

**Modelling Software-based Systems:  
Correct by Construction Paradigm  
Design of Correct by Construction Sequential Algorithms**

Dominique MÉRY

October 20, 2024



# Contents

<b>Design of Correct by Construction Sequential Algorithms</b> . . . . .	<b>5</b>
Design of Correct by Construction Sequential Algorithms	
0.1. Introduction . . . . .	5
0.2. Design of a Iterative Sequential Algorithm . . . . .	7
0.2.1. Problem 1 : Calculating the sum of a vevtor v of integer values . . . . .	7
0.2.1.1. Specification of the problem to solve . . . . .	8
0.2.1.2. Refining to compute inductively . . . . .	8
0.2.1.3. Focus on the value to be preserved . . . . .	9
0.2.1.4. Obtaining an algorithmic machine . . . . .	10
0.2.1.5. Comments on the methodology . . . . .	11
0.2.2. Transformations machines Event-B into sequential algorithms . . . . .	11
0.3. Examples of development . . . . .	13
0.3.1. Problem 2: Computing the function power 3 $\lambda x.x^3$ using only additions . . . . .	13
0.3.2. Problem 3: Searching a value in an array . . . . .	15
0.3.3. Problem 4: Computing a primitive recursive function . . . . .	16
0.4. Design of a Recursive Sequential Algorithm . . . . .	17
0.4.1. The “Call as Event” Idea . . . . .	17
0.4.2. Applying the call as event technique . . . . .	19
0.4.2.1. Problem 1: Computing the power 2 ( $\lambda x.x^2$ ) . . . . .	19
0.4.2.2. Problem 2: Binary search in an array . . . . .	21
0.4.3. Comments on the call as event idea . . . . .	24
0.5. Final comments . . . . .	24
0.6. Bibliography . . . . .	25



# Design of Correct by Construction Sequential Algorithms

Dominique MÉRY<sup>1</sup>

<sup>1</sup> LORIA, Telecom Nancy & Université de Lorraine

## 0.1. Introduction

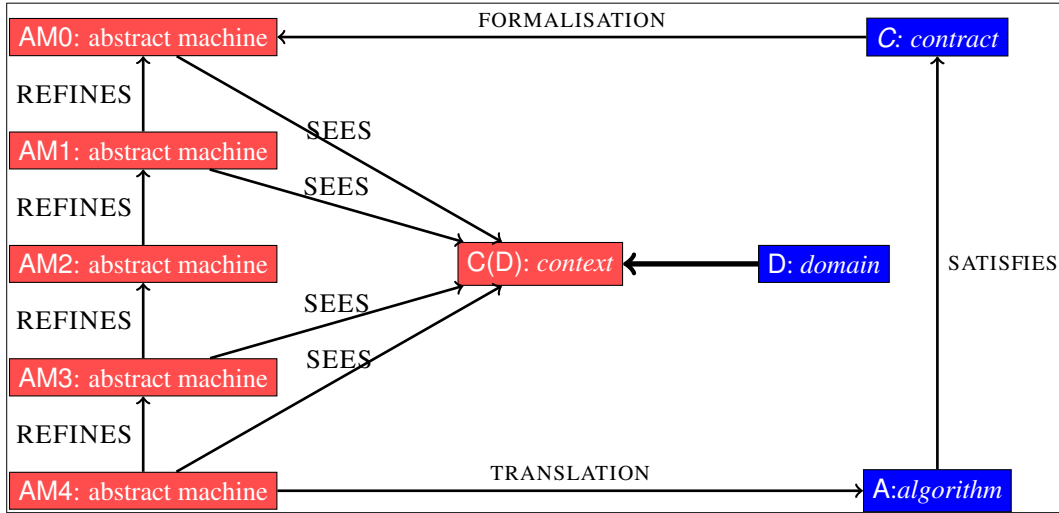
The development of correct algorithms or sequential programs from specifications (Dijkstra 1976) is a scientific theme linked to that of the verification of programs or algorithms (Turing 1949 ; Floyd 1967 ; Hoare 1969). program or algorithm verification turing49a,floyd67a,hoare69a. The fundamental question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

$$\forall v_0 \in D. \text{pre}(v_0) \Rightarrow \left( \begin{array}{l} \forall v_f \in D. v_0 \xrightarrow{A} v_f \Rightarrow \text{post}(v_0, v_f) \\ \exists v_f \in D. v_0 \xrightarrow{A} v_f \end{array} \right)$$

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

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

This discussion led us to give meaning to the correctness of an algorithm by considering its partial correctness as well as its termination. Hoare logic most often expresses partial correctness and in all rigour it would be necessary to use two notations, one expressing partial correctness and the other total correctness, but the objective here is not to verify an algorithm  $A$  and therefore to verify a list of verification conditions as Floyd method indicates, but to find an algorithm  $A$  which satisfies this equation  $\forall v_0 \in D. v = v_0 \wedge \text{pre}(v_0) \Rightarrow wp(A)(\text{post}(v_0, v))$ . The problem is therefore to construct an algorithm  $A$  enabling the contract  $C$  to be fulfilled in the domain  $D$ . In the *a posteriori* approach to correcting algorithms, we propose a solution for  $A$  and then apply the list of verification conditions. This technique also consists of applying the verification conditions without having clearly stated the contract  $C$ . Semantic analysis techniques can thus be developed with the abstract interpretation (Cousot 2021) and this is



**Figure 1.** Correctness by construction in Event-B

based on semantic techniques that simplify the life of the programmer who obtains analysis feedback. Correctness by construction is a technique which starts with an abstract algorithm  $A_0$  which fulfils the contract in a verified way and which is progressively enriched with increasingly complex control structures while observing the property of correctness with respect to the contract  $C$ . This progressive strategy of adding elements to make the result more precise is guided by the refinement relationship between the algorithms. Each transformation or refinement step guarantees that the resulting algorithm is correct. C. Morgan (Morgan 1990) develops the refinement calculus which makes it possible to progressively and correctly transform one algorithm into another algorithm by guaranteeing that the final algorithm is correct with respect to the first algorithm which is a pre/post specification considered as an algorithmic action of the form  $v : |(pre, post)$ .  $v : |(pre, post)$  designates an algorithmic instruction  $I$  whose effect is to modify  $v$  by respecting the contract defined by  $pre$  and  $post$ . Thus, the strategy consists of constructing a sequence of algorithms  $A_0, \dots, A_i, \dots, A_n$  with the properties:

