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

Dominique MÉRY

September 27, 2024

Contents

Design of Correct by Construction Sequential Algorithms

Dominique MÃ©RY[1]
[1] *LORIA, Telecom Nancy & Université de Lorraine*

0.1. Design of a Recursive Sequential Algorithm

In this section, we will apply the simple idea of analysing the diagram in figure **??** 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.

0.1.1. *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 $P(x, r)$.

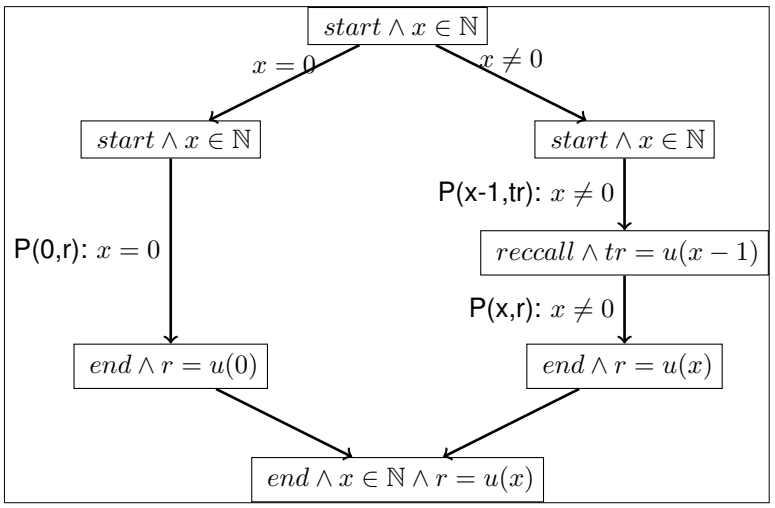


Figure 1. *Organisation of the computation in a recursive solution using assertion diagram*

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

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

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

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

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

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

The machine is simulating the organisation of the computations following two cases according to the figure 1. The first case is the path on the left part of the diagram and is when x is 0 and the second case if when x is not 0.

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

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

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

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

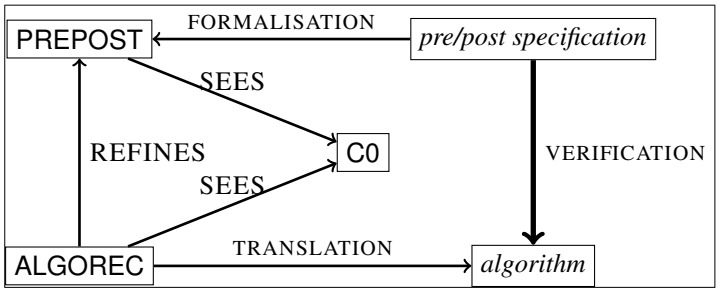


Figure 2. *The recursive pattern*

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

0.1.2. *Applying the call as event technique*

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

0.1.2.1. *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. 1 is the diagram analysing the way to solve the computation of the value of $u(x)$ following the call-as-event paradigm.

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

VARIABLES $r\ l\ tr$

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

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

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

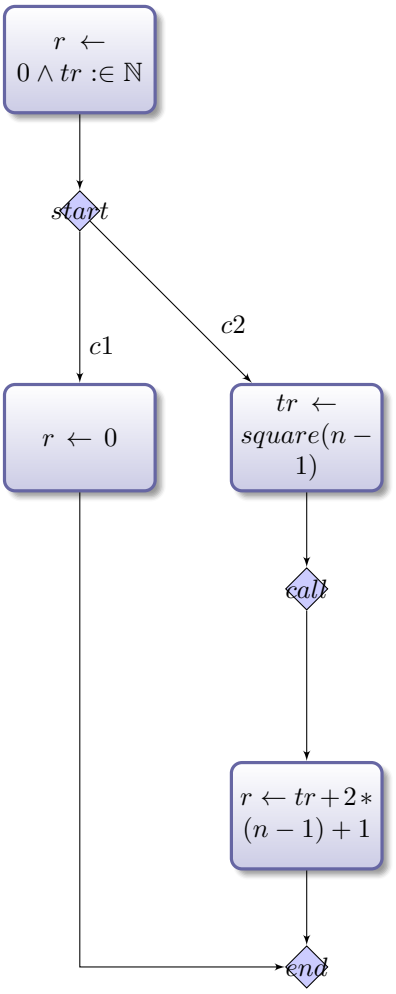
EVENT $square(n;r)@n/ = 0$ REFINES $square(n;r)$
where
$@grd1\ l = call$
then
$@act1\ r := tr + 2 * (n-1) + 1$
$@act2\ l := end$
end

