \mathsf{D} \\ \ldots \\ pc = \ell \Rightarrow P_\ell(x0, x) \\ \ldots \end{cases}$$

Now, we have to state events as $e(\ell, \ell')$ and we can consider the cases according to the annotation and the syntax of P. The annotation of P is assigning a label before any statement S and a next label depending the enabled transition. We derive the property for defining events.

Property 2. *$e(\ell, \ell')$ is defined as follow:*

- $\ell : \mathsf{x} := e(\mathsf{x}); \ell' : \ldots$:
  $e(\ell, \ell') \hat{=} \mathsf{when}\ pc = \ell\ \mathsf{then}\ (pc, x) := (\ell', e(x))\ \mathsf{end}$ *when the program counter pc is equal to ℓ, then it is updated to ℓ' and x is set to the values of e*
- $\ell : \mathsf{if}\ b(x)\ \mathsf{then}\ \ell' : S1 \ldots$:
  $e(\ell, \ell') \hat{=} \mathsf{when}\ pc = \ell\ \wedge b(x)\ \mathsf{then}\ pc := \ell'\ \mathsf{end}$
- $\ell : \mathsf{if}\ b(x)\ \mathsf{then}\ \ldots\ \mathsf{else}\ \ell' : \ldots \mathsf{end}$:
  $e(\ell, \ell') \hat{=} \mathsf{when}\ pc = \ell\ \wedge \mathsf{not}\ b(x)\ \mathsf{then}\ pc := \ell'\ \mathsf{end}$
- $\ell : \mathsf{while}\ b(x)\ \mathsf{then}\ \ell' : S1 \ldots$:
  $e(\ell, \ell') \hat{=} \mathsf{when}\ pc = \ell\ \wedge b(x)\ \mathsf{then}\ pc := \ell')$
- $\ell : \mathsf{while}\ b(x)\ \mathsf{then} \ldots \mathsf{od}; \ell' : \ldots$:
  $e(\ell, \ell') \hat{=} \mathsf{when}\ pc = \ell\ \wedge \mathsf{not}\ b(x)\ \mathsf{then}\ pc := \ell')$
- $\ell' : \mathsf{while}\ b(x)\ \mathsf{then} \ldots \ell : \mathsf{x} := e(\mathsf{x}) \mathsf{od}; \ldots$:
  $e(\ell, \ell') \hat{=} \mathsf{when}\ pc = \ell\ \mathsf{then}\ (pc, x) := (\ell', e(x))$

The translation is based on the sos semantics for P. We have now an event-based expression of progam transition and we can state the next property which is showing that event preserve the invariant $J(\ell 0, x0, \ell, x)$. The set of events of P is called E(P).

We should now state the safety property by substituting z by (ℓ, x). A property $S(z0, z)$ is a safety for an annotated program P, if $\forall z0, z \in \mathsf{L} \times \mathsf{D}.init(z0) \wedge (z0 \xrightarrow[P]{\star} z) \Rightarrow S(z0, z)$. We can reformulate as follow. A property $S(\ell 0, x0, \ell, x)$ is a safety for an annotated program P, if $\forall \ell 0, \ell \in \mathsf{L}, x0, x \in \mathsf{D}.pre(x0) \wedge \ell 0 \in \mathsf{L0} \wedge (\ell 0, x0) \xrightarrow[P]{\star} (\ell, x)) \Rightarrow S(\ell 0, x0, \ell, x)$. L0 is the set of initial labels and we are assuming that there is only one element in L0. From now, $e(\ell, \ell')$ designates the event leading the control from ℓ to ℓ' and it may be possible that it does not exist when labels are not consecutive, when considering the property 2. $BA(e(\ell, \ell'),)$ is the relation between before values of pc and x and next values of pc and x.

The induction principle stated in the property 1 is reformulated as follow.

