Checking contracts in EventB

Dominique Méry

July 15, 2024

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 correctness and the absence of run-time 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

Contents

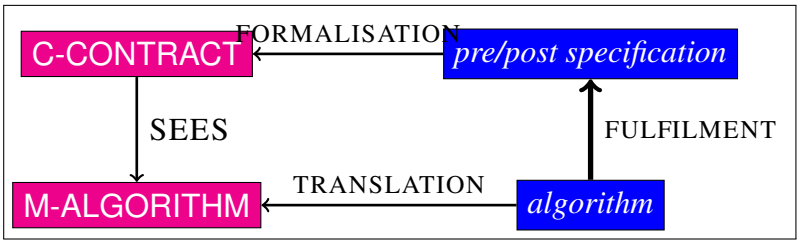


Figure 1: The verification pattern

1 Introduction

Programming by contract, as outlined in Meyer's work [23], 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 [2] 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 [13].This exercise also introduces the Event-B language and the use of the Rodin [3] and Atelier-B [1,5,9] 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 such as Dafny [15], Why3 [24] and Frama-c [6]. 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 pet programming language.

Our main reference is the work of Patrick and Radhia Cousot [12] 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.

In [7], 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 dealing with proofs of program properties. Finally, we found the way to write the proof checking using Event-B and Rodin without toil.

Before we present the technical and formal elements, we must set out the framework for this experiment. Its aim is to introduce formal methods into the academic curriculum and train students in their use. The rest of the paper is organised as follow.

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

2 Academic Context

Our experimentation is long and began during the academic year 1983/1984 at the University of Metz. As assistant professor, I taught tutorials on the Floyd-Hoare method, including the treatment of sequential programming language concepts like pointers, functions, procedures, etc. As professor, Patrick Cousot was my colleague and taught lectures on the same topics. Then our paths diverged. From 1983 to 1993, I taught the same course as Patrick Cousot, who joined the Ecole Polytechnique in 1984. I led a project based on the publications of Clarke, Emerson and Sistla [8]. In this project, students implemented the method from this publication. One of the projects carried out in Pascal on a Micral PC running DOS was a resounding success. It was possible to verify temporal logic properties on finite transition systems. However, I remained fascinated by the general case and finite models seemed limited.

At the time, we had created a verification tool for SDL programs in the CONCERTO environment using the young ISABELLE prover [22]. This experience convinced me of the usefulness of the tools and the feasibility of such an approach. My appointment as a professor at the University of Nancy 1 in 1993 placed me once again in a new context: that of a computer science engineering school. I was given the course called Models and Algorithms (MALG), which covered computability, complexity, fixed-point theory (Kleene), propositional calculus, first-order calculus, resolution, verification of algorithms and functional programming. It had an hourly volume of 48 h 00 for lectures and 48 h 00 for tutorials. From 1988 to 1996, formal modelling languages were the focus of my research activities and teaching on formal methods. The CROCOS experiment [22] proposed a model of systems as a set of conditions/actions and a linear temporal logic.

My teaching history has led me to make some important observations that will inform the development of my teaching methods for formal methods: (1) It is clear that the MALG teaching programme was not sufficiently based on tools and projects; (2) The concepts linked to the semantics and logic of programs must be put in direct relation with the uses of programming features; (3) A student of computer science must manage models of different types; and (4) modelling remains an activity based on practice and the use of tools.

The MALG course is the first formal introduction to computer engineering training. It is taken in the second year of the course. Students have already received basic training in computer science in the first year, including mastery of a programming language, computer architecture and databases. The second year builds on this and leads to a third year during which a formal modelling course using the Event-B language is given. MALG is therefore a preparation for this 24-hour formal modelling course. Lastly, the school's students are recruited after a highly selective competitive examination, following two years of preparation in preparatory classes for entry to engineering schools. They are awarded a Master's degree in computer science at the end of their three years at the school. The number of students enrolled on the MALG course is

[1] https://mery54.github.io/fmt/

between 50 and 70; the number of third-year students on the formal modelling course is between 25 and 30.