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

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

Proofs are simpler and invariants are easier to extract from the inductive definitions. The use of Frama-C shows that the proofs are also very simple in the case of a recursive algorithm. Missing expertise in using Frama-C leads to the introduction of auxiliary lemmas as $((x-1)+1)^2 = (x-1)^2 + 2x + 1$. In this example, 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.



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.

Algorithm 1 Algorithm: bin

```
1: r ← 0
2: tr :∈ ℕ
3: if n = 0 then
4:     r ← 0
5: else if n ≠ 0 then
6:     tr ← square(n − 1)
7:     r ← tr + 2 ∗ (n − 1) + 1
8: end if
```

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

VARIABLES $r\ l\ tr$

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

0.2. Bibliography

Cheng, Z., Méry, D., Monahan, R. (2016), On two friends for getting correct programs - automatically translating event B specifications to recursive algorithms in rodin, *in* T. Margaria, B. Steffen, (eds), Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I, vol. 9952 of *Lecture Notes in*

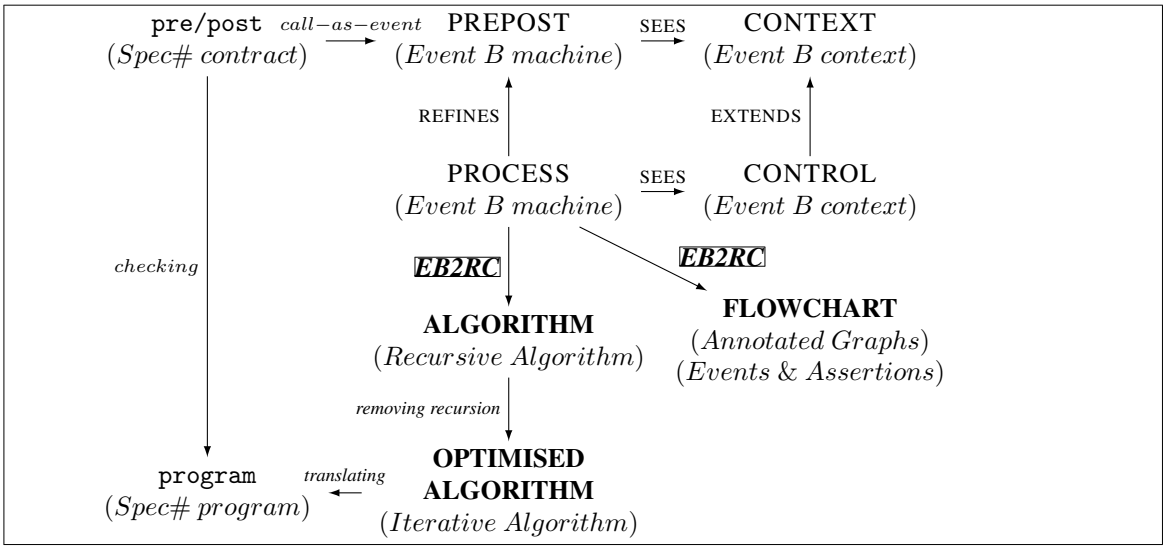


Figure 3. *An overview of our integrated development framework to combine program refinement with program verification ((Cheng* et al*. 2016))*

Computer Science, pp. 821–838.
URL: *https://doi.org/10.1007/978-3-319-47166-2_57*

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