Property 3. *(Induction Principle (II))*
A property $S(\ell 0, x0, \ell, x)$ is a safety property for an annotated program P if, and only if, there exists a property $I(\ell 0, x0, \ell, x)$ satisfying:

1. $\forall \ell 0, \in \mathsf{L}, x0 \in \mathsf{D}.\ell 0 \in \mathsf{L0} \wedge pre(x0) \wedge x = x0 \wedge pc = \ell 0 \Rightarrow J(\ell 0, x0, \ell, x)$
2. $\forall \ell, \ell' \in \mathsf{L}, x, x0 \in \mathsf{D}.\ell 0 \in \mathsf{L0} \wedge pre(x0) \wedge J(\ell 0, x0, \ell, x) \wedge BA(e(\ell, \ell'),)(\ell, x, \ell', x') \Rightarrow J(\ell 0, x0, \ell', x')$
3. $\forall \ell 0, \ell \in \mathsf{L}, x0, x \in \mathsf{D}.pre(x0) \wedge \ell 0 \in \mathsf{L0} \wedge J(\ell 0, x0, \ell, x) \Rightarrow S(\ell 0, x0, \ell, x)$

The induction principle stated by property 3 is adapted from the induction principle in property labelprop:ip. The assumption on the set L0 can be translated and made simpler.

Property 4. *(Induction Principle (III))*
A property $S(x0, \ell, x)$ is a safety for an annotated program P with one entry point if, and only if, there exists a property $I(x0, \ell, x)$ satisfying:

1. $\forall x0 \in \mathsf{D}.pre(x0) \wedge x = x0 \wedge \ell = \ell 0 \Rightarrow J(x0, \ell, x)$
2. $\forall \ell, \ell' \in \mathsf{L}, x, x0 \in \mathsf{D}.pre(x0) \wedge J(x0, \ell, x) \wedge BA(e(\ell, \ell'),)(\ell, x, \ell', x') \Rightarrow J(x0, \ell', x')$
3. $\forall \ell \in \mathsf{L}, x0, x \in \mathsf{D}.pre(x0) \wedge J(x0, \ell, x) \Rightarrow S(x0, \ell, x)$

The statement of the induction principle (III) in the property 4 is expressed using the Event-B formalism.

Property 5. *(INITIALISATION) The two statements are equivalent*

- $\forall x0 \in \mathsf{D}.pre(x0) \wedge x = x0 \wedge \ell = \ell 0 \wedge \Rightarrow J(x0, \ell, x)$
- $pre(x0) \wedge x = x0 \wedge \Rightarrow P_{\ell 0}(x0, x)$ *(pre)*

The statement (pre) should be proved as a theorem in the main machine.

Property 6. *(GENERALISATION) The two statements are equivalent*

- $\forall \ell \in \mathsf{L}, x0, x \in \mathsf{D}.pre(x0) \wedge J(x0, \ell, x) \Rightarrow S(x0, \ell, x)$
- $\forall \ell \in \mathsf{L}.pre(x0) \wedge J(x0, \ell, x) \Rightarrow S(x0, \ell, x)$ *(gen)*

The statement (gen) should be proved as a theorem in the main machine.

The two properties 5 and help in proving any safety property. We use the induction principle for the following cases:

- Partial correctness (PC): we denote by LF the set of termination points of P; $PC(x0, \ell, x) \hat{=} \ell \in \mathsf{LF} \wedge pc = \ell \Rightarrow post(x0, x)$.
- RunTime Error (RTE): we define for each pair (ℓ, ℓ') a condition defined by $DEF(\ell, \ell')(x)$ when (ℓ, ℓ') defines an event denoted $(\ell, \ell')\,\mathrm{dom}(e)$; however from a label ℓ, there is one next label (assignment) or two next labels (condition). We will denote $next(\ell)$ the set of next labels for ℓ:
  $RTE(x0, \ell, x) \hat{=} \bigwedge_{\ell' \in next(\ell)} DEF(\ell, \ell')(x)$.

Now, we can derive a new property for showing how the induction step of our induction principle in property 4 is handled in the Event-B environment. In fact, the invariant $J(x0, \ell, x)$ is written as a conjonction of implications and the invariant $J(x0, \ell, x)$ is proved to be preserved by the events of $\mathsf{E(P)}$. Thanks to the inductive proof of Event-B proof obligations. It leads us to the final property.