The students have a good level of science but have chosen computer science to develop their careers. By learning the basics of computer science, they are made aware of the issues surrounding bugs in programs. However, a large number of comments indicate that they are often unconcerned about run-time errors, the storage of information, and issues related to addresses in programs. It is clear that modern programming languages have been able to protect users from certain pitfalls associated with function or procedure calls and parameter passing mechanisms. Amongst the students, one group is aiming to go further into the design of embedded systems, using low-level computing techniques; my colleagues leave me free to choose the languages and tools, but it is important to develop skills in model checking. Among the languages, there is the C language used by Frama-c in an extended but ACSL-compatible form, as well as LUSTRE, which was chosen to introduce the synchronous hypothesis, with tools from the VERIMAG platform [18], as well as using the KIND2 tool [25].

In the first lesson we use examples of C programs to justify what we are going to introduce. The C program of listing 1, for example, returns an unexpected value for almost all the students. Only one or two students explain and predict the result. We then look at tests, which are still not very well mastered; we then give an example of a non-testable program in the listing 2, and show how Frama-c reacts by suggesting a condition. We give the sequence of programs that we have developed from these two examples in the appendix to the complete version (see `https://mery54.github.io/fmt/`) of this document.

Listing 1: Function average

```
#include <stdio.h>
#include <limits.h>
int average(int a,int b)
{
  return((a+b)/2);
}

int main()
{
  int x,y;
  x=INT_MAX;y=INT_MAX;
  printf("Average for %d
    and %d is %d\n",x,y,
          average(x,y));
  return 0;
}
```