- $A_0$  is the expression of the contract:  $D, A_0 \Rightarrow C$
- for all  $i$  in  $0..n - 1$ , the algorithm  $A_i$  refines  $A_{i-1}$ :  $D, A_i \Rightarrow C$  and  $D, A_{i-1} \Rightarrow C$ .
- $A_n$  is the algorithm satisfying the contract:  $D, A_n \Rightarrow C$ .

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

This approach to developing correct by construction algorithms is quite simply implemented in the Event-B language and in fact equipped by the Rodin environment. Figure 1 describes the general idea. This idea consists of translating the contract as a *event* which performs the calculation described by the contract. The contract expresses the *what* but carries out the calculation as an observation of the event. This event is placed in a first abstract machine  $AM_0$ , which uses the mathematical elements extracted from the problem domain  $D$  and expressed in the context Event-B  $C(D)$ . The development of an algorithm consists in the gradual enrichment of the extracted machines  $AM_0, \dots, AM_n$ , by expressing the computations necessary to systematically translate the last machine into an algorithmic form so as to guarantee the correctness of the  $A$  algorithm thanks to the correctness of the transformations provided by the refinement. It should be noted that the correctness concerns partial correctness and

termination, and that the abstract machines are models containing variables that will not be implemented in the algorithm produced.

We will present two techniques that implement this development pattern:

- the inductive pattern based on transformations of abstract machines by Jean-Raymond Abrial (Abrial 2010, Chapter 15).
- the recursive pattern based on the relation `call_textevent` that we have developed (Méry 2009 ; Cheng *et al.* 2016).

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

## 0.2. Design of a Iterative Sequential Algorithm

### 0.2.1. Problem 1 : Calculating the sum of a vector $v$ of integer values

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

<b>variables</b> $n, v, r$
<b>definitions</b>
$pre(n_0, v_0, r_0) \stackrel{def}{=} \begin{cases} n_0 \in \mathbb{N} \wedge n_0 \neq 0 \\ v_0 \in 1..n_0 \rightarrow \mathbb{Z} \\ r_0 \in \mathbb{Z} \wedge i_0 \in \mathbb{Z} \end{cases}$
<b>requires</b> $\begin{cases} n_0 \in \mathbb{N} \wedge n_0 \neq 0 \\ v_0 \in 1..n_0 \rightarrow \mathbb{Z} \end{cases}$
<b>ensures</b> $\begin{cases} r_f = \sum_{k=1}^{k=n_0} v_0(k) \\ n_f = n_0 \\ v_f = v_0 \end{cases}$

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

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

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

- $u$  is a total function of  $\mathbb{N}$  in  $\mathbb{Z}$  (axiom axm4).
- Initially, the summation starts with 0 and  $u(0) = 0$  (axiom axm5).
- For values of  $i$  less than  $n_0$ , the value of  $u(i)$  is defined from that of  $u(i - 1)$  and  $v_0(i)$  (axiom axm6).
- For all values greater than  $n_0$ , the value of  $u(i)$  is equal to that of  $u(n_0)$  (axiom axm7).

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

```

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

```

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

#### 0.2.1.1. Specification of the problem to solve

```

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

```

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

We can express it as a HOARE triple:

$$\{n = n_0 \wedge v = v_0 \wedge n_0 > 0 \wedge v_0 \in 1..n_0 \rightarrow \mathbb{N}\} \text{SUM} \{r = u(n_0)\}.$$

Note that the data is *visible* from the context *s0*. The problem is therefore to find an algorithm that calculates the value  $u(n)$  and stores it in  $r$ .

We have therefore described the domain of the problem to be solved and we have formulated what we want to calculate. The next step is to inventing a *method of calculation* and this requires a *idea of solution* and the use of refinement.

#### 0.2.1.2. Refining to compute inductively

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



```

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

```

```

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

```

```

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

```

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

#### 0.2.1.3. Focus on the value to be preserved

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

```

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

```

```

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

```

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

#### 0.2.1.4. Obtaining an algorithmic machine

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

- The event **final** is observed when the value of  $i$  is  $n$  and, in this case, the variable  $cu$  contains the value  $u(n)$ . The invariant guarantees that the value of  $cu$  is  $u(n)$ .
- The event **step4** is observed, when the value of  $i$  is less than  $n$ . This also means that, as long as this value is less than  $n$ , the event can be observed and the traces generated from these events therefore correspond to an iteration algorithmic structure.

```

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

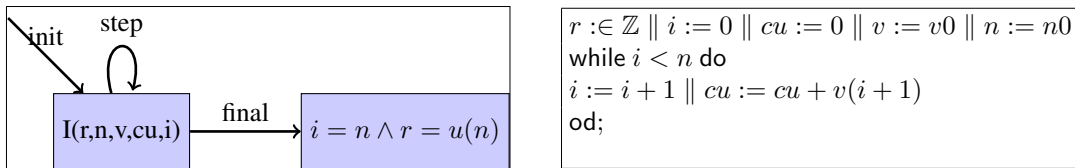
```

```

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

```

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



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

variables  $n, v, r$

definitions

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

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

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

int SUM(int n, v; int r)

variables

int r, i = 0, cu = 0, v = v0, n = n0;

while i < n do

cu := cu + v(i + 1);

i := i + 1;

od;

r := cu;

return(r);