Property 7. *(Spundness of the method) If the initialisation init, the generalisation gen and the step induction are proved to be correct by the Rodin platform, the property $S(x0, \ell, x)$ is a correct safety property for the program P. In particular, one can handle the partial correctness and the run time error safety properties.*

Now, we can notice that the transition relation ‘ is defined over set of states and the variable *pc* can also model a multiple control point. It means that we can also derive a proof of a safety property for a concurrent or distributed algorithm by defining the annotation of each process of the program and by defining related events modelling each transition relation for each process. We can also mention that it is also possible to use the Atelier-B platform and there is a simple syntactical translation of Event-B models between the two platforms.

6 Teaching Formal Modelling and Verification

The evolution of teaching in our courses on software engineering and distributed algorithms is marked by the use of a number of tools with master level students.

Context of our courses on verification, validation and modelling - 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 the principles of induction. 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.

Basic formal notations for dealing with programming concepts - Upon their arrival in the fourth year, we realized that there was a lack of connection between the problem posed by a particularly distressing 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. Demonstrating 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.

Lectures on distributed algorithms - In the fourth year, students are taught the main algorithms of distributed algorithms and we use two frameworks: Event-B and TLA. The aim is to show the students what abstractions these algorithms are based on. We make informal use of the notion of refinement, but we only illustrate it on the few cases of distributed algorithms we deal with. Among these abstractions, we can mention the queue for mutual exclusion [**?**] or the acyclic graph for leader election [**?**]. It is crucial to familiarise students with the notation used in both sequential and distributed programming. This is the same notation used by Leslie Lamport to describe his distributed algorithms and by Gerard Tel. PlusCal is used to write an algorithmic expression, and our training in translations into the Event-B and TLA$^+$ languages helps in the dissemination of concepts.

Lectures on software-modelling modelling The second stage of our project to integrate formal methods into a university curriculum is to teach the Event-B language and to use incremental development based on refinement. We will 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 two previous 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 [**?**]. was the starting point for this work. It made it possible to explain this algorithm simply to students. The attentive reader may wonder why TLA$^+$ (TLAPS) should be used rather than Event-B (Rodin). Our answer is clear and

definitive. The TLA$^+$ language is more expressive than the Event-B language. It allows the expression of both safety properties and liveness properties, as well as fairness hypotheses. The TLAPS environment is also an important element. It enables students to be educated in the proof style defined by Leslie Lamport. On the other hand, the eb language produces a set of verification conditions that guide the user through the exercise. The modelling language is based on the notion of set rather than the TLA$^+$ language, which emphasises total functions. We will integrate this verification condition decomposition strategy into TLA$^+$ and thus simplify the proof for the TLAPS tool. Our point is more nuanced and is based on two key elements. Firstly, the TLA$^+$ language is more expressive than the Event-B language, as it enables the expression of both safety properties and liveness properties, as well as fairness hypotheses. Secondly, the TLAPS environment is a crucial factor, as it facilitates the education of students in the proof style defined by Leslie Lamport. Conversely, the Event-B language produces a set of verification conditions called *proof obligations* that guide the user through the proof exercise. The modelling language Event-B is based on the notion of *set* rather than the TLA$^+$ language, which emphasises total functions. The objective is to integrate this verification condition decomposition strategy into TLAP$^+$ and thus simplify the proof for the TLAPS tool. We have developed a solution enabling the solutions of the Event-B models to be linked to the TLA+ modelling framework and its TLAPS development prover, which will be published soon. Our teaching is based on the principle that students must learn the process of interactive proof. This is a fundamental skill that cannot be acquired without a minimum of mathematical proof skills. Our student 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 [**?**] theorem, which also allows us to play with inductions and inductive definitions.

7 Conclusion and Future Works

This document presents a straightforward approach for developing a framework for verifying safety properties using the Event-B language and the Rodin environment. We have employed the work of P. and R. Cousot [**?**] to define an induction principle for proving safety properties in the case of sequential programs, which can be generalised to concurrent or distributed programs. This translation demonstrates how verification tools operate and illustrates the link between the semantics of programming languages and the verification process. The process is employed to illustrate concepts of formal methods and more practically formal engineering methodology. Atelier-B may be used as well for translating the contracts and has been used on some examples. We have also indicated the possible link with the TLA+ framework. A few examples illustrate the approach. Future works include the construction of an automatic translation tool in the Rodin platform and the development of more complex examples. The

relationship between the fully proven Event-B models and the TLA+ framework is a key objective. It has the potential to facilitate cross-certification of Rodin proofs. We provide a link with Event-B models as Rodin archives, as well as a comprehensive report.