Listing 2: Non testable Function

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int x, y;
    // Seed the random number
    // generator
    // with the current time
    srand(time(NULL));
    // Generate a random number
    // between 1 and 100
    x = rand() % 100 + 1;
    // Perform some calculations
    y = x / (100 - x);
    printf("Result: %d\n", y);
    return 0;
}
```

In our paper, we will focus on the technical concepts that we have used to model programs, but also on program properties and on the available software tools that implement analysis, simulation and proof techniques. We will also comment on the tools used, which are not limited to Rodin but are used jointly and in a complementary way.

To conclude this presentation of the academic context, we set out our pedagogical objectives in relation to the targeted training course, which concerns the design of embedded or non-embedded software systems with modelling, verification and validation. The challenge is to teach the mastery of software modelling and its properties and the use of verification techniques, in particular mathematical proof.

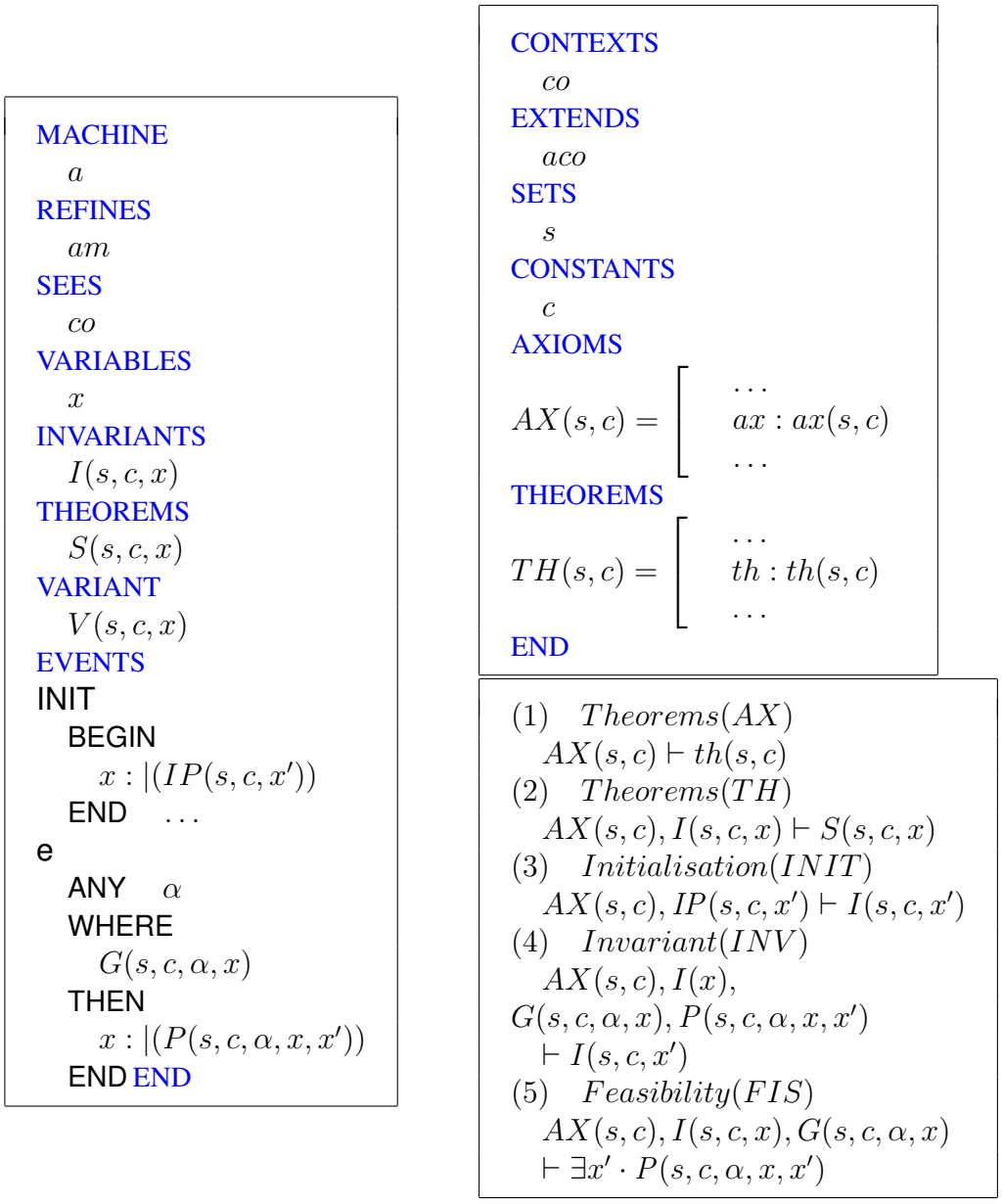


Figure 2: Event-B structures: Context & Machine and Proof Obligations

3 Modelling and Programming Languages

3.1 Summary on Event-B

Event-B is a correct-by-construction, stated-based formal modelling language for system design [2]. 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* [20] 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 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[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 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.

Comments and observations 1. *The main difficulty in modelling with Event-B is that the modelling language is very abstract and the notation* $x :| R(s, c, x, x')$ *is very general. Set theory allows a certain amount of freedom in modelling, in particular to express relationships between values that are not always computable. For example, an Event-B variable corresponds to a state of an algorithm or an observed program, but this variable is by nature perdurant or flexible; it has a current value, an initial value and sometimes a final value. The term "variable" is in fact very overloaded and a distinction must be made between these perdurable vqriables and the variables of logical formalims. We separate the initial values from the variables by deciding to use an index 0 or f, where x0 is the initial value of x and xf is the final value of x. The ACSL language can be used to designate variable values at given points, and it is important to unpack or dissect these concepts before using them effectively. The introduction of Event-B was in response to a comment made by students who were first introduced to TLA+ using the ToolBox (or VSCode) platform and who had to write lists of definitions that were tested using the platform's model checker. The PlusCal language has made the use of TLA concepts more transparent, but our objective is still to learn a language in two stages: firstly, to check or describe algorithms, and then to use it as a modelling language, but with model correctness through refinement. The aim is to train students to manipulate formal concepts using IT tools.*

3.2 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.

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

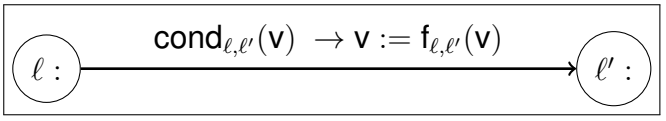
ℓ_0 :
$k := 0$;
ℓ_1 :
$co := 0$;
ℓ_2 :
while $(k < n)$ do
  ℓ_3 :
    if $(k \,\%2 \;==\; 0)$
  ℓ_4 :
      $co := co + k + 1$;
    fi;
  ℓ_5 :
    $k := k + 1$;
od;
  ℓ_6 :
$ro := co$;
  ℓ_5 :

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

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



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