#### 0.2.1.5. Comments on the methodology

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

#### 0.2.2. Transformations machines Event-B into sequential algorithms

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

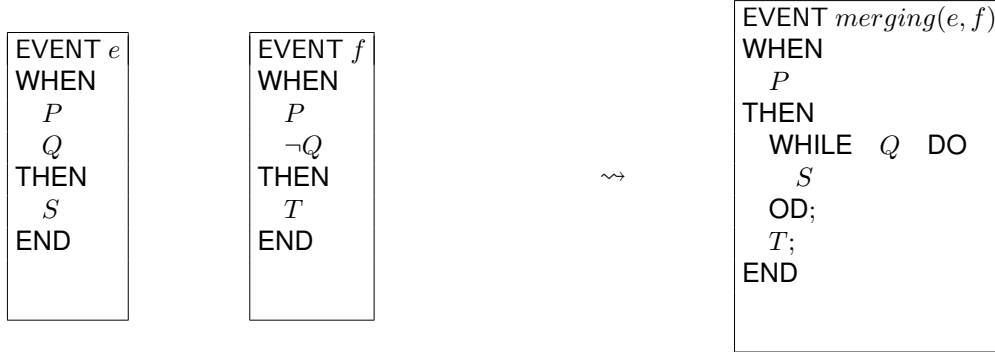
##### Definition 1 (The Pidgin Programming Language)

Statements of the language are:

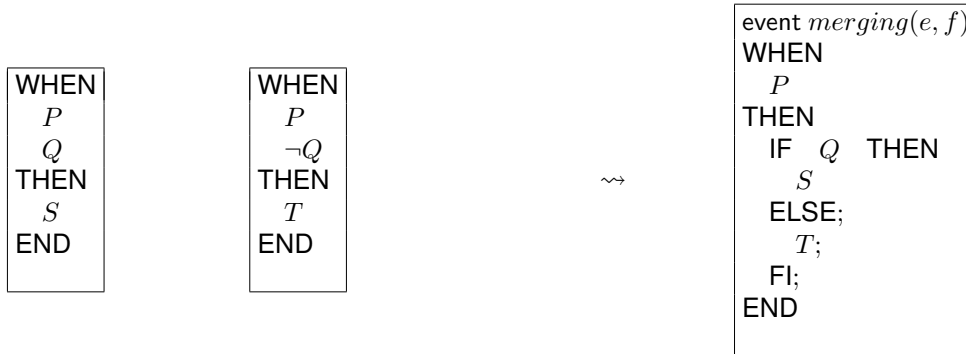
- *variable\_list* := *expression\_list*
- **statement**; **statement**
- **if** *condition* **then** *statement* **else** *statement* **end**
- **if** *condition* **then** *statement* **elseif** ... **else** *statement* **end**
- **while** *condition* **do** *statement* **end**

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

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



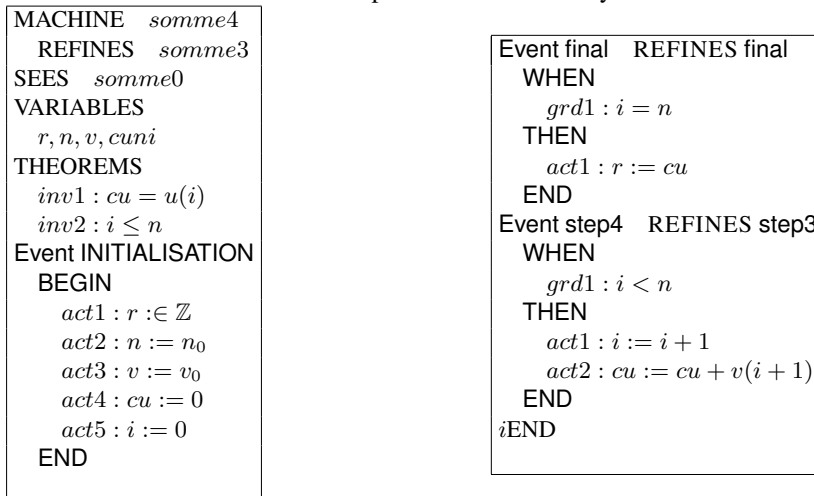
We must specify the conditions of application. The event  $e$  appears as new or non-anticipated, therefore convergent at a lower level of refinement than that of  $f$ . We can be sure that there is a variant which terminates the loop. We must also assume that  $P$  is invariant to the event  $e$ . The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