A Archives of the Rodin projects for checking contracts

A.1 Example 1

CONTEXT *context*
SETS

CONSANTS
$x0, l0, l1, l2, l3,$
AXIOMS
$pre1(x0, y0) : x0 \in \mathbb{N}$
$axm2 : partition(L, \{l0\}, \{l1\}, \{l2\}, \{l3\})$

MACHINE $algorithm$ SEES $context$
VARIABLES
x l
INVARIANTS
$inv1 : x \in \mathbb{N}$
$inv2 : l \in L$
$inv3 : l = l0 \Rightarrow 0 \leq x \wedge x \leq x0 \wedge x0 \in \mathbb{N}$
$inv4 : l = l1 \Rightarrow 0 < x \wedge x \leq x0 \wedge x0 \in \mathbb{N}$
$inv5 : l = l2 \Rightarrow x = 0$
$pre : x = x0 \wedge x0 \in \mathbb{N} \Rightarrow 0 \leq x \wedge x \leq x0 \wedge x0 \in \mathbb{N}$
$post : x0 \in \mathbb{N} \wedge x = 0 \Rightarrow x = 0$
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
$act1 : l := l0$
$act2 : x := x - 1$

end

A.2 Example 2

CONTEXT $context0$
SETS
CONSANTS

$x0, y0, z0, l0, l1, l2, l3, l4, l5, l6, l7, l8, l9$,AXIOMS
$axm1 : x0 \in \mathbb{N}$
$axm2 : y0 \in \mathbb{N}$
$axm5 : z0 \in \mathbb{Z}$
$axm3 : partition(C, \{l0\}, \{l1\}, \{l2\}, \{l3\}, \{l4\}, \{l5\}, \{l6\}, \{l7\}, \{l8\}, \{l9\})$
$axm7 : x0 = 12$
$axm6 : y0 = 34$
end

MACHINE $algorithm$ SEES $context0$
VARIABLES
$pc\ x\ y\ z$
INVARIANTS
$inv1 : pc \in C$
$inv2 : x \in \mathbb{N}$
$inv3 : y \in \mathbb{N}$
$inv4 :$
$z \in \mathbb{Z}$
$inv5 : x = x0 \land y = y0$
$inv6 : pc = l0 \Rightarrow x = x0 \land y = y0 \land z = z0 \land x0 \in \mathbb{N} \land y0 \in \mathbb{N} \land z0 \in \mathbb{Z}$
$inv7 : pc = l1 \Rightarrow x = x0 \land y = y0 \land z = z0 \land x < y$
$inv9 : pc = l2 \Rightarrow x = x0 \land y = y0 \land z = z0 \land x \geq y$
$inv11 : pc = l3 \Rightarrow x = x0 \land y = y0 \land z \in \{x0, y0\} \land x \leq z \land y \leq z$
$pre : x0 \in \mathbb{N} \land y0 \in \mathbb{N} \land z0 \in \mathbb{Z} \land x = x0 \land y = y0 \land z = z0 \Rightarrow x = x0 \land y = y0 \land z = z0 \land x0 \in \mathbb{N} \land y0 \in \mathbb{N} \land z0$
$post : pc = l3 \land x = x0 \land y = y0 \land z \in \{x0, y0\} \land x \leq z \land y \leq z \Rightarrow z \in \{x0, y0\} \land x0 \leq z \land y0 \leq z$
THEN
$act5 : pc := l0$
$act6 : x := x0$
$act7 : y := y0$
$act8 : z := z0$