Comments and observations 2. *Initially we used examples borrowed from Manna [21], which uses flowcharts and a structured language to describe algorithms. This book is very comprehensive and gives an overview of important topics in theoretical computer science, and is still useful for students. The course consists of translating some flowcharts into TLA notation by hand, then doing the same with algorithms in structured form, and finally experimenting with PlusCal. The course therefore starts by gradually learning how to translate a model and then to check the model obtained for correctness properties such as partial correctness or absence of errors at runtime. During this phase, students learn the relationship between non-primed and primed variables, as well as set notation. Students first discover TLA/TLA+ through the ToolBox tool [17] , which allows them to write state transformations in a modelling language that includes set theory. The students quickly understood the limitations of the model checking tool and the advantages of parametrisation by a constant. It's a question of convincing with effective tools, and it's clear that the Rodin tool hides the induction stages with a list of proof obligations, which we'll come back to. Our idea is also to train students in the design of a method for proving program properties for their programming language.*

It could be argued that the TLA+ ToolBox platform [17] provides a powerful and effective proof tool based on Isabelle/HOL and SMT solvers, but the problem is that its use requires a greater effort on the part of the student, who has to manage the generation of verification conditions himself. In fact, the Event-B language describes a model which must be inductive and which describes the observations of changes of state of the variables in the model.

4 Programming by Contract

Programming by contract [23] 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 [19]) 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 [10] 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: $v_0 \xrightarrow{\text{P}} v_f$
- v_0 satisfies pre: $\mathsf{pre}(v_0)$ and v_f satisfies a relation post : $\mathsf{post}(v_0, v_f)$
- $\mathsf{pre}(v_0) \wedge v_0 \xrightarrow{\text{P}} v_f \Rightarrow \mathsf{post}(v_0, v_f)$

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 [11, 12] 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.

Comments and observations 3. *We highlight the elements that characterise a contract and emphasise the importance of the initial and final values of flexible variables. Students will be introduced to the question of* ***what*** *to compute and what not to compute. The post-condition is a relationship between initial and final values. A distinction is made between logical and non-logical variables. Finally, it is important to justify the link between this contract and the algorithm in question (the* ***how****). This stage justifies the principles of induction and highlights the theory of fixed-points on complete lattices. We emphasise the notion of computability associated with the algorithm. A set of verification conditions is derived from this induction and*

used to verify the annotations. At this point we talk about the connection with the strongest invariant and we obtain the Floyd-Hoare verification conditions.

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 4.1. *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 proof process and it will be generally checked using *verification conditions* following the *Floyd-Hoare method [14, 16]*. 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 $v, v' \in$ D
  $$\left(\begin{array}{l} \left(\begin{array}{l} pre(v_0) \wedge P_\ell(v_0, v)) \\ \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \end{array} \right) \\ \Rightarrow P_{\ell'}(v_0, v') \end{array} \right),$$

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.

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,

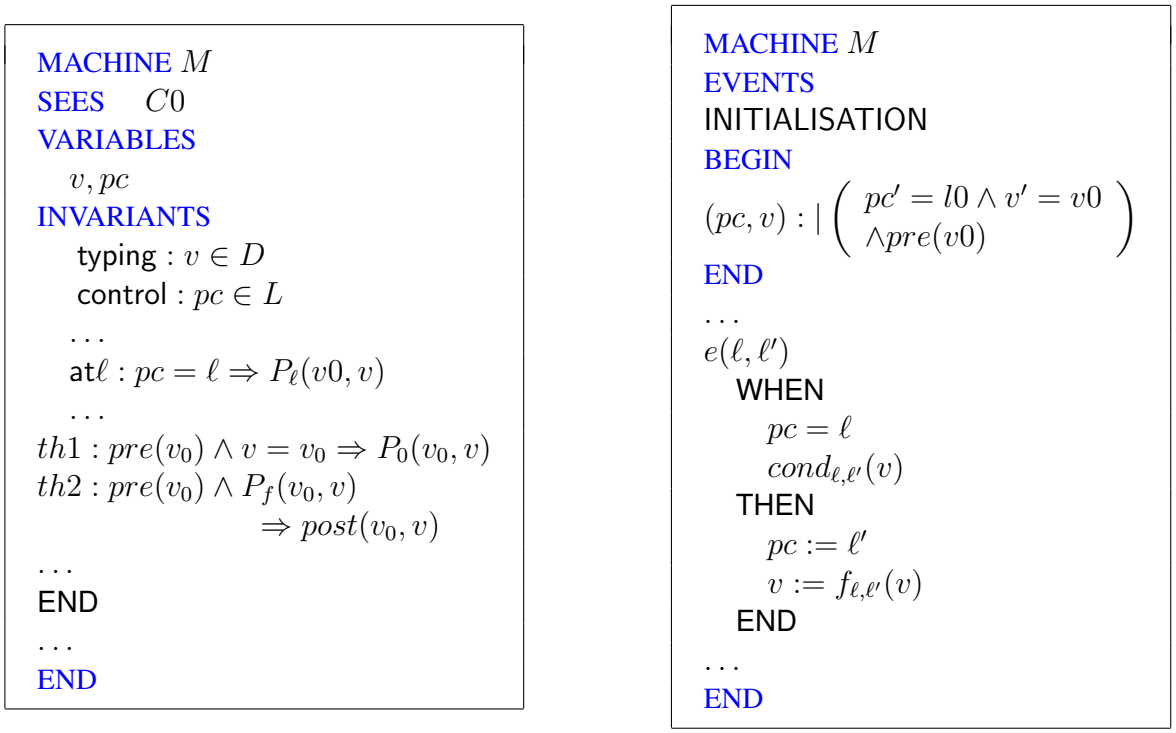


Figure 3: Event-B machine for checking contract

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 5, we give examples which illustrate the methodology.