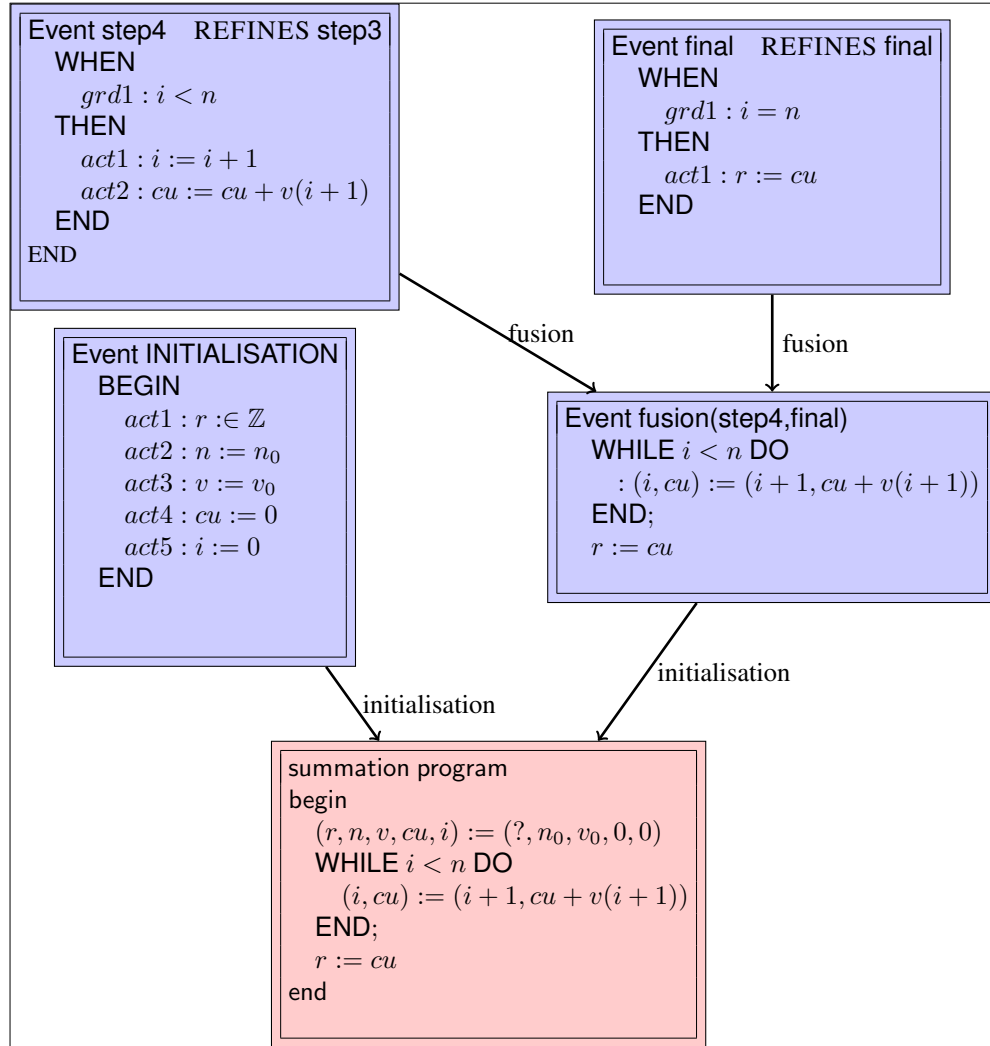


This transformation should be applied when the two events have been introduced at the same time. The event  $\text{merge}(e, f)$  must appear at the same level as the component. The guard  $P$  may not be present.

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

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





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

### 0.3. Examples of development

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

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

<b>variables</b> $x, r$
<b>definitions</b>
$pre(x_0, r_0) \stackrel{def}{=} \begin{cases} x_0 \in \mathbb{N} \\ r_0 \in \mathbb{Z} \end{cases}$
<b>requires</b> $\begin{pmatrix} n_0 \in \mathbb{N} \\ r_0 \in \mathbb{Z} \end{pmatrix}$
<b>ensures</b> $\begin{pmatrix} r_f = x_0 * x_0 * x_0 \\ x_f = x_0 \end{pmatrix}$

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

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

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

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

**Property 1** (Soundness of the sequence  $z$ )

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

This property guarantees that calculating the terms of the  $z$  sequence allows the value to be calculated using auxiliary sequences and only addition operations. These elements are given in the context **P3-0**. The contract can then be written in the form of a machine. **P3-1** contains a single event, **FINAL**, whose action is  $z := z(x_0)$ .

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

**Property 2** (Properties of sequences  $u, v, w, t$ )

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

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

$$\begin{aligned} inv1 : 0 \leq i \wedge i \leq x_0 \wedge cv = vv(i) \\ inv2 : cw = ww(i) \\ inv3 : cz = zz(i) \\ inv4 : ct = tt(i) \\ inv5 : cu = uu(i) \end{aligned}$$

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

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

Listing 1: ACSL power3.c

```

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

```

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

### 0.3.2. Problem 3: Searching a value in an array

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

variables  $t, n, x, r$

definitions

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

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

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

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

We will define the sequence  $u$  in the context of **S-0** and use the same methodology.

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

$$\begin{aligned}
& axm4 : u \in 0 \dots n \rightarrow \mathbb{Z} \times \text{BOOL} \\
& axm5 : u(0) = 0 \\
& axm6 : \forall i \cdot i \in 0 \dots n - 1 \wedge t(i+1) = x \wedge prj2(u(i)) = \text{FALSE} \Rightarrow u(i+1) = i+1 \mapsto \text{TRUE} \\
& axm7 : \forall i \cdot i \in 0 \dots n - 1 \wedge t(i+1) \neq x \Rightarrow u(i+1) = u(i) \\
& axm8 : \forall i \cdot i \in 0 \dots n - 1 \wedge t(i+1) = x \wedge prj2(u(i)) = \text{TRUE} \Rightarrow u(i+1) = u(i) \\
& th1 : \forall i \cdot i \in 1 \dots n \wedge prj2(u(i)) = \text{FALSE} \Rightarrow (\forall k \cdot k \in 1 \dots i \Rightarrow t(k) \neq x) \\
& th2 : \forall i \cdot i \in 1 \dots n \wedge prj2(u(i)) = \text{TRUE} \Rightarrow (\exists k \cdot k \in 1 \dots i \Rightarrow t(k) = x) \\
& th3 : \forall i \cdot i \in 1 \dots n \wedge prj2(u(i)) = \text{TRUE} \Rightarrow (t(prj1(u(i))) = x)
\end{aligned}$$

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

```

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

```

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

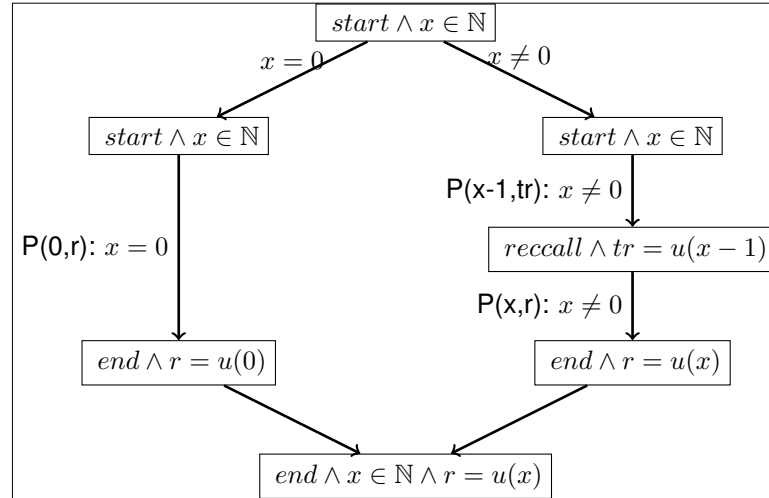
### 0.3.3. Problem 4: Computing a primitive recursive function