Event $al0l1$
WHEN
$grd1 : pc = l0$
$grd2 : x < y$

THEN
$act4 : pc := l1$

Event $al0l2$
WHEN
$grd1 : pc = l0$
$grd2 : x \geq y$

THEN
$act1 : pc := l2$

Event $al1l3$
WHEN
$grd1 : pc = l2$

THEN
$act1 : pc := l3$
$act2 : z := y$

Event $al2l3$
WHEN
$grd1 : pc = l2$

THEN
$act1 : pc := l3$
$act2 : z := x$

end

A.3 Example 3

CONTEXT $context0$
SETS
CONSANTS

$f, n, l0, l1, l2, l3, l4, l5, l6, l7, l8, l9$,AXIOMS
$axm1 : n \in \mathbb{N}_1$
$axm2 : f \in 0 \mathinner{..} n-1 \rightarrow \mathbb{N}$
$axm3 : partition(C, \{l0\}, \{l1\}, \{l2\}, \{l3\}, \{l4\}, \{l5\}, \{l6\}, \{l7\}, \{l8\}, \{l9\})$
$axm4 : \forall P \cdot P \subseteq \mathbb{N} \land finite(P) \Rightarrow (\exists am \cdot am \in P \land (\forall k \cdot k \in P \Rightarrow k \leq am))$
end

MACHINE $algorithm$ SEES $context0$
VARIABLES
l m i
INVARIANTS
$inv1 : l \in C$
$inv2 : m \in \mathbb{N}$
$inv3 : i \in \mathbb{N}$
$inv4 : i \in 0 \mathinner{..} n$
$inv5 : l = l0 \Rightarrow m \in \mathbb{N} \land i \in \mathbb{N}$
$inv6 : l = l1 \Rightarrow m = f(0)$
$inv7 : l = l2 \Rightarrow i \leq n \land 0 \mathinner{..} i-1 \subseteq dom(f) \land (\forall j \cdot j \in 0 \mathinner{..} i-1 \Rightarrow f(j) \leq m) \land m \in ran(f)$
$inv8 : l = l3 \Rightarrow i < n \land 0 \mathinner{..} i \subseteq dom(f) \land (\forall j \cdot j \in 0 \mathinner{..} i-1 \Rightarrow f(j) \leq m) \land m \in ran(f)$
$inv9 : l = l4$
$\Rightarrow i < n \land 0 \mathinner{..} i \subseteq dom(f) \land (\forall j \cdot j \in 0 \mathinner{..} i-1 \Rightarrow f(j) \leq m) \land f(i) > m \land m \in ran(f)$
$inv11 : l = l6 \Rightarrow i < n \land 0 \mathinner{..} i \subseteq dom(f) \land (\forall j \cdot j \in 0 \mathinner{..} i \Rightarrow f(j) \leq m) \land m \in ran(f)$
$inv13 : l = l8$
$\Rightarrow i = n \land dom(f) \subseteq 0 \mathinner{..} i-1 \land (\forall j \cdot j \in 0 \mathinner{..} i-1 \Rightarrow f(j) \leq m) \land m \in ran(f)$
$post : l = l8$
$\Rightarrow (\forall j \cdot j \in 0 \mathinner{..} n-1 \Rightarrow f(j) \leq m) \land m \in ran(f)$
$pre : f \in 0 \mathinner{..} n-1 \to \mathbb{N} \land i \in 0 \mathinner{..} n \land m \in \mathbb{N} \Rightarrow m \in \mathbb{N} \land i \in \mathbb{N}$

THEN
$act5 : l := l0$
$act6 : m :\in \mathbb{N}$
$act7 : i :\in 0 \mathinner{..} n$

Event $al0l1$
WHEN
$grd1 : l = l0$

THEN
$act4 : l := l1$
$act5 : m :| (m' = f(0))$

Event $al1l2$
WHEN
$grd1 : l = l1$

THEN
$act1 : l := l2$
$act2 : i := 1$

Event $al2l3$
WHEN
$grd1 : l = l2$
$grd2 : i < n$

THEN
$act1 : l := l3$

Event $al2l8$
WHEN
$grd1 : l = l2$
$grd2 : i \geq n$

THEN
$act1 : l := l8$

Event $am3l4$
WHEN
$grd1 : l = l3$
$grd2 : f(i) > m$

THEN
$act1 : l := l4$

Event $el3l6$
WHEN
$grd1 : l = l3$
$grd2 : f(i) \leq m$

THEN
$act1 : l := l6$

Event $al4l6$
WHEN
$grd1 : l = l4$

THEN
$act1 : l := l6$
$act2 : m := f(i)$

Event $al6l2$
WHEN
$grd1 : l = l6$

THEN
$act1 : l := l2$
$act2 : i := i + 1$

Event $el3l8$
WHEN
$grd1 : l = l3$
$grd2 : i \geq n$

THEN
$act1 : l := l8$