Comments and observations 4. *We translated the verification conditions derived from our strongest invariant given by the fixed-point definition. Then we derived a set of annotations for verifying the algorithm against the contract, but this is just a translation of what we've been doing for many years and it came across as very natural and immediate during a tutorial. The idea was to try this translation and this communication translates this episode. But from a contract point of view, we can use the ACSL language, which can be used to express all the above concepts and which is equipped with a Frama-c verification tool with the wp plugin. We could stop there, but the story continues. In fact, the wp plugin uses the wp calculation to generate the verification conditions. This leads us to introduce the students to the wp calculation in its two forms, depending on whether we are considering total or partial correctness. It turns out that the wp plugin analyses the contract according to the Hoare logic system and we must therefore show the equivalence of the two approaches Floyd-Hoare and wp. In the short version, we omit the proof of correctness of our method but the students' understanding depends on the use of the wp calculus by hand. The work of Patrick and Radhia Cousot [12] is clear and we refer the reader to Patrick Cousot's book [10] which explains the relationship between these two ways of checking an algorithm against its contract. We have simply taken up the concepts and disseminated them over the past few years.*

5 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

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.

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

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

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.

6 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 [11, 12]. 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 statement 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 property for an annotated program* P*, if*
$\forall z0, z \in \mathsf{L} \times \mathsf{D}.init(z0) \wedge (z0 \xrightarrow[P]{\star} z) \Rightarrow S(z0, z)$.

A safety property is a state property true for any reachable states (z) from initial states ($z0$).

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. $\forall z0, z, z' \in \mathsf{L} \times \mathsf{D}.init(z0) \wedge I(z0, z) \wedge (z \xrightarrow[P]{} z') \Rightarrow I(z0, z')$
3. $\forall z0, z \in \mathsf{L} \times \mathsf{D}.init(z0) \wedge I(z0, z) \Rightarrow S(z0, z)$

The relation $z \xrightarrow[P]{} z'$ 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 $z \xrightarrow[P]{} z'$ and it means that $BA(e(\ell, \ell'))(x, x') \hat{=} z \xrightarrow[P]{} z' \wedge z = (\ell, x) \wedge z' = (\ell', x')$. 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:

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

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:*

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

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 $\forall \ell 0, \ell \in \mathsf{L}, x0, x \in \mathsf{D}.pre(x0) \wedge \ell 0 \in \mathsf{L0} \wedge (\ell 0, x0) \xrightarrow[P]{\star} (\ell, x)) \Rightarrow S(\ell 0, x0, \ell, x)$. $\mathsf{L0}$ is the set of initial labels and we are assuming that there is only one element in $\mathsf{L0}$. 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 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.

7 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.

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 [4]. 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 [10] theorem, which also allows us to play with inductions and inductive definitions. We have employed the work of P. and R. Cousot [12] 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.

References

[1] Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 1996.

[2] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[3] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[4] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of IEEE 1394 tree identify protocol. *Formal Aspects Comput.*, 14(3):215–227, 2003.

[5] H.-Ruyz Barradas. Event-B: Syntax and proof oglogations in Atelier-B. Technical report, ClearSy, 2020.

[6] 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.

[7] Néstor Cataño, K. Rustan M. Leino, and Víctor Rivera. The eventb2dafny rodin plug-in. In Diego Garbervetsky and Sunghun Kim, editors, *Proceedings of the Second International Workshop on Developing Tools as Plug-Ins, TOPI 2012, Zurich, Switzerland, June 3, 2012*, pages 49–54. IEEE Computer Society, 2012.

[8] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. 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 117–126. ACM Press, 1983.