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

<p><b>variables</b> <math>x, n, r</math></p> <hr/> <p><b>services</b></p> <p><math>H \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}</math></p> <p><math>G \in \mathbb{N} \rightarrow \mathbb{N}</math></p> <p><math>F \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}</math></p> <p><math>F = \text{PRIMREC}(G, H)</math></p> <hr/> <p><b>requires</b> <math>\left( \begin{array}{l} x_0 \in \mathbb{N} \\ n_0 \in \mathbb{N} \end{array} \right.</math></p> <p><b>ensures</b> <math>\left( \begin{array}{l} x_f = x_0 \wedge n_f = n_0 \\ r_f = F(x_0, n_0) \end{array} \right.</math></p> <hr/>	<p>Le problème est de calculer la fonction <math>F</math> définie à l'aide de deux autres fonctions <math>G</math> et <math>H</math> qui sont calculées par des algorithmes. <math>F</math> est définie par l'équation suivante: pour toute valeur naturelle <math>x, y</math>,</p> $ \begin{cases} F(x, 0) = G(x) \\ F(x; y+1) = H(x, y, F(x, y)) \end{cases} $ <p>The function <math>F</math> is defined using <math>G</math> and <math>H</math>, which are themselves defined by the same calculability schemes. This function <math>F</math> is itself a sequence which is specialised in <math>v</math>. We obtain the following algorithm.</p> <div style="border: 1px solid black; padding: 10px; margin-top: 10px;"> <pre> r := G(x)    k := 0    cv := G(x) while k &lt; x do   k := k+1    cv := H(x, k, cv) od; </pre> </div>
--	--

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





**Figure 2.** Organisation of the computation in a recursive solution using assertion diagram

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

#### 0.4. Design of a Recursive Sequential Algorithm

In this section, we will apply the simple idea of analysing the diagram in figure 1 to develop a recursive program or algorithm. We will also promote one-shot refinement. In the previous section, we refined as long as necessary, especially as long as we obtained a set of events corresponding to computable elements. We have produced the EB2RC plugin (Cheng *et al.* 2016), which implements this idea.

##### 0.4.1. The “Call as Event” Idea

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

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

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

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

```

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

```

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

```

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

```

```

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

```

```

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

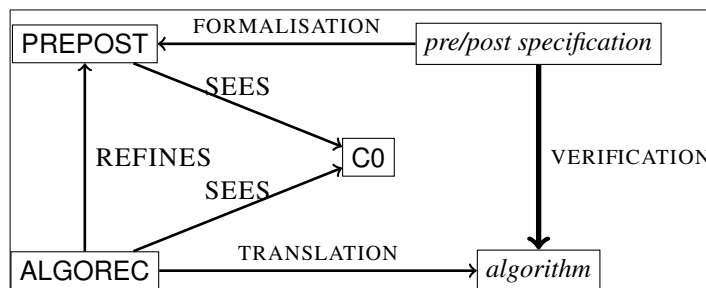
```

The machine is simulating the organisation of the computations following two cases according to the figure 2. The first case is the path on the left part of the diagram and is when  $x$  is 0 and the second case if when  $x$  is not 0.

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

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

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



**Figure 3.** The recursive pattern

The recursive pattern is linked to a diagram which is helping to structure the solution. We have labelled arrows by guards or by events. The diagram helps to structure the analysis based on the inductive definitions. Following

this pattern, we have developed the ERB2RC plugin based on the identification of three possible events. When a pre/post specification is stated, the program to build can be expressed by a simple event expressing the relationship between input and output and it provides a way to express pre/post specification as events. The first model is a very abstract model containing the pre/post events.

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

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

```

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

```

```

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

```

```

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

```

#### 0.4.2. Applying the call as event technique

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

##### 0.4.2.1. Problem 1: Computing the power 2 ( $\lambda x.x^2$ )

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

```

MACHINE square // square(n; r)
  REFINES specsquare SEES control0

VARIABLES r l tr

INVARIANTS
  @inv1 r ∈ ℕ
  @inv2 l = end ⇒ r = n * n
  @inv3 l = call ⇒ n ≠ 0
  @inv4 l = call ⇒ tr = (n - 1) * (n - 1)
  @inv5 l ∈ C
  @inv6 tr ∈ ℕ
  @inv7 l = end ⇒ r = n * n
  @inv8 l = end ∧ n ≠ 0
    ⇒ tr = (n - 1) * (n - 1) ∧ r = tr + 2 * (n - 1) + 1
  theorem @inv9 l = call ⇒ n * n = tr + 2 * (n - 1) + 1

```

```

EVENTS

EVENT INITIALISATION
  then
    @act1 r := 0
    @act2 l := start
    @act3 tr := 0
  end

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

```

```

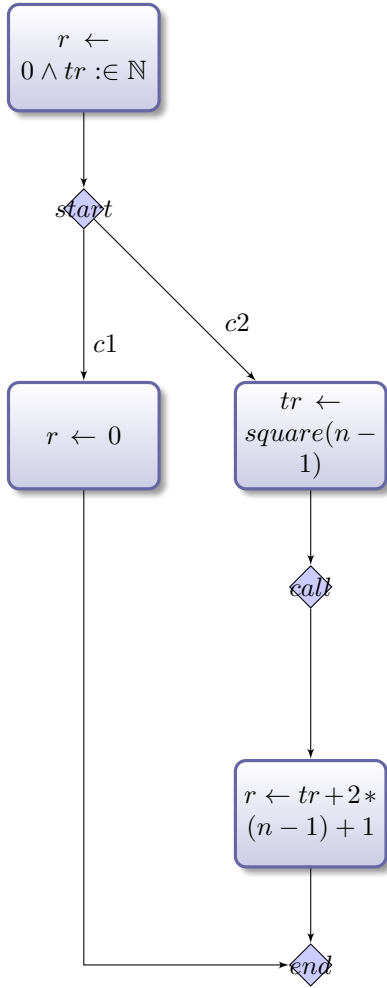
EVENT square(n; r)@n / = 0 REFINES square(n; r)
  where
    @grd1 l = call
    then
      @act1 r := tr + 2 * (n - 1) + 1
      @act2 l := end
    end

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