[9] ClearSy. *B Language reference manual ver.1.8.10*, 2022.

[10] Patrick Cousot. *Principles of Abstract Interpretation*. The MIT Press, 2021.

[11] Patrick Cousot. Calculational design of [in]correctness transformational program logics by abstract interpretation. *Proc. ACM Program. Lang.*, 8(POPL):175–208, 2024.

[12] 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.

[13] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In *10th International conference on software engineering and formal methods*, pages 233–247. Springer, 2012.

[14] 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.

[15] Richard L. Ford and K. Rustan M. Leino. *Dafny Reference Manual*, 2017.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[17] Markus Alexander Kuppe, Leslie Lamport, and Daniel Ricketts. The TLA+ toolbox. In Rosemary Monahan, Virgile Prevosto, and José Proença, editors, *Proceedings Fifth Workshop on Formal Integrated Development Environment, F-IDE@FM 2019, Porto, Portugal, 7th October 2019*, volume 310 of *EPTCS*, pages 50–62, 2019.

[18] Verimag Laboratory. The synchrone reactive toolbox. https://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/reactive-toolbox/, 2022.

[19] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[20] Issam Maamria and Asieh Salehi Fathabadi. *Theory Plug-in User Manual*. University of Southampton.

[21] Zohar Manna. *Mathematical Theory of Computation*. Dover Publications, Inc., US, 2003.

[22] 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.

[23] Bertrand Meyer. Design by contract. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 1–50, 1991.

[24] Why3 Team. Why3. https://why3.lri.fr.

[25] Iowa University. Kind2 multi-engine smt-based automatic model checker for synchronous reactive systems. https://kind2-mc.github.io/kind2/, 2024.

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
$inv1 : pc \in C$
$inv2 : x \in \mathbb{N}$
$inv3 : y \in \mathbb{N}$
$inv4 :$
$z \in \mathbb{Z}$
$inv5 : x = x0 \land y = y0$
$inv6 : pc = l0 \Rightarrow x = x0 \land y = y0 \land z = z0 \land x0 \in \mathbb{N} \land y0 \in \mathbb{N} \land z0 \in \mathbb{Z}$
$inv7 : pc = l1 \Rightarrow x = x0 \land y = y0 \land z = z0 \land x < y$
$inv9 : pc = l2 \Rightarrow x = x0 \land y = y0 \land z = z0 \land x \geq y$
$inv11 : pc = l3 \Rightarrow x = x0 \land y = y0 \land z \in \{x0, y0\} \land x \leq z \land y \leq z$
$pre : x0 \in \mathbb{N} \land y0 \in \mathbb{N} \land z0 \in \mathbb{Z} \land x = x0 \land y = y0 \land z = z0 \Rightarrow x = x0 \land y = y0 \land z = z0 \land x0 \in \mathbb{N} \land$
$post : pc = l3 \land x = x0 \land y = y0 \land z \in \{x0, y0\} \land x \leq z \land y \leq z \Rightarrow z \in \{x0, y0\} \land x0 \leq z \land y0 \leq z$
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
$inv1 : l \in C$
$inv2 : m \in \mathbb{N}$
$inv3 : i \in \mathbb{N}$
$inv4 : i \in 0 \mathinner{..} n$
$inv5 : l = l0 \Rightarrow m \in \mathbb{N} \land i \in \mathbb{N}$
$inv6 : l = l1 \Rightarrow m = f(0)$
$inv7 : l = l2 \Rightarrow i \leq n \land 0 \mathinner{..} i-1 \subseteq dom(f) \land (\forall j \cdot j \in 0 \mathinner{..} i-1 \Rightarrow f(j) \leq m) \land m \in ran(f)$
$inv8 : l = l3 \Rightarrow i < n \land 0 \mathinner{..} i \subseteq dom(f) \land (\forall j \cdot j \in 0 \mathinner{..} i-1 \Rightarrow f(j) \leq m) \land m \in ran(f)$
$inv9 : l = l4$
$\Rightarrow i < n \land 0 \mathinner{..} i \subseteq dom(f) \land (\forall j \cdot j \in 0 \mathinner{..} i-1 \Rightarrow f(j) \leq m) \land f(i) > m \land m \in ran(f)$
$inv11 : l = l6 \Rightarrow i < n \land 0 \mathinner{..} i \subseteq dom(f) \land (\forall j \cdot j \in 0 \mathinner{..} i \Rightarrow f(j) \leq m) \land m \in ran(f)$
$inv13 : l = l8$
$\Rightarrow i = n \land dom(f) \subseteq 0 \mathinner{..} i-1 \land (\forall j \cdot j \in 0 \mathinner{..} i-1 \Rightarrow f(j) \leq m) \land m \in ran(f)$
$post : l = l8$
$\Rightarrow (\forall j \cdot j \in 0 \mathinner{..} n-1 \Rightarrow f(j) \leq m) \land m \in ran(f)$
$pre : f \in 0 \mathinner{..} n-1 \rightarrow \mathbb{N} \land i \in 0 \mathinner{..} n \land m \in \mathbb{N} \Rightarrow m \in \mathbb{N} \land i \in \mathbb{N}$

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
$act1 : l := l8$

B Sequence of C programs

B.1 Sequence 1

Listing 3: Function average