```

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

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



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

**c1**  $n = 0$

**c2**  $n \neq 0$

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

```

procedure square(n; r)
begin
  r ← 0
  tr :∈ ℕ
  if n = 0
    r ← 0
  elsif n ≠ 0
    tr ← square(n - 1)
    r ← tr + 2 * (n - 1) + 1
  endif
end
  
```

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

```

MACHINE square // square(n; r)
REFINES specsquare SEES control0

VARIABLES r l tr

INVARIANTS
@inv1 r ∈ ℕ
@inv2 l = end ⇒ r = n * n
@inv3 l = call ⇒ n ≠ 0
@inv4 l = call ⇒ tr = (n - 1) * (n - 1)
@inv5 l ∈ C
@inv6 tr ∈ ℕ
@inv7 l = end ⇒ r = n * n
@inv8 l = end ∧ n ≠ 0
    ⇒ tr = (n - 1) * (n - 1) ∧ r = tr + 2 * (n - 1) + 1
theorem @inv9 l = call ⇒ n * n = tr + 2 * (n - 1) + 1
  
```

#### 0.4.2.2. Problem 2: Binary search in an array

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

```

contract binsearch(t, val, lo, hi, ok, result)
requires  $\left( \begin{array}{l} t \in 0..t.Length \rightarrow \mathbb{N} \\ \forall k. k \in lo..hi - 1 \Rightarrow t(k) \leq t(k+1) \\ val \in \mathbb{N} \\ l, h \in 0..t.Length \\ lo \leq hi \end{array} \right)$ 
ensures  $\left( \begin{array}{l} ok = TRUE \Rightarrow t(result) = val \\ ok = FALSE \Rightarrow (\forall i. i \in lo..hi \Rightarrow t(i) \neq val) \end{array} \right)$ 

```

```

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

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

```

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

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

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

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

These two events form the machine called *binsearch1* which is refined to obtain *binsearch2* (corresponding to PROCESS of Figure 7). In addition to these events, the events of this refined machine contains a new control label,  $l$ , which *simulates* how the binary search is achieved.

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

```

MACHINE binsearch2 REFINES binsearch1 SEES binsearch0

```

```

VARIABLES i ok l mi

```

```

INVARIANTS

```

```

@inv1 i ∈ 1..n

```

```

@inv2 l ∈ LOC

```

```

@inv3 dom(t) = 1..n

```

```

@inv4 mi ∈ 1..n

```

```

@inv5 l = middle  $\Rightarrow lo < hi \wedge mi \in lo..hi$ 

```

```

@inv6 l = middle  $\wedge key < t(mi) \Rightarrow (\forall k. k \in mi..hi \Rightarrow t(k) \neq key)$ 

```

```

@inv7 l = middle  $\wedge key > t(mi) \Rightarrow (\forall k. k \in lo..mi \Rightarrow t(k) \neq key)$ 

```

```

@inv8 l = end  $\wedge ok = TRUE \Rightarrow i \in lo..hi \wedge t(i) = key$ 

```

```

@inv9 l = end  $\wedge ok = FALSE \Rightarrow (\forall k. k \in lo..hi \Rightarrow t(k) \neq key)$ 

```

```

theorem @inv10 lo..hi ⊆ 1..n

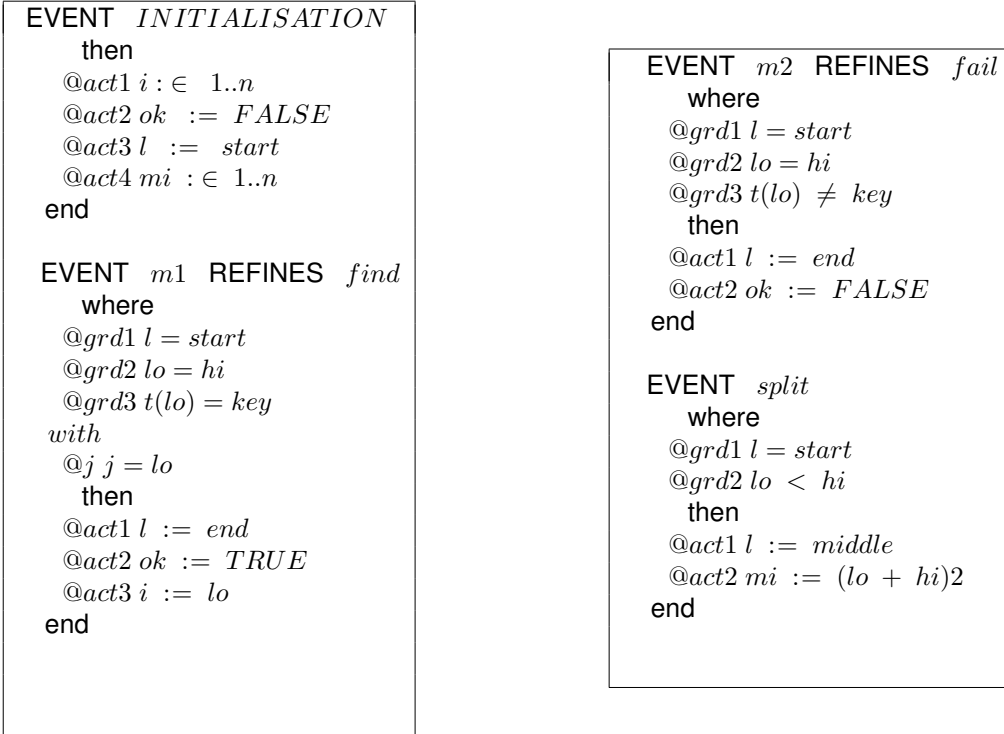
```

```

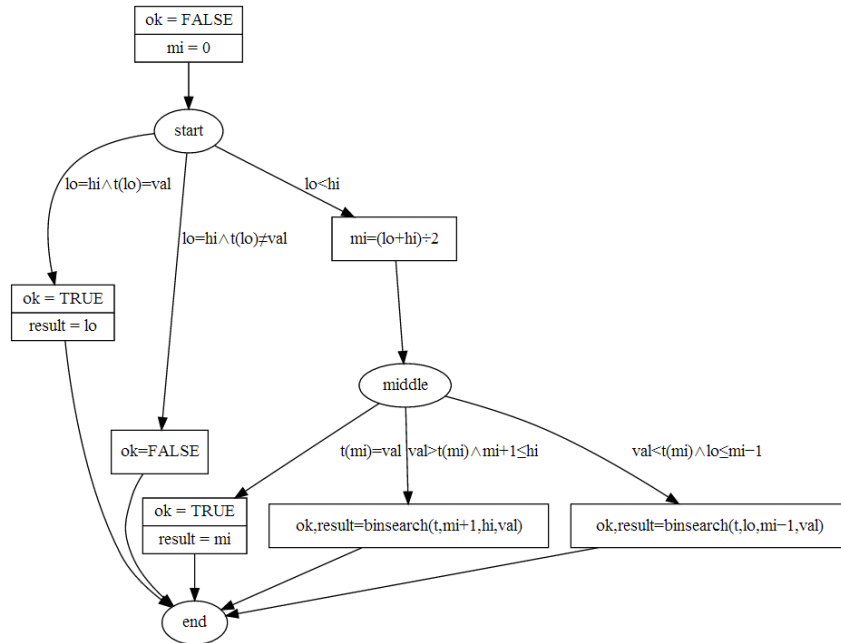
theorem @inv11  $(\exists j. l = middle \wedge j \in mi+1..hi \wedge key > t(mi) \wedge t(j) = key \wedge mi+1 \leq hi) \Rightarrow (mi+1 \leq hi)$ 

```

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



The *split* event directs the analysis and divides the exploration space into three parts of the indexes. Figure 0.4.2.2, the *m3* event considers the case where the *mi* index is the one where the searched value is found. Then the other two events correspond to the segment between *mi*+1 and *hi* and are in fact recursive calls to the procedure under construction. These two events refine *find* and *fail* respectively. We need to add two more events corresponding to the segment *lo*, *mi* - 1 segment to complete the model.



**Figure 4.** Visualized Representation of the Binary Search Algorithm

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

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

**Table 1.** Proof effort of our refinement approach for the binary search case study

The proof effort of our refinement approach for the Binary Search case study is illustrated in Table 1. The first abstract model is proved automatically and the second concrete model is automatic in 78 % of its proof obligations.

The integrated development framework takes this into consideration. As shown in Fig. 7, we suggest to translate every recursive algorithm ALGORITHM into a partially annotated and iterative OPTIMISED ALGORITHM to be verified within the Spec# Programming System. In (Méry and Monahan 2013), we have proposed and proved a sound translation procedure from ALGORITHM to OPTIMISED ALGORITHM to perform this task. For example, the iterative version of the binary search algorithm in Spec# is shown in Fig. 5.

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

```

Recursive Algorithm binsearch(t,lo,hi,val;ok,result) generated by EB2RC
begin
ok := FALSE; mi := 0;
if lo = hi ∧ t(lo) = val then
  ok := TRUE;
  result := lo;
elseif lo = hi ∧ t(lo) ≠ val then
  ok := FALSE;
elseif lo < hi then
  mi := (lo + hi) ÷ 2;
  if t(mi) = val then
    ok := TRUE;
    result := mi;
  elseif val > t(mi) ∧ mi + 1 ≤ hi then
    ok, result := binsearch(t, mi + 1, hi, val);
  elseif val < t(mi) ∧ lo ≤ mi - 1 then
    ok, result := binsearch(t, lo, mi - 1, val);
end

```

#### 0.4.3. Comments on the call as event idea

In the example of subsection 0.4.2.1, we do not use the event like `call%APROC(h(x),y)%P(y)` but the event is clearly a call for another procedure or function. For instance, when a sorting algorithm is developed, you may need an auxiliary operation for scanning a list of values to get the index of the minimum. It means that we have a way to define a library of models and to use correct-by-construction procedures or functions. In (Cheng *et al.* 2016), we detail the tool and the way to define a library of *correct-by-construction programs*. The EB2RC plugin is used on this project and we obtain two files: one containing the algorithm and another containing the diagram built from the Event-B model.

#### 0.5. Final comments

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



```

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

```

**Figure 5.** Binary Search C# program corresponding to the generated iterative procedure.

value of the square or the cube. terms is the value of the square or cube. Insofar as refinement is a very general relationship on a set of very general models too, we must try to set ourselves a refinement discipline. The first technique invites us to play with the model variables and to reduce the variables to those which are exclusively used for the calculation. In this way, we can better understand the notions of model variable or ghost variable used in certain proof tools which look for these variables. In our case, we introduce them and then hide them in the abstract models. in the abstract models and they make it easier to prove and state invariants. The second technique is simpler because it relies on inductive definitions that are interpreted in the recursive paradigm. It's about saving refinement and developing at a step of refinement. The resulting program is recursive and the recursiveity should be deleted and it is relatively easy to produce using our event conventions. In this case, we noted that the proofs were relatively simpler to derive. Both techniques rely on experimental plugins but could be combined. Figure 7 illustrates an environment for co-ordinating the various techniques for relatively complex sequential systems, but it is necessary to develop a new formal IDE integrating these techniques and close to the diagram in figure. 7 . Naturally, the development of concurrent, parallel or distributed algorithms or programs is a challenge to be taken up, and the chapter entitled Design of Correct by Construction Distributed Algorithms will provide some ideas.