```
#include <stdio.h>
#include <limits.h>
int average(int a,int b)
```

```
{
  return((a+b)/2);
}

int main()
{
  int x,y;
  x=INT_MAX;y=INT_MAX;
  printf("Average␣␣for␣%d
␣␣␣␣and␣%d␣is␣%d\n",x,y,
         average(x,y));
  return 0;
}
```

Listing 4: Function average

```
#include <stdio.h>
#include <limits.h>
/*@  requires  0 <= a;
     requires a <= INT_MAX ;
     requires  0 <= b;
     requires b <= INT_MAX ;
     requires  0 <= a+b;
     requires a+b <= INT_MAX ;
     ensures \result <= INT_MAX;
*/
int average(int a,int b)
{
  return((a+b)/2);
}

int main()
{
  int x,y;
  x=INT_MAX / 2;y=INT_MAX / 2;
  //  printf("Average  for %d and %d is %d\n",x,y,
  //      );
  return average(x,y);
}
```

Listing 5: Function average

```
#include <stdio.h>
#include <limits.h>
/*@  requires  0 <= a;
     requires a <= INT_MAX ;
     requires  0 <= b;
     requires b <= INT_MAX ;
     requires  0 <= a+b;
     requires a+b <= INT_MAX ;
     ensures \result <= INT_MAX;
*/
int average(int a,int b)
{
  return((a+b)/2);
}

int main()
{
  int x,y,r;
  x=56;y=46;
  r=average(x,y);
  printf("Average␣␣for␣%d␣and␣%d␣is␣%d\n",x,y,r);
  return 0;
}
```

B.2 Sequence 2

Listing 6: Non testable Function

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int x, y;
    // Seed the random number
    // generator
    // with the current time
    srand(time(NULL));
    // Generate a random number
    // between 1 and 100
    x = rand() % 100 + 1;
    // Perform some calculations
    y = x / (100 - x);
    printf("Result:␣%d\n", y);
    return 0;
}
```

Listing 7: Non testable Function

```
// Heisenbug
#include <stdio.h>
#include <stdlib.h>
#include <time.h>



int main() {
  int x, y,i=0;

    for (i = 0; i <= 100000; i++) {
    // Seed the random number generator with the current time
    srand(time(NULL));

    // Generate a random number between 1 and 100
    x = rand() % 100 + 1;
        printf("Result: x=  %d\n",x);
    // Perform some calculations
    y = x / (100 - x);

    printf("Result: i=%d  %d\n",i, y);
    }

    return 0;
}
```

Listing 8: Non testable Function

```
// Heisenbug
#include <stdio.h>
#include <stdlib.h>
#include <time.h>



int main() {
  int x, y,i=0;

    for (i = 0; i <= 100; i++) {
    // Seed the random number generator with the current time
    srand(time(NULL)+i);

    // Generate a random number between 1 and 100
    x = rand() % 100 + 1;
        printf("Result: x=  %d\n",x);
    // Perform some calculations
    y = x / (100 - x);

    printf("Result: i=%d  %d\n",i, y);
    }

    return 0;
}
```