## 0.6. Bibliography

- Abrial, J.-R. (2010), *Modeling in Event-B: System and Software Engineering*, Cambridge University Press.
- Chandy, K. M., Misra, J. (1988), *Parallel Program Design A Foundation*, Addison-Wesley Publishing Company. ISBN 0-201-05866-9.
- Cheng, Z., Méry, D., Monahan, R. (2016), On two friends for getting correct programs - automatically translating event B specifications to recursive algorithms in rodin, in T. Margaria, B. Steffen, (eds), *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISO LA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, vol. 9952 of *Lecture Notes in Computer Science*, pp. 821–838.  
**URL:** [https://doi.org/10.1007/978-3-319-47166-2\\_57](https://doi.org/10.1007/978-3-319-47166-2_57)
- Cousot, P. (2021), *Abstract Interpretation*, The MIT Press. ISBN: 9780262044905.
- Dijkstra, E. W. (1976), *A Discipline of Programming*, Prentice-Hall.
- Floyd, R. W. (1967), Assigning meanings to programs, in J. T. Schwartz, (ed.), *Proc. Symp. Appl. Math.* 19, Mathematical Aspects of Computer Science, American Mathematical Society, pp. 19 – 32.
- Hoare, C. A. R. (1969), An axiomatic basis for computer programming, *Communications of the Association for Computing Machinery*, 12, 576–580.

```

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

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

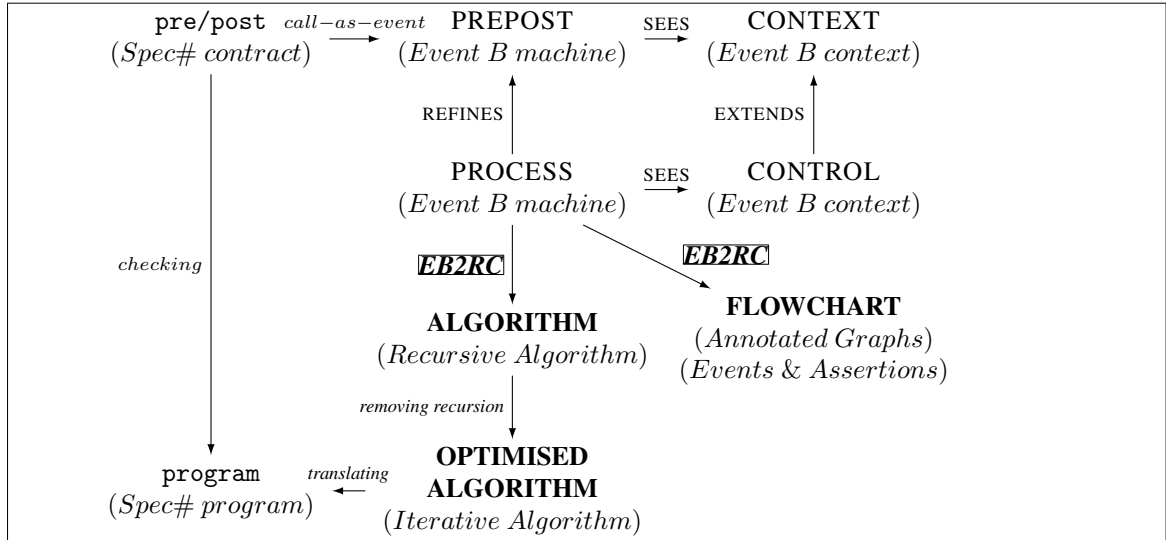
EVENT rec@search(t, val, mi + 1, hi, ok, result)@ok = FALSE REFINES fail
  where
    @grd1 l = middle
    @grd2 key > t(mi)
    @grd4 ∀ j j ∈ mi + 1..hi ⇒ t(j) ≠ key
    @grd5 mi + 1 ≤ hi
  then
    @act3 l := end
    @act2 ok := FALSE
  end
end

```

fig:

**Figure 6.** Events for recursive calls

- Kourie, D. G., Watson, B. W. (2012), *The Correctness-by-Construction Approach to Programming*, Springer.  
**URL:** <https://doi.org/10.1007/978-3-642-27919-5>
- Méry, D. (2009), Refinement-based guidelines for algorithmic systems, *Int. J. Softw. Informatics*, 3(2-3), 197–239.  
**URL:** [http://www.ijsi.org/ch/reader/view\\_abstract.aspx?file\\_no=197&flag=1](http://www.ijsi.org/ch/reader/view_abstract.aspx?file_no=197&flag=1)
- Méry, D., Monahan, R. (2013), Transforming event B models into verified c# implementations, in A. Lisitsa, A. P. Nemytykh, (eds), *First International Workshop on Verification and Program Transformation, VPT 2013*, Saint Petersburg, Russia, July 12-13, 2013, vol. 16 of *EPiC Series in Computing*, EasyChair, pp. 57–73.  
**URL:** <https://doi.org/10.29007/9wm9>
- Morgan, C. (1990), *Programming from Specifications*, Prentice Hall International Series in Computer Science, Prentice Hall.
- Owicki, S. S., Lamport, L. (1982), Proving liveness properties of concurrent programs, *ACM Trans. Program. Lang. Syst.*, 4(3), 455–495.
- Rogers, H. J. (1967), *Theory of Recursive Functions and Effective Computability*, The MIT Press.
- Singh, N. K. (2024), EB2ALGO. Plugin Rodin.



**Figure 7.** An overview of our integrated development framework to combine program refinement with program verification ((Cheng et al. 2016